Vrije Universiteit Brussel
Faculteit der Toegepaste Wetenschappen
Departement ELEC
Pleinlaan 2, 1050 Brussel, Belgium

Caching of Intermediate Results in Dataflow Environments

Eli Steenput

Promotoren:
Prof. Dr. ir. J. Schoukens
Prof. Dr. ir. Y. Rolain

Proefschrift ingediend tot het behalen van de academische graad van doctor in de Toegepaste Wetenschappen

Augustus 1999
I would like to express my thanks to the following people:

Paula K. Austel from the IBM Thomas J. Watson Research Center for discussing research on benchmarks for cache hits within IBM.

Greg Goebel from HP-MXD for explaining some details of HP VEE propagation.

Donna L. Gresh from IBM Data Explorer Software Development for explaining the state of research on the effectiveness of their caching methods at IBM.

Jeff Kodosky from National Instruments for discussing LabView and pointing out some potential problems with low-level IO and caching.

Phil McLaughlin from The Bloomsbury Software Company for his explanation on VisualAge Technology.

Edward A. Lee from the University of California for his helpful discussion on Ptolemy, multiple models of computation, and determinism, and his affirmation that fine grain dataflow will benefit most from changing the graph scheduling scheme.

Gilles Mauris from the Universite de Savoie for his comments on temporal coherence in sensor networks.

Emil M. Petriu for his encouraging discussion on virtual measurement systems.

Rik Pintelon for suggesting improvements to the mathematical cost and overhead calculations, and for proof-reading my manuscript.

Yves Rolain for his tireless assistance and support when things looked unfavourable.

Johan Schoukens for making this work possible.

Hans Spoelder for drawing attention to the caching scheme used in Data Explorer, and for sending me Frank Rijnders’ thesis.

Perry Steger from National Instruments Lookout R&D for his information on Lookout.

Dick Stevens from the Naval Research Laboratory (NRL) in Washington, DC for his information on the Processing Graph Method and the PGM Tool.

Lloyd A. Treinish from the IBM Thomas J. Watson Research Center for his suggestion of dataflow papers.

Allison Woodruff from the University of California for sharing information on their database search pattern simulations.
Contents

1. INTRODUCTION ........................................................................................................... 1
   1.1 THE MEASUREMENT ENVIRONMENT SITUATION ...................................................... 1
   1.2 REASONS FOR CACHING INTERMEDIATE RESULTS ............................................... 2
   1.3 USING DATA DEPENDENCY INFORMATION ................................................................. 3
   1.4 THIS WORK ................................................................................................................ 4

2. DATAFLOW ENVIRONMENTS ................................................................................. 6
   2.1 VIRTUAL INSTRUMENTS AND VISUAL ENVIRONMENTS ....................................... 7
   2.2 DATAFLOW GRAPHS .................................................................................................. 9
      2.2.1 Basic concepts ......................................................................................................... 9
      2.2.2 Constructing dataflow graphs from basic code blocks ......................................... 11
      2.2.3 Algebraic graph optimisation ............................................................................... 12
      2.2.4 Summary ............................................................................................................... 12
   2.3 DATAFLOW EXECUTION SCHEMES ......................................................................... 13
      2.3.1 Data driven Dataflow ............................................................................................ 13
      2.3.2 Demand driven Dataflow ..................................................................................... 15
   2.4 DATAFLOW EXTENSIONS ......................................................................................... 18
      2.4.1 Overview ................................................................................................................ 18
      2.4.2 Procedures ............................................................................................................. 19
      2.4.3 Sequential Execution ............................................................................................ 20
      2.4.4 Conditional Execution .......................................................................................... 21
      2.4.5 Iteration ................................................................................................................ 23
      2.4.6 A few words on parallelism ................................................................................... 26
      2.4.7 The effects of dataflow extensions on the execution scheme ................................ 27

3. CACHING OF INTERMEDIATE RESULTS IN DATAFLOW ENVIRONMENTS ................. 29
   3.1 CACHING IN DATA DRIVEN DATAFLOW ................................................................. 30
      3.1.1 Threads ................................................................................................................... 31
      3.1.2 Some observations ............................................................................................... 33
   3.2 CACHING IN DEMAND DRIVEN DATAFLOW ........................................................... 34
      3.2.1 Difference from data driven dataflow .................................................................. 34
      3.2.2 Time stamps ......................................................................................................... 35
1. Introduction

1.1 The measurement environment situation

In a measurement and modelling environment, the user is often directly involved in the
design and implementation of the software needed to automate his tests or experiments.
Even when external programmers are available, it may be difficult and time consuming for
the user to convey the necessary details of the problem, unless the programmer is well
versed in the measurement discipline [1, 2].

Measurement software is generally a mix of control instructions to manipulate the
measurement equipment, and numerical algorithms to process and display the
measurement results (Figure 1). Implementation can involve writing large and complex
programs that access multiple data sets and utilise numerous different statistical, numerical
and graphical processing routines and instrumentation drivers. Software development time
can take on the order of many months to years of effort.

As the tests and experiments in a measurement environment are continually changing,
the programming time and its cost become important concerns for the measurement
community. For this reason, techniques that would allow a reduction in the development
time and effort deserve attention.

The continual increase in computer performance has not yet rendered attempts to
increase program efficiency obsolete. Application complexity has so far kept up with...
ongoing hardware improvements. In the measurement field, the increased computing power has been allocated mostly to the reduction of expertise required from the users. Where in the past the selection of excitation signals, models and model order for example was carefully deliberated in advance, today’s brute-force approach allows multiple options to be executed and compared, often in largely automatic fashion.

While it may be argued that a consistent and correct application of software engineering principles (whichever ones are popular at the time) could easily handle the increasing complexity of measurement applications, experience shows that software expertise is precisely what is distinctly lacking among most measurement system developers, who are largely unprepared for the often unexpected expansion of the software involved with their systems under development. The developers need environments that not only support, but actually assist structured programming.

### 1.2 Reasons for caching intermediate results

Once the initial code for a measurement and identification system is drafted, the developer will have to run multiple experiments to verify the program. During this verification and debugging phase, the developer will repeatedly make changes, not only to the code, but also to the settings for the measurement, for the data processing and for its representation, so as to observe their effect on the final result. Repeated execution with minor changes to the settings will continue during the operational use of the system. Both the measurement itself and the processing of the measured data are often very time-consuming. Since the developer usually wants to view the effects of only a few changes at a time, some of the time-consuming operations will be executed more often than necessary during these repeated executions. The resulting idle time is frustrating to the developer. The continued increase in computer performance has done nothing to reduce the number of repeated executions.

When the argument values of an operation are unchanged since the previous execution of the operation, repeating the operation is redundant, since its recalculation will reproduce exactly the same result. To avoid such redundant executions, the previous arguments of an operation and the corresponding results can be stored in a data cache. When the current argument values are found in the cache, the corresponding result can be retrieved instead of being recalculated (Figure 2).

![Figure 2](image.png)

**Figure 2** A cache is required to avoid redundant operations

Storing an operation’s previous results in a table indexed by previous argument values is proposed in [3] and has reportedly been used successfully in various applications [4, 5, 6]. An example of caching intermediate results to avoid repeated execution of identical operations can be found in the Maple mathematical software [5]. The Maple remember option associates with each function a table of previous arguments and corresponding
results. At each function call, the table is checked for the presence of the given combination of argument values. If the arguments are already in the table, the corresponding result is retrieved from the table. Otherwise, the function is executed (with the new arguments), and a new entry is made in the table for the new result and its arguments. The least frequently used entries can be discarded if the available space is limited.

If sufficient memory is available, this scheme will reduce the number of operations to a nearly optimum level, as repetition of identical operations in the algorithm is automatically avoided as well. Option remember allows the coding of a function (for example, one with a recursive definition) in a sometimes more natural manner without loss of efficiency. A naive DFT algorithm, for example, will run like an FFT because the duplicate operations are executed only once.

A remember table allows the reuse of results from different loop iterations or recursive function calls because all invocations of a certain procedure or function, from anywhere in the program, are grouped in a single table. However, this advantage is only significant when identical calculations are repeated in several recursive or iteration steps (in a well-written program this should rarely be the case).

A disadvantage of the remember table concept in Maple is that a procedure is treated the same way no matter in which context it is invoked. This may be acceptable in a symbolic algebra environment, but in a measurement application, not all operations are side-effect free. The remember option in the Maple implementation is specified in a procedure’s definition and thus it cannot be turned off for selected invocations of the procedure. However, it is not impossible to imagine an implementation where caching could be turned off for selected invocations. Another possible disadvantage is that the memory used for caching is centralised and thus a remember table is less easily applicable in some forms of distributed computing.

Remember table lookup will create some overhead, and available memory will limit the number of entries in the cache, especially when large matrices are involved. The lookup or comparison overhead, which is inherent to the caching scheme, will increase with the number of previous results stored in the cache. Because data can be quite large in a measurement or modelling environment, caching all intermediate results is not always feasible.

1.3 Using data dependency information

This study proposes to use data dependency information to reduce the necessary cache size and cache lookup overhead. Associating a cache with each node in the dataflow graph of the program will have several advantages. Because each instance of a function has a separate cache, the number of entries in each cache is small, and hence the individual cache size and lookup overhead are reduced. Additionally, the validity of a node’s cache can be derived from the validity of the arguments of the node, which eliminates the need to look up arguments in a table.

Dataflow information can also be used to decide which nodes are most likely to profit from caching, and how to reduce the number of cached values without excessively compromising performance.
To integrate dataflow information in a caching scheme, this information must be available to the environment. Dataflow-based environments have this information readily available. Dataflow environments offer a highly structured approach to system development, closely linked to the field of virtual instrumentation.

1.4 This work

This work is structured as follows:

Chapter 2 is a discussion of the basic properties of dataflow-based virtual instrumentation environments, starting with their supposed ease of programming. Proponents of visual programming often praise its ease of programming, but this praise does not really reflect the actual experience of visual programming. Considering literature of various user interaction studies, this section concludes that “visual” programming is not, as usually stated, fundamentally easier to program. What visual programs offer is a closer mapping between tasks and program primitives than is possible in any textual program. This allows them to be used as a design tool, but one in which the completed design will actually be an executable program. This is especially useful for users who have to design and develop their own software, while lacking software engineering expertise.

Next follows a description of dataflow graphs and of dataflow based programming environments. In dataflow based programming, a dataflow graph becomes an executable program by adding a dataflow execution scheme. Data driven and demand driven execution are presented. Both execution schemes have proponents and detractors. This author proposes that comparisons of both methods are not always fair. Comparing execution schemes is rather akin to comparing different programming languages.

Dataflow extensions seem to occur in almost infinite variety. Various classifications can be found in the literature [24]. This section does not present yet another classification system, but rather attempts an overview (which is by no means thorough in either depth or breadth) of some common extensions and their similarities and differences. This overview was substantially inspired by [11]. It is further proposed that the data driven / demand driven controversy loses most of its meaning once dataflow extensions are introduced.

Chapter 3 introduces caching schemes that can utilise the data dependency information available, and discusses how they can be implemented in the data driven and demand driven scheme. The concept of threads, and how they allow a reduction of the required cache space, is introduced in this chapter (the term is loosely based on terminology used in [39] for clusters of sequential code that are asynchronously scheduled in parallel computers).

The concept of time tags and input terminal sets proposed here for demand driven execution was suggested in [7], it also bears some similarity to UNIX’ Make command, and the update mechanism of some spreadsheet implementations.

This chapter introduces three different caching schemes for both dataflow execution methods (data driven and demand driven), each reducing caching overhead and memory requirements to a different extent, at the expense of a possible reduction of caching effectiveness.
It is observed that non-dataflow based environments can also collect data dependency information to make caching more efficient (as in Maple, though the documentation [5] does not identify the adopted methods as such).

The chapter concludes with a demonstration of how caching could be handled manually, how this unavoidably results in an unreadable program, and why caching should be handled automatically.

In chapter 4, a broad evaluation of the performance of the presented caching schemes, averaged for a large number of graphs and as a function of generic graph properties, is presented. The lack of a library of suitable characteristic example algorithms for analysis posed a problem. Studying caching performance on classic benchmark algorithms was considered, but most benchmarks were unsuitable for measurement-system-like user interactions. The use of simulations using random generated graphs as put forward in this chapter, was inspired by [51]. Chapter 4 gives a discussion of the simulator, as well as a highly condensed overview of the simulation results.

Chapter 5 offers a more in-depth analysis of the simulation results. It is proposed that performance will depend on four distinct types of properties; node properties, graph properties, interaction properties and environment properties. Results are normalised where possible to reduce the number of parameters.

The graph properties are the most difficult to express by generalised parameters. It was hoped that a single graph parameter could have been found to adequately describe the influence of the graph structure. Unfortunately, the proposed ThreadFraction parameter, while useful to explain the caching performance of some schemes, behaves rather peculiarly and produces figures that are difficult to interpret.

Chapter 6 discusses caching performance evaluation for a specific, individual graph. Probabilistic methods are presented to calculate cost and overhead for the various caching schemes, given a specific graph. The results for two example graphs are compared to the generalised performance evaluation based on the simulations. This chapter also presents a discussion of caching overhead importance relative to calculation cost, and whether caching and the use of dataflow information are profitable in this regard.

Chapter 7 presents an overview of a prototype implementation, and discusses some of the practical difficulties that arise when implementing caching in an environment with dataflow extensions, and with converting a text program to a dataflow graph.
2. Dataflow environments

This chapter discusses the basic properties of dataflow-based virtual instrumentation environments, starting with their supposed ease of programming. Considering literature of various user interaction studies, this author concludes that “visual” programming is not, as usually stated, fundamentally easier to program. Visual programming environments can be used as a design tool, in which the completed design will actually be an executable program. This is especially useful for users who have to develop their own software, while lacking software engineering expertise.

Next follow a basic description of dataflow graphs, and an introduction to the terminology and graphical conventions used in this work.

A dataflow graph can become an executable program by adding a dataflow execution scheme. Data driven and demand driven execution are presented. Both execution schemes have proponents and detractors. This author proposes that comparisons of both methods are not always fair.

Dataflow extensions exist in almost infinite variety. Their discussion in this chapter does not attempt yet another classification system, but rather endeavours to present an overview (which is by no means thorough in either depth or breadth) of some of the most common extensions and the similarities that are often hidden behind superficial differences. This overview was substantially inspired by [11, 24]. The overview demonstrates that the data dependency information is not always as readily available as one would expect. The dataflow extensions in particular tend to obscure data dependencies.

Finally, the comparison of data driven and demand driven execution is revisited, this time taking dataflow extensions into account. It is demonstrated that the choice of extensions can offset much of the perceived advantages of demand driven execution.
2.1 Virtual instruments and visual environments

The combination of hardware and software in measurement and instrumentation applications resulted in the concept of virtual instruments. The principles of dataflow programming have become associated with virtual instruments and measurement environments. There is also significant interest in visual programming (representing program structure and content graphically) as environments to develop virtual instruments. Recently developed tools help users to graphically construct software as executable block diagrams. Environments like LabView [8] IRIS [9], and HP VEE [10], among others, are well-known examples of the potential usefulness of the visual programming paradigm for measurement and instrumentation technology. The visual programming paradigm seems particularly well suited to areas that require the use of a few well defined and high-level atomic operations, so that sophisticated programs can be created by relatively simple networks of tools chosen from a predefined set [11]. Many virtual instrument applications seem to fall in this category.

The great complexity of measurement and instrumentation application development suggests that the benefits of application specific environments may be substantial in this area. Because an application specific environment restricts the user to a certain set of problem solving clichés, the tool can incorporate specialised strategies that implement those clichés efficiently.

Visual programming started with attempts to produce executable flowcharts, partly influenced by the belief that flowcharts would be useful for teaching novice programmers. Executable flowcharts have since declined in popularity. In contrast, the use of dataflow diagrams as a language has been particularly well received [12]. Dataflow-based visual environments allow to explicitly depict dependency relations and can give immediate visual feedback of the execution state of computations. Data dependencies are also critical to solid understanding of program behaviour. By scanning a data flow diagram, users can understand how each quantity is derived from the others.

A large number of different dataflow based visual programming languages have been invented. Proponents claim that visual programs are inherently better and easier to understand than textual ones [1, 13, 14]. However, experiments suggest that often the program structure is paradoxically harder to scan (to experienced programmers) in a visual programming language than in the text version. Visual programs are not easy to read, and this problem is worsened by the fact that almost each implementation essentially defines a new programming language.

Thomas Green [15, 16] suggests a list of more or less independent structural features (cognitive dimensions) of programming languages or environments. Applying this to visual programming leads to the following observations:

1. Visual programming offers a closer mapping of the program to the problem task, and introduces less “unnatural” primitives than text based programming languages.

2. Visual programs also do a better job of revealing local dependency relations.

3. However, visual environments have high “viscosity”, or resistance to local changes. This is often disastrously combined with an imposed guess-ahead on the layout level. Visual programming is easy only if one can plan the program exactly beforehand.
Otherwise the program quickly turns into a visual spaghetti. Tools to support development like global editing are generally lacking in visual environments [15].

4. Also, organising the program into an elegant and meaningful graph is far more difficult than organising a text program into paragraphs. On a global level, data dependency is seldom obvious in visual programming, due to screen size limits. Existing environments often lack tools for global dataflow analysis [15].

5. In text-based programs, layout, paragraphing and comments offer a ‘secondary notation’ that is nearly impossible to achieve in a visual program. The need to save screen space prompts the programmer to ‘hide’ parts of the graph through a hierarchy of windows. The number of windows involved can make an essentially simple program effectively unreadable.

Green’s cognitive dimensions [15] may be an unfair method to judge visual programming. Converting an algorithm specification into a program (the test on which Green’s comparison is based) is notably easier in traditional text-based programming. However, this comparison ignores the specification phase of a project completely. The visual program may be harder to edit for an expert programmer, yet be easier for the end user to assist in establishing the program’s specifications. By visualising the program structure, and the execution, visual programming environments can involve the end user in the development of a system in a way that is impossible in traditional programming. The visual program merges the design, implementation and testing phases of a project into one, thereby reducing overall development time [17]. Visual programming also encourages the encapsulation of frequently used tasks as language primitives. This allows the end user to edit and maintain the system without expert programming skill (standardised array operators, for example, drastically reduce the potential for programming errors related to array indexing). Encapsulation into a visual element could also help provide an understandable interface to libraries and toolboxes (something that is often lacking).

With an execution mechanism based on dataflow principles, a dataflow graph becomes an executable program representation. However, ‘pure’ dataflow is too restricted to be usable for advanced applications. Efforts to apply such systems to problems of realistic size and complexity illustrate a major deficiency: dataflow diagrams represent data dependencies between operations in a basic block of code, but were never meant to describe program control. Programming real applications is impossible without some means of procedural abstraction and sequential, conditional and iterative control. The challenge from the perspective of developing tools based upon the dataflow paradigm is to preserve the virtues of such an approach while trying to minimise the inherent limitations embodied by the use of a naive dataflow execution model for the visual programs.


2.2 Dataflow graphs

2.2.1 Basic concepts

This study will not consistently conform to topological graph theory terminology [18], as some of these definitions appear somewhat artificial when used for directed graphs.

Data dependencies in an algorithm are commonly represented by a directed acyclic graph. The graph consists of nodes and directed arcs. The number of nodes is called the graph size. Each node (module, box, vertex, ...) represents an operator or function (a task) and each directed arc (link, wire, edge, ...) represents a path (dependency) over which data is exchanged between nodes. The arcs are oriented in the direction of the data exchange. Incoming arcs on a node are called the node’s inputs or arguments, departing arcs are called outputs or results. The graph part connected to the inputs of a node is called upstream, the graph part connected to the outputs is called downstream of the node.

Two nodes are adjacent if they are incident to a common arc. The neighbours of a node \( n \) are the set of nodes that are adjacent to \( n \). The upstream neighbours are called parents, the downstream neighbours are called children.

A path is an alternating sequence of nodes and arcs, with each arc being directed from the node immediately preceding to the one immediately succeeding it in the sequence. A cycle is a path with at least one edge and with no repeated nodes except that the initial node is the terminal node. A graph with no cycles is called acyclic.

A “pure” dataflow graph gives a picture of how the value computed by each statement in a basic block is used by subsequent statements in the block. It is important not to confuse dataflow graphs with control flow graphs. Each node of a control flow graph stands for a basic block that can be represented by a directed, acyclic data flow graph. Figure 3 shows a control flow graph with two basic code blocks, block B2 is repeatedly executed. Figure 4 shows the dataflow graph for block B2.

![Figure 3 A control flow graph with 2 blocks.](image)

This example illustrates the important differences between control flow graphs and dataflow graphs. Just as control flow offers no information on data dependency, so dataflow cannot represent the iteration of block B2.
Input and output terminals can be drawn to represent the connections to the external world. In Figure 4, the variables on the bottom row can be considered as input terminals, and nodes whose value is accessible after execution of the graph can be considered as output terminals (represented by horizontal lines in Figure 4). The output terminals are not actually nodes. While the input terminals represent a separate graph element, the output terminals only denote that the node to which they attach has an associated identifier (and hence is accessible). Unless stated otherwise, a reference to a node or nodes in the remainder of this text will generally not exclude input terminals.

Figure 5 presents the conventions that will be used to represent dataflow graphs (except in examples using specific visual languages, where the conventions of those languages will be used). The direction in which data flows is denoted by the little triangles or arrowheads (the graph orientation can thus be top-down, bottom-up, or sideways without confusion). A node is represented by a box. The node’s arguments are represented by triangles pointing into the box. Each argument can receive at most one arc. The arguments are arranged in order from left to right (or top to bottom). The node’s results are represented by triangles pointing out of the box. A node can produce more than one result. A result can also be used by several other nodes, hence several arcs can be attached to the same result (this notation was developed to differentiate between this case and a node with more than one result). An input terminal is represented by only the result side of a node.
An output terminal is a simple straight line. These conventions are loosely based on those of the Prograph environment [14].

### 2.2.2 Constructing dataflow graphs from basic code blocks

This section presents a method for constructing the dataflow graph corresponding to a block of code, which is adapted from [3].

Each statement of the basic block is processed in turn. For a statement like \( A = B + C \) the elements which represent the current values of \( B \) and \( C \) are identified. These could be input terminals, or they could be interior nodes of the graph if \( B \) and/or \( C \) had been defined by previous statements in the block.

Then the graph is scanned for the presence of a node with \( B + C \) as its execution method. If a node denoting \( B + C \) already exists in the graph, the existing node is given the additional label \( A \). If \( B + C \) is not found in the graph, a new node with operation \(+\) is created, with two input arcs to the node’s parents. The left parent will be the node for \( B \), the right one the node for \( C \). These are the nodes found (or input terminals created) during the previous step. The new node is then labelled \( A \).

If \( A \) had previously labelled some other node or input, that label is removed, since the current value of \( A \) is the node just created.

For an assignment such as \( A := B \), label \( A \) is appended to the node currently associated with the identifier \( B \).

There are certain limitations to the translation of a basic block to a dataflow. For an assignment using a pointer reference such as \( ^*P := W \) (\(^*P \) meaning the memory location pointer \( P \) references), it is not always known to what \( P \) might point. Also, a procedure call in a basic block may change any variable as a side effect.

Operations on array elements can not be considered independently, for example, \( z := a[j] \) must follow \( a[j] := y \). The simplest solution to this is to view every operation on an array element as an operation on the entire array. Dependencies of individual array elements can be determined only if the array indices are known (and constant) at the moment the dataflow graph is created [19].

While these are minor points when the dataflow graph is constructed as intermediate representation for a compiler, they are important to the transformation of text code to an executable dataflow program.

Contrary to a control flow graph, which imposes a (very often arbitrary) total ordering on its operations, dataflow specifies only a partial order on the operations in the computation, based on data dependencies. No execution order is implied for operations not depending on each other’s data. As a result, operators in dataflow are not allowed to have side-effects that would cause different results when the operators are executed in a different sequence. Because program control is not represented in a dataflow graph, it is not generally possible to automatically convert a program written in a third generation language to an equivalent dataflow program, although some results were obtained with variations on dataflow [20].
2.2.3 Algebraic graph optimisation

The dataflow graph construction method described in 2.2.2 automatically avoids duplication of common sub-expressions, and creates the smallest graph possible without altering the algorithm [21, 22]. If an expression is encountered which already exists in the graph, no duplicate nodes are created for it, but the existing representation of the expression is identified and used.

While checking whether a node with specified parents and a specified operator exists, the laws of commutativity, associativity and distributivity could be applied to find and share larger equivalent graph sections. Because this leaves more than one possible graph representing the program, some heuristic scheme will be required to choose the most efficient ordering. For example, the statement \( d := c \times (a + b) \) could be merged with \( e := b + a \) or with \( f := a \times c \), but not with both. Which form is more efficient can not be determined at the time the node is added to the graph, since this also depends on other, later added nodes (and on the execution frequency of each node and hence on the user interaction sequence as well in the case of repeated partial execution of the dataflow program). Thus, finding the absolute “best” graph is unlikely.

A possible disadvantage to the application of algebraic transformations is that it can change the values returned by a program, by introducing overflows and underflows where none existed before. It should therefore not be applied indiscriminately.

Detecting program components with equivalent behaviour is possible in only a very limited way [23]. Some work has been done to develop algorithms for comparing versions of the same program, but the problem of deciding whether two program components have the same execution behaviour is in general undecidable. Avoiding duplication of program parts that perform the same function can be somewhat helped by supporting the creation of reusable procedures. In a visual environment this means supporting procedural abstraction and encouraging the user to develop useful new primitive functions.

2.2.4 Summary

Dataflow graphs represent the data dependencies in a basic program block. They are not meant to represent control flow, and therefore lack representations for procedural abstraction, iteration and conditional execution.

A method was presented to convert a basic block into a dataflow graph. This conversion automatically detects common sub-expressions. When the purpose of this conversion is to obtain an executable dataflow program, any side effects must be made explicit. Operations on array elements should be regarded as operations on the entire array.

Additions to dataflow must be introduced to allow for iteration and conditional execution. This will be discussed in the next sections.
2.3 Dataflow execution schemes

A dataflow representation of a program is a data structure expressing data dependency between tasks, where a task stands for a data transformation. By introducing a dataflow execution mechanism, this representation becomes a program in its own right, with a parallel, local model of execution.

This section introduces programming environments based on dataflow, their properties and limitations, and how the dataflow paradigm can be expanded to allow the expression of realistic programs [11, 24].

A computational environment consists of primitive units of computation, but it must also contain an execution mechanism that selects units for execution. The general principle of the dataflow computational model is that program execution is driven by the flow of data instead of by a linear instruction sequence. There are two main modes of execution for a dataflow program: data driven and demand driven execution.

Some environments use an execution method that cannot be clearly classified as either data driven or demand driven [1, 9]. Various dataflow environments concentrate on DSP applications. In these environments, the arcs carry the stream of data to be processed. The arcs can have different data rates, and cycles in the flow graph allow feedback loops. Usually, settings on the nodes themselves can be adjusted in a way unrelated to the dataflow structure, while the stream of data is being processed (an interesting hybrid system is proposed in [25]). Though these systems can be referred to as dataflow, they are different from the dataflow execution schemes for measurement applications considered here.

2.3.1 Data driven Dataflow

As a dataflow program is represented by a graph instead of a sequence of instructions, no single thread of control moves from instruction to instruction retrieving data, operating on it, and producing new data. Rather, data flows to the instructions, causing evaluation to occur as soon as all necessary operands are available. [26]

For the purpose of discussion, the execution of a dataflow program can be visualised as packets of data carried along the arcs of the dataflow graph by data tokens (sometimes called activation tokens, and represented by discs in Figure 6). A node will, upon completion of its method, place its results as data tokens on its output arcs. Tokens are removed from the arcs as soon as they are accessed as input by other nodes. A node is ready to execute once data tokens are available on each of its input arcs. Computations are triggered not by a demand for values or data, but by their availability. Concurrent execution is a natural result of the fact that many tokens can be present on the dataflow graph at any time; the only constraint on evaluation order is the presence of tokens on arcs in the graph.

Data driven execution is the most frequently used execution scheme in existing dataflow-based environments. In data driven execution, a node executes as soon as data from upstream nodes becomes available on each of its input arcs (upstream nodes are found by following input arcs from the node backwards, downstream nodes are found by following output arcs forward out of the node). With data driven execution, all data flows
downstream, until each data token reaches a node without children (presumably an output terminal).

![Figure 6 Data driven execution: nodes execute as data becomes available on each input arc](image)

A property of data driven execution is that, in general, all nodes in the graph will be executed. This can become a disadvantage when the graph has multiple output terminals, of which only one at a time produces a required result: all nodes are executed even though the final output of the program does not always make use of their results. This property is often quoted as an argument supporting the supposed superiority of demand driven execution over data driven execution. However, the result of the comparison depends heavily on the graph structure under consideration (as will be illustrated later), and this apparent weakness of data driven execution can be avoided entirely by, for example, making the graph parts leading to different outputs depend on a condition.

A node can only execute when data arrives on all its inputs. Data driven execution supposes that input values are presented to all input terminals for each execution of the program. This is because dataflow is almost always conservative, which means that a data token is used exactly once. A node cannot use the same token a second time. On the other hand, unused tokens are not discarded. (Non-conservative execution is used in some spreadsheet update methods, which repeatedly execute the graph until all values remain constant.)

Suppose an input terminal produces a data token when a new value is assigned to it. In the data driven system, a change at one input terminal will thus only affect an output if all other input terminals produce data tokens as well (whether their values have changed or not since a previous execution being irrelevant). In Figure 7, no output is produced because one input terminal failed to produce a data token.

Constant values used as arguments to operations can be viewed as input terminals with fixed values, or could be considered as some special kind of node. The constant node must also produce a data token each time the graph has to be executed. Alternatively, a node could be made aware of a constant argument, remember the value of the constant and not wait for the arrival of data tokens on the constant input.

![Figure 7 Output values are only produced when all inputs produce tokens in equal numbers](image)
For multiple executions of the program, the input terminals are supposed to have the
same input rate. If input terminals produce different numbers of tokens, ‘old’ data tokens
will remain on the arcs after execution of the graph (because some parts of the graph will
not execute). In such a case, it would be unclear whether the obtained result is consistent
with the latest settings (Figure 8).

![Figure 8 If ‘old’ tokens remain on the arcs, results won’t reflect the latest inputs](image)

### 2.3.2 Demand driven Dataflow

With a demand driven execution mechanism, a node will produce an output only when
downstream nodes request data from a node’s output arcs. If the node requires argument
values to execute, it will send data requests to its parent nodes. The node will wait for the
parents to send back the requested data. After execution, it will send the results over its
output arcs to the downstream nodes that made the request [27].

In demand driven dataflow, there is thus a two-way traffic in the communication lines.
In one direction data flow in the usual way from input to output. In the other direction
requests for data are sent, from the output terminals upstream to the inputs.

Demand driven execution is implemented in a number of different ways. The ‘natural’
implementation supposes that the nodes transmit demands to one another and send data in
return. However, due to its greater complexity, this method has not been used to
implement the dataflow execution of any existing environment known to the author. The
propagation of demands suggested in the theoretical model is very awkward on extensions
of pure dataflow like iteration constructions or cycles (as will be demonstrated later).
When these extensions involve multiple data rates, they would require multiple demand
rates as well.

![Figure 9 Propagation of a demand](image)

A practical implementation of “demand driven” execution (as used in [4] for example)
is more likely to traverse 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 data driven execution scheme. Alternatively,
nodes whose execution is required to obtain the demanded result are maintained in a list
which is then searched periodically for nodes that have valid data on all input arcs and can
be executed. The difference between this approach and “real” demand driven execution is only apparent when dataflow extensions are used.

![Data flows downstream in response to the demand.](image)

Most of the problems associated with demand propagation can be avoided by keeping the demand propagation phase separate from the execution of the demanded graph part.

Additional complexities arise when the graph is executed repeatedly with different requested outputs. There is probably some overlap in the calculations leading to the different outputs. When a node executes, it will pass its data only to those downstream nodes that demanded it (Figure 10). In this case, overlapping parts of the graph will be executed more often than necessary. Whereas the data driven execution scheme may calculate unwanted results (though only if the dataflow graph has multiple outputs), the demand driven scheme will result in redundant execution of overlapping graph parts of different outputs (though only if subsequent demands arrive at a node after it has already passed on its result).

This weakness of the demand driven scheme is a strong argument to cache the last result of a node for the purpose of responding to later demands by downstream nodes, rather than recalculating each time. However, this solution requires the node to determine whether the previously computed value is still consistent with the current values at the input terminals. Additionally the execution scheme is no longer conservative: a data token might be consumed more than once, or be discarded before use by all its children, when it is no longer valid (due to a changed input value). Demand driven execution separates the act of providing input values to a node (or more general, a graph) from the decision to execute the node (or graph). It becomes entirely possible that various subsequent input values are never actually used in a calculation, because no demand occurred before the input value changes again.

An advantage of demand driven execution is that only nodes that contribute to the requested output of the program are computed. This may save significant time during execution, depending on the structure of the graph. However, the properties of the execution scheme are likely to influence the programmer, who will try to avoid constructions that lead to inefficient execution in the programming environment used. It is difficult to compare the efficiency of execution schemes, when this efficiency depends in part on the programming style. For example, in demand driven execution there is no penalty for a graph with multiple output nodes, since only the requested output is calculated. It is unlikely that a programmer will use the same graph structure in a data driven environment, particularly when dataflow extensions are available. A condition to select the appropriate output is a natural choice in a data driven environment, even though it may look meaningless in a demand driven one.

In data driven execution, all input terminals must be prompted to produce a data token (by the user setting their value or otherwise) at the time an execution of the graph is
desired. A small advantage of demand driven execution is that all the input terminals relevant to the requested output will receive a demand for data, which cancels the need for any prompting to produce a data token.

It could be argued that "demand driven" dataflow is not dataflow at all. To many people, the defining property of dataflow is that operations are performed as soon as the operands are available. Others view data driven execution as a special case of demand driven execution [28], where tokens are always automatically demanded. From this perspective, demand driven execution is more general than data driven execution.
2.4 Dataflow Extensions

2.4.1 Overview

As a one to one translation from control flow language algorithms into dataflow is not possible in general [20], visual programming environments must provide means to overcome these limitations to be able to use dataflow as a computational model. Because of the absence of control flow information in the dataflow model, some program constructions can not be represented by standard dataflow graphs.

In a dataflow environment, all task dependencies, which are usually implicit in other representations, must be explicitly represented as data flow to ensure that data changing operations occur in the correct order. Computation supposedly does not produce side effects. Generally dataflow is asynchronous, there is no assumption on the speed of the tasks or data transport inherent in the representation. Pure dataflow graphs do not include the notion of variables, since there are no named memory locations holding values.

Some dataflow environments simply require the developer to do without program control [29]. For the environment designer this is of course the easiest solution. Sequential execution, procedural abstraction and iteration are nevertheless deemed necessary to tackle complex problems. A dataflow-based environment requires some extensions to implement these constructions.

Some important constructions not supported by pure dataflow are:

- Functions/procedures and recursion
- Sequential execution
- Iteration
- Conditional execution

These can be implemented in many different ways [11, 24]. Most implementations can be recognised as belonging to one of two distinct approaches to implement control. One 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 a controlling node. An example of this would be a subgraph contained in a “while-structure” node, as used in LabView [8] for example. The control node will operate on the subgraph in ways that can not be expressed in the pure dataflow formalism, but will appear to behave like any other node to the rest of the graph.

The second approach uses the dataflow execution mechanism to provide control (see [20, 28] for examples). Conditional execution can be implemented by introducing selector (or merge) and distributor (or switch) nodes. 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. Combined with cycles in the dataflow graph these special nodes allow iteration.

The second approach offers the virtues of a uniform data model, whereas in the first, different execution mechanisms are mixed. 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 less prone to error, especially in combination with other high-level primitives, and most environments prefer this approach. Ptolemy [30] even deliberately mixes heterogeneous models of computation, to unify a diversity of design styles. In Ptolemy, a node of a dataflow graph can be specified by a different model, for example a finite state machine, whose states can in turn be dataflow graphs, and so on (practical implementation of this is not without difficulty though).

The following sections will present various extensions dealing with the limitations of pure dataflow for both of these approaches.

2.4.2 Procedures

The nodes in a dataflow graph can be language primitives or they can contain subprograms specified in another language. Dataflow then serves as a co-ordination language for subprograms written in a host language (using hierarchy to mix different models of computation). A pure dataflow subprogram is not allowed to have side effects, all data changes must be made explicit through data dependency links because the order in which nodes are executed is only partially fixed.

Dataflow graphs do not include the concept of a function that returns a value. One could surround part of a dataflow graph with a boundary and call it a function or procedure, where all inbound arcs to the region would be the parameters and all outgoing arcs would be the results. To obtain a form of procedural abstraction, a subgraph may even be condensed into a single node. A graph can thus be encapsulated into an atomic primitive. Such an organisation leads to hierarchically nested dataflow graphs.

Suppose a subgraph “procedure” is represented by a single node and used as a primitive operator. The same operator might be invoked simultaneously in different parts of the graph. The execution scheme should be capable of handling these multiple simultaneous invocations of the procedure’s subgraph [26, 31]. This can be approached from various angles:

Research into dataflow computer architectures makes the distinction between static and dynamic dataflow (several execution models for dataflow programs have emerged from research into dataflow computer architectures). Dataflow architectures are classified as either static or dynamic [13,32, 33].

In static architectures, the program graph is loaded into memory in its final form before the program begins execution. Static architectures often use the same storage space for instructions and data. For each node, a static dataflow architecture provides space for an operation code, a set of operand slots and a destination pointer. The node is ready to execute when all operand slots are full and all destination slots are empty. A static environment can copy the subgraph for each invocation of a procedure, as a compiler might expand procedures ‘inline’. This is only a limited solution, as it is for a compiler. It does not support re-entrant dataflow code (that is, code that can be used simultaneously in multiple places in the dataflow graph) nor recursion. For recursive functions in particular, the recursion depth and number of invocations is generally unknown until runtime.
On a dynamic dataflow architecture, program execution may be modelled as a process that extends and re-evaluates a dataflow graph step by step. It executes the following two steps as long as the constructed data flow graph changes:

1. evaluate any built-in system function with available input data
2. expand any call of a user defined function with available data input by copying its subgraph, and connecting its input and output arcs to the corresponding nodes in place of the function call.

Nodes are created at runtime in dynamic architectures. Such architectures support simultaneous multiple instances of a procedure, recursion, and loop unravelling.

Separating instruction and data space, a “context” can be created for each invocation, where a context is a separate data space for the data tokens of a single, specific invocation of the procedure. Several instances of a node can execute simultaneously in different contexts.

A related approach is to add a different tag to the data tokens of different invocations. In that case, one “instance” of the subgraph must handle the differently tagged tokens simultaneously. The subgraph can have multiple, differently tagged tokens on each arc, and a node executes when tokens with identical tags are available for each input arc. Tokens are labelled to distinguish values arising in different contexts or from different incarnations of an activity. The tokens are labelled to insure that values produced during the computation can be ascribed to the appropriate input value. Iteration and function calls can be accomplished by using nodes that manipulate the token labels.

Ptolemy [34] implements recursion by the dynamic runtime creation of as many versions of the recursive subgraph as required by the recursion depth. Higher order functions too (functions taking functions as arguments) can be implemented by runtime dynamic creation of dataflow graphs. With this approach, iteration can be handled as a special case of recursion.

Because dynamic dataflow programs are harder to represent graphically, many visual environments are limited to static dataflow and do not support recursion.

2.4.3 Sequential Execution

Dataflow diagrams do not identify any particular sequence of execution beyond the order specified by data dependency. This assumes that none of the operations produce side effects, which is not very sensible. A measurement system will not exclusively consist of data transforming nodes, (e.g. matrix manipulation) but will also include some instrument control. External coupling between such functions (not defined as part of the flow graph) poses a problem, because pure dataflow does not allow a user to specify that some actions should be performed in a sequence.

An example of such a sequence would be a data acquisition situation, where an instrument set-up must be performed before acquisition can start, although the set-up generates no data that is used by the acquisition process.

To support sequential execution, nodes can be expanded with extra inputs that do not have any function other than introducing an additional dependency for sequential control. These extra inputs accept any incoming value as a trigger. Sequential execution of operations can then be forced by connecting them in the desired sequence with a
dependency link to the extra inputs. Such a synchronising link will not carry meaningful
data, since its only function is to sequence operations, or to express external data coupling
between operations. The node with the extra input will delay its execution until all trigger
signals have arrived.

The use of virtual data dependency links to express task dependency does not deviate
much from the conceptual model of dataflow programming.

Since there is no assumption on the speed of the tasks or of the data transport, there is
little real-time control in a dataflow program. A delay could be implemented by a delay
node, but dataflow as such offers no way to guarantee execution of a particular node at a
specific time, because the method to select nodes ready for execution is not specified in the
dataflow paradigm, and an unspecified delay may occur between the moment the node is
ready and its actual execution.

![Figure 11 HP VEE module with its pins](image)

The HP VEE [10] visual environment uses a distinct type of links for data dependency
and control dependency (Figure 11). When one type is used for a node, the other becomes
optional. This emphasises how virtual data dependency links can be viewed as mixing the
control flow and data flow paradigms. HP VEE makes use of a virtual input terminal
called ‘start’ to generate tokens on the control arcs. The tasks in a sequence may lack any
‘real’ data dependency, which makes the association to ‘pure’ dataflow look feeble at best.
Still, the separation is not complete, as data outputs will often be connected to control
inputs.

### 2.4.4 Conditional Execution

Conditional execution means that one of two subgraphs will be executed, depending
on a condition. The conditional subgraphs can be separated from the main graph, and
incorporated in a single abstraction, like the LabView case structure in Figure 12 (the
border represents the case structure, the “question mark” input is the controlling argument,
the left triangle is the comparison operation). Control flow is then applied to execute one
of the subgraphs, depending on the condition.

![Figure 12 CASE structure (LabView)](image)

if \((A < B)\) then \(C = A - B\) else \(C = A + B\)
Another approach is to use dataflow conditionals to control dataflow from and to the subgraphs. In this approach a distributor (or switch) node and a selector (or merge) node are used to introduce conditional data paths (Figure 13).

![Figure 13 The same condition with merge and switch nodes](image)

Switch and merge nodes do not inevitably appear in pairs. There are three possible configurations. In the case of a ‘conditional assignment’, a merge node will select one of the two incoming data tokens and discard the other, based on the condition. (In demand driven execution, the merge node should first evaluate the condition, then demand data from the appropriate node.) In data driven execution this configuration is pointless.

A single switch node will pass on an incoming data token to one of its two outputs, based on a condition. When the two datapaths do not reunite downstream of the switch, this construction selects which of two outputs it has to calculate (Figure 14). A single switch with only one output corresponds to an if-then clause. Such a switch can halt graph execution.

![Figure 14 Single Switch example](image)

The single switch configuration is pointless for demand driven execution, and so is the single merge configuration for data driven execution. This illustrates how the execution scheme will influence the graph structure (as discussed in 2.4.7).

![Figure 15 A Merge with multiple inputs](image)

A switch and merge can be generalised to have multiple outputs or inputs. In that case the condition value will be used to select the appropriate input or output. This corresponds to a CASE statement in imperative programming.

Switch and merge nodes, even though appearing more similar to dataflow, depart fundamentally from the principle of dataflow execution: such nodes change the rate of data tokens on the different data paths. In a correct implementation of conditional execution, a switch will separate the data paths, and a merge will recombine them. However, the
syntactic correctness of the graph may become obscure when multiple switches and merges are intertwined. The different data rates in the graph caused by switch and merge nodes can lead to program inconsistencies. The use of a higher-level conditional abstraction, like the presented CASE-structure, reduces the programmer’s capacity to create such inconsistent programs.

2.4.5 Iteration

First of all we have to distinguish two kinds of iteration, usually called parallel iteration and sequential iteration [33]. Parallel iteration means the simultaneous execution of a function for all elements of a given list or set of input elements. All computed results of parallel iteration are usually returned as another list.

The concept of parallel iteration conforms well to the dataflow paradigm and can be added without much trouble to any dataflow environment. This is no longer true for sequential iteration. This kind of iteration evaluates the same piece of code repeatedly with its own output values as new input values, until a certain condition holds true. Most dataflow environments support some form of sequential iteration.

A construction that could be viewed as related to parallel iteration, splits an array or list into its components, let some nodes operate on these components in sequence, and then collect the results back into an array. Of similar use are count nodes, which produce a number of data tokens in sequence, corresponding to input specification. These are often used in conjunction with collect nodes (Figure 16). This construction changes the rate of data tokens, the subgraph with the different data rate should be isolated from the rest of the graph, with connections only to the count node and the collect node.

![Figure 16 Count-Collect construction](image)

The various implementation methods for sequential iteration will be demonstrated using the simple iteration in Figure 17.

```plaintext
sum = 0;
i = 0;
doi++;
   sum = sum+i;
}
while (i<=n && i!=x)
```

![Figure 17 Iteration example](image)

2.4.5.1 Iteration abstractions

Some languages have sequential iteration abstractions [8] that control repeated executions of a contained subgraph (as in Figure 18). The enclosing node must repeatedly assign values to the input terminals of the subgraph, evaluate a loop termination condition, and decide whether to execute the subgraph again or to return the last values. In these iteration abstractions, cycles in the graph are generally broken by implicitly introducing
special loop variables, which possess a pair of values (old, new) instead of a single value. In Figure 18 the old and new values of a loop variable are represented by boxes on the subgraph border marked with up or down pointing arrowheads (the box marked i is an iteration counter, the box with the curved arrow is the loop terminator).

![Figure 18 WHILE structure](image)

2.4.5.2 Implicit cycles through explicit loop variables

Explicit variables are an extension to dataflow that uses ‘get’ and ‘set’, or ‘read’ and ‘write’ nodes to access a specific memory location. They are used in several environments [4] to evade the limitations of dataflow. The example in Figure 19 uses the Data Explorer syntax. The two input nodes enter the value for x and n. The “foreachN” node counts to a value specified by its argument, n in this case, and fills the role of loop counter i. The “sum” node adds i to the sum stored in the loop variable. When i is equal to x, the “done” node acts as a loop terminator, and the resulting sum is returned as output.

Implicit control structures based on variables are not universally considered to belong in visual programming, as these control structures are in fact invisible. However, except for the missing enclosure around the subgraph and the explicit accessing of loop variables, this construction is very similar to the one in Figure 18. Note that the local variables must be initialised (not visible in Figure 19).

![Figure 19 Iteration through variables](image)

2.4.5.3 Cyclic graphs

The dataflow execution paradigm can be applied to generate iteration by introducing cycles in the graph. These cycles are implicit when using ‘variables’, but can be made explicit, using switch and merge nodes to control the iteration [24,35,36].
When using explicit cycles, it may be necessary to initialise certain arcs with a specific token to start the first iteration, as is the case for the condition arcs to the merge nodes in Figure 20. The condition arcs determine whether the “True” or “False” input (or output) is selected.

Handling of input values is somewhat complicated, as some inputs must repeatedly provide a data token with the same value, and should only be cleared when the iteration is completed, while other input nodes change with every iteration step, and yet others must provide only one token during the iteration. At the end of the iteration, it could be necessary to remove some remaining tokens (to allow repeated executions).

It may seem at first glance that the use of cycles offers more flexibility to schedule operations, but the evaluation of the condition will still implicitly synchronise the independent tasks within an iteration.

The cyclic dataflow extensions create iteration structures using the equivalent of a GOTO instruction, a giant step backwards in programming methodology. Loops and conditionals should be built-in as fundamental components rather than be constructed of more primitive pieces.

2.4.5.4 Difficulties with iteration

Except for parallel iteration, all dataflow iteration constructions have a different data rate in the iteration subgraph than in the rest of the graph. As a result, the subgraph must be isolated from the rest of the graph, whether or not the subgraph is actually surrounded by some visible boundary. Iteration constructions using switch and merge nodes may appear to stay closer to the dataflow principle, but even there some controlling agent will be needed to initialise arcs with data tokens, and to regulate the number of tokens to be produced by each argument node of the subgraph: some inputs of the subgraph must repeatedly provide a data token with the same value, and should only be cleared when the iteration is completed, other input nodes change with every iteration step, and yet others must provide only one token during the iteration.

The operation details become rather confusing in the case of a demand driven execution scheme that actually propagates demands through the graph during execution. Even if the implementation can prevent the demands from propagating forever in the loop, the problem of keeping the demand up until the iteration terminates remains. A count node for example will not work demand driven, unless somehow the corresponding collect node will produce exactly the required amount of demands, as in a true demand driven scheme.
available data tokens are simply ignored if no demand is present. The same goes for any multiple data rate structure. This may explain why the true demand driven execution scheme is apparently never used in extended dataflow environments.

The iteration constructions in dataflow environments generally behave differently from their sequential equivalents, and differ from one environment to another. The execution model of these constructions can not always be derived easily from their visual expression. A programmer must be aware of all aspects of the dataflow execution model, and the semantics of the added constructions have to be well understood.

2.4.5.5 A different approach: dynamic loop unfolding

Dynamic loop unfolding means that the dataflow graph (or at least its data space) for the iteration is created at runtime by adding as many copies of the subgraph as needed [34]. This approach avoids the construction of cycles and reflects the opinion that iteration is just a special form of recursion. While this approach maintains the ‘pure’ dataflow form of the graph itself, it requires a control mechanism to keep track of the termination condition, and to handle the creation and destruction of the multiple subgraph copies and their connection to the rest of the graph. For each execution the iterated graph part has to be recreated. An advantage is that all values are known during the graph creation process, so common subexpressions can be eliminated more effectively by taking the actual values into account.

2.4.6 A few words on parallelism

Its ability to express parallel algorithms has always been a special incentive to use the dataflow model of computation. In the dataflow model instructions are scheduled at runtime based on the availability of data. Scheduling and synchronisation of concurrent threads is provided implicitly by the model. In an environment that supports parallelism, the elimination of explicit, machine dependent, parallel constructions can clearly improve the productivity of the programmer by eliminating most of the bugs and assorted synchronisation errors that tend to plague imperative parallel implementations [37].

One can distinguish two major kinds of parallelism in applications:

1. Data parallelism, consisting of parallel execution of similar operations on a large number of data elements.
2. Functional parallelism, where the execution of separate functions is overlapped. This is the kind of parallelism dataflow execution lends itself to most.

In the dynamic or tagged token dataflow model, each data token has a context tag distinguishing it from others executing the same node, thus allowing for maximum parallelism in recursive functions and loops. The dataflow execution model also allows easy switching among a set of ready instructions. This makes the dataflow model a natural choice for the implementation of distributed or parallel implementations [38].

However, it must be noted that the maximum parallelism as offered by the dataflow model is not necessarily the most efficient level of parallelism. Fine grain parallelism will make no use of instruction level locality, and may result in large peak demands for resources. Various methods for selecting an appropriate grain size are suggested in [39, 40]. Other considerations could dictate the distribution of operations, rather than their dataflow relations: load balancing in a heterogeneous distributed environment means
spreading calculations evenly while minimising communications, taking the power of each processing element into account.

### 2.4.7 The effects of dataflow extensions on the execution scheme

Without dataflow extensions, data driven and demand driven execution will only differ for graphs with multiple output terminals. In Figure 21 for example, data driven execution will execute all nodes once input values are available.

![Figure 21 Example graph](image)

Suppose the user is only interested in output 2. Then the execution of nodes A and B is redundant. In demand driven execution, only the nodes leading to the requested output are executed. The demand driven scheme seems superior to the data driven scheme.

However, suppose the user later decides to view result 1 also. With data driven execution, this result is already available. In demand driven execution, all nodes leading to the requested output are calculated. Node C will be calculated again! Unless the demand driven execution method is supported by caching (and methods to determine whether the cached results are still consistent with the input values), the data driven scheme now seems superior.

While this last example is somewhat unfair towards demand driven execution (it should actually count as two ‘runs’ of the program, and most demand driven implementations avoid repeated execution when the different outputs are requested together in the same run) it nevertheless illustrates the too simplistic nature of some execution scheme superiority arguments. This is especially true once we leave the land of pure dataflow and enter the world of actual dataflow environments with their near infinite variety of dataflow extensions. Because what actually happens in the above example is that the demand driven scheme is used as a primitive program control method to implement conditional execution.

Figure 22 shows essentially the same algorithm as Figure 21, but in this version it is unambiguous that only one of the two ‘results’ is wanted at each time. If the demand driven execution scheme evaluates the condition of the Merge node first, and then sends a demand to the appropriate branch, the modified version changes nothing for demand driven execution. In Figure 22 the condition still depends on the user (through the condition input terminal), but the main advantage of this structure is that the condition could also be decided by the algorithm.
The modified version does not change a thing for data driven execution, since nodes A, B and C are already executed before the Merge node is evaluated. To select a graph part for execution in the data driven scheme, the condition must be placed upstream of the nodes in question, as in Figure 23.

The switches before node A and B are needed to absorb unused data tokens. This structure will restrict data driven execution to only the needed graph part. The switches have no effect on demand driven execution. This may seem like an awkward construction at first sight, but the previously discussed LabView CASE structure for one is exactly based on this principle.

The examples above show that the decision to use data driven or demand driven execution depends strongly on the dataflow extensions available. In Figure 21, no extensions were present, and demand driven execution was used as a primitive way of selecting graph parts for execution. While this may seem like a natural way to implement conditional execution (especially compared to Figure 23), it will only work for very simple problems, where the decision can be left to the user.

In contrast, Figure 23 uses dataflow extensions where the execution scheme (data driven or demand driven) has no effect on the number of executed nodes.
3. Caching of Intermediate Results in Dataflow Environments

The execution scheme of a dataflow environment can be adapted to accommodate caching. This chapter introduces caching schemes that can utilise the data dependency information available, and discusses how they can be implemented in the data driven and demand driven scheme.

The concept of threads, and how they allow a reduction of the required cache space, is introduced in this chapter (the term is loosely based on terminology used in parallel machines [39] for clusters of sequential code that are asynchronously scheduled).

The concept of time tags and input terminal sets for demand driven execution was proposed in [41], it also bears some similarity to UNIX’ Make command, and the update mechanism of some spreadsheet implementations.

This chapter introduces three different caching schemes, reducing caching overhead and memory requirements to different extent, at the expense of possible reduction in caching effectiveness. Unless stated otherwise, it is assumed that the environment uses static dataflow with high-level abstractions.

Non-dataflow based environments can also collect data dependency information to make caching more efficient (Maple for example, though the documentation [5] does not identify the adopted methods as such).

The chapter concludes with a demonstration of why caching should be handled automatically.
3.1 Caching in data driven dataflow

In data driven execution schemes, a node will execute as soon as all of its inputs have received a data token. Upon execution, the node produces one data token on each of its output arcs. The data tokens on the input arcs are removed once the node has generated an output. Data tokens flow through the graph, initiating calculations, until each data token reaches an output terminal.

Suppose each node (or input terminal) has an associated cache to store its last value. An executing node - about to place a data token on an arc - can compare it first with the cached value. If the new result is the same as the previous value in the cache, the data token is marked “unchanged” by a tag that is added to the token. A node that has only unchanged tokens 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, visualised in Figure 24, prevents needless execution of operations that do not depend on a changed input terminal. It even 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).

When the graph itself is edited, each node whose input arcs or operation is changed must be forced to recalculate on the next execution. The result of an edited node is tagged depending on whether the new result is equal to the cached value, other nodes are treated according to the caching scheme used.
Note that the use of tags limits cache comparison to newly calculated results, because tokens are only compared after each operation that was actually executed. If all input terminals have unchanged values, not a single comparison takes place downstream of the input terminals. Storing only the last result for each node eliminates the cache look-up overhead, since the cache contains just one value. Because the data tokens already contain validity tags, it is unnecessary to store previous arguments with the cached result for comparison.

### 3.1.1 Threads

If nodes sometimes produce the same output for different arguments, it is useful to compare a newly calculated result with the previous value in the cache. Otherwise (if nodes with a different input are extremely unlikely to yield the same output), no comparisons at all are needed downstream of the input terminals, as the added tags will provide sufficient information to determine which operations require execution.

In this case, not all intermediate results need to be stored either. This can be explained as follows:
The *parents* of a node \( n \) (the nodes that \( n \) uses as arguments) will be designated as \( \text{Pa}(n) \). The *children* of node \( n \) (the nodes that use \( n \) as an argument) will be designated \( \text{Ch}(n) \).

Each node \( n \) ultimately depends on a *set of input terminals* as its ancestors, which will be called \( I(n) \). If \( n \) is an input terminal, let \( I(n) = \{n\} \). The set of input terminals of a (non-input) node is the union of the input terminal sets of its parents \( I(n) = \cup (I(\text{Pa}(n))) \).

![Figure 25 Parents, children and input sets of a node n](image)

A node \( n \) will only need to be recalculated if a change occurred to one of the elements of \( I(n) \). There is no need to store the intermediate result for a node \( n \) for which \( \forall n_j \in \text{Ch}(n) \), \( I(n) = I(n_j) \). This is because the value of \( n \) will only be needed if one of its children must be recalculated. If none of the children depend on other input terminals, their recalculation implies that the value of \( n \) must be recalculated as well (still assuming nodes with a different input will not yield the same output).

In Figure 25 for example, node \( n \) will have to be recalculated if a change occurred to one of the input terminals \( a, b, c \) or \( d \). Suppose input \( a \) has changed. Then node \( n_1 \), node \( n \) and node \( n_3 \) have to be recalculated, while the cached result from \( n_2 \) can be used as argument to \( n \) and \( n_3 \). There is no scenario in which node \( n_3 \) will ever use the cached value of node \( n \) to recalculate itself, since both nodes depend on the same set of inputs. If the value of \( n \) is still valid, then so is \( n_3 \), and if \( n_3 \) requires recalculation, then so does \( n \). Since node \( n \) has no children with a different set of input terminals, and is not itself an output node, caching the result of \( n \) for subsequent use is futile.

A *thread* is defined as a set of the largest number of connected nodes (including input terminals) having identical sets of input terminals. Each node (or input terminal) in the graph belongs to exactly one thread or constitutes a thread by itself (for the sake of uniformity, an input terminal is considered to depend on 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 (in data driven dataflow). 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. In Figure 25, nodes \( n \) and \( n_3 \) form (part of) one thread, \( n_1 \) and \( n_2 \) each constitute a thread all by
themselves. Each input terminal (a, b, c, d) also constitutes a separate thread in this example. The number of threads in a graph will determine the number of results that require caching.

Reducing the number of cached nodes only works effectively once the algorithm is fixed. If a node is edited, by a change to its update method or set of arguments, it will have to be recalculated on the next execution. This requires the argument values to be available, even when they depend on the same set of input terminals. An edited node should always tag its result as changed on the next execution. That way a change in the algorithm is automatically reflected in the result, while only the nodes affected by the change are recalculated. If not all results are cached, a change to the graph requires complete recalculation to guarantee that the changed nodes have arguments available.

3.1.2 Some observations

The data tokens are often stored anyway in dataflow environments [10]. As a node generates a token, it dynamically allocates the memory needed. Often, the memory is not deallocated until the next token is generated by the node. If a token contains a large array, the node allocates a lot of memory, if the next token it generates is smaller, it releases some of the memory at that time. In such an implementation of a dataflow environment, the presented caching scheme would not even increase the memory requirements of the system.

In the previous discussion, 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 imagine possible situations where some action, particularly one with side effects (I/O operation, timer start, instrument reset) is required whenever a node is visited, whether its arguments changed or not.
3.2 Caching in demand driven dataflow

For demand driven dataflow, execution of a node occurs 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. The node will wait for these argument nodes to provide data. After execution, it will send the results over its output arcs to the downstream nodes that issued the request.

In demand driven execution, a node that has a cached result available, will have to determine whether the cached result is still valid, before it can be submitted in answer to a demand.

3.2.1 Difference from data driven dataflow

In the case of multiple outputs, a node is eligible to execute only when it is on a path from a requested output terminal to a changed input terminal. A path was defined as an alternating sequence of nodes and arcs, with each arc being directed from the node immediately preceding to the one succeeding it in the sequence.

A path from to is designated as . In Figure 26 for example, only node is on a path from output 1 to the input terminals.

Suppose a new value is assigned to input terminal . The user is interested in output terminal . Consider a node . The following cases can be distinguished:

- if , there is no need to execute .
- if ,
  - if , there is no need to execute .
  - if , must be executed.

A data driven system can not discriminate between the last two cases. A demand driven execution scheme, on the other hand, will reduce the number of operations to the amount required for the output under consideration. As a result, in the demand driven scheme a node will not always execute when its input data becomes available.

Since a node in the demand driven scheme will not always execute when its input data becomes available, it is not sufficient to look at the ‘unchanged’ status of the arriving data tokens to determine whether the node still has a valid result in its cache. For example,
suppose an input terminal is assigned a value several times before a demand reaches it. The unchanged status says only whether the value changed since the last assignment (input) or recalculation (interior node), but not whether it changed since its last use. This problem can be solved by providing data with version or time information.

### 3.2.2 Time stamps

Assume a clock is incremented each time an input is changed, and every node or terminal is stamped with the time value of its last update. A recalculated node can take on the time stamp of its most recent argument. A node is up-to-date - and hence needs no recalculation - if its parent nodes are up-to-date and none are newer than the node itself.

![Diagram of demand driven caching scheme](image-url)

*Figure 27 Demand driven caching scheme*

The time stamp effectively replaces the ‘changed’ tag, except when a node with ‘new’ argument values produces an unchanged result. To avoid the execution of nodes
downstream of an unchanged node, the result must carry a “time of last change” stamp in addition to the time stamp (Figure 27).

The validity of the cached result of a node is checked by recursively determining the validity of the node’s arguments. To determine whether to execute or not, the node will compare its time stamp with the time stamps of its arguments (after the arguments first established their own validity). When a node is no longer up to date according to the time stamps of its arguments, it means one of the input terminals the node depends on have changed since the node was last calculated.

![Figure 28 Node is up to date](image)

In Figure 28, the time stamps on the arguments are not more recent than the time stamp on the cached value. The node is up to date and can forward its cached value downstream.

![Figure 29 Node is not up to date but arguments are unchanged](image)

If the time stamps on the arguments are more recent than on the node’s cache, the node will compare its “time of last change” stamp with theirs to check if its newer arguments have changed value since its last update. If the arguments have not changed, the node will update its time stamp to the time of the newest argument and send its cached value downstream (Figure 29).

![Figure 30 The node must be recalculated, the new result is different](image)

If the arguments did change since the last execution of the node, it will execute its recalculation method and compare the new result to the cached value. If the new result is
different, the node will make its “time of last change” stamp equal to its time stamp (Figure 30).

Figure 31 Recalculation produces same result as in cache

If the new result is identical to the value in the cache, the node will only update the time stamp, and its “time of last change” stamp will retain its value (Figure 31).

The dual time stamps alone can effectively avoid unneeded operations, and overhead is reduced as in the data driven case.

If a change is made to the algorithm, a node whose update method is edited (arguments or operation is changed) should always recalculate during the next execution where its result is demanded. The time stamp should be changed to the time of recalculation, so that dependent nodes will be automatically updated. The time of last change stamp is determined in the usual fashion, depending on whether the value in the cache is equal to the newly calculated value. This will only work properly when all nodes cache their results.

### 3.2.3 Input terminal sets

In the presented demand driven scheme, a validity check is propagated through the graph every time an output is requested. If the propagation overhead is important (in a distributed system with slow or intermittent connections for example) propagation of the demand through the graph can be limited to outdated graph parts.

The validity of a node is ultimately determined by a (small) set of input terminals. By providing each node with a list of the input terminals it depends on, checking the validity of the node is reduced to comparing the time tag of the node with the tags of the terminals in the list. The demand will then be propagated only to nodes that are no longer valid. If the graph is very large, or connection time is important (like in a distributed system) this could result in important savings.

However, determining the input terminal set of a node is not always straightforward. If the graph contains a conditional node (a merge node), part of the calculation becomes conditional. Hence, the dependencies of a conditional node (and all other nodes depending on this one) become dependent on the runtime value of the condition.

There are two ways to handle this. The input terminal sets could be enlarged to include all the input terminals used in all possible different outcomes of the conditions. However, the input terminal set of a particular node then includes terminals that may or may not be used, and the validity is checked too often.
A better way is to adapt the input terminal sets to the value of the conditions. It takes only a small additional effort to recompile the input terminal sets of the nodes during recalculation.
3.3 Caching and dataflow extensions

As described above, caching intermediate results can be implemented easily in the ‘pure’ dataflow execution schemes. However, not all dataflow extensions are equally compatible with caching.

3.3.1 Side effects

Execution schemes that do not always execute the entire graph require special care in the handling of virtual data dependency links and side-effects in general. The criteria to determine whether a node should execute, although appropriate in the general case, may be unable to deal correctly with nodes linked through side effects. For example, when all other arguments to a node are unchanged, should a virtual dependency argument cause the execution of a node? This can only be determined case by case. The user may have to annotate certain arcs accordingly.

3.3.2 Dynamic dataflow

In dynamic dataflow, the dataflow graph is created during execution, and destroyed afterwards. Caching the results of nodes in a dynamically created part of the graph would probably be futile. This graph part is unlikely to exist during a subsequent execution. If the dynamic graph part changed since the previous execution, even though some of the operations may receive the same values as before, the data dependency relations are different, and cannot be safely relied on to determine cache validity.

Structures in which a subgraph is repeatedly executed (iteration, recursive function…) can cache results from one iteration step to the next, by keeping only the last value for each node in the subgraph. However, this way cached values will, in general, not be transferable between executions.

Alternatively a caching scheme with a centralised cache can be adopted for only the dynamic part of the graph, but this would not be able to benefit as much from data dependency information.

3.3.3 Demand driven execution of iteration

In a demand driven iteration, time stamps cannot be determined by input terminal assignment time as in the scheme suggested in section 3.2. Unless the time stamps change between iteration steps, the nodes in the subgraph will not execute when their input consists of “old” data from a previous execution step. The iteration mechanism will have to increment the time stamps between each iteration step. For “true” demand driven execution, cycles with switches additionally require that somehow demands are generated in the subgraph at the data token rate, which is a different rate at which demands are generated in the rest of the graph. The argument nodes of the subgraph must be able to determine when a demand should be passed upstream, and when it should not leave the subgraph. A high-level iteration abstraction that handles the demands automatically will greatly reduce the programming effort in this case.
3.3.4 Input terminal sets

A property of caching is that, unless a node is connected to a virtual input terminal and, in demand driven execution, on a certain path to an output, there is no way to make sure it will execute, if none of the arguments changed. The basic assumption of caching was that a node whose arguments are unchanged does not need to execute.

To handle the occasional exception, the node can be marked to execute anyway. When a node is visited, it has a chance to execute, whatever the status of its arguments, and influence the output value, by adjusting its time stamps or changed tag. In the data driven scheme, the node will certainly be visited by the execution scheme.

In the demand driven scheme, a node requiring special treatment should be on a (perhaps virtual) path to each output for which it must be executed. The node will then be visited by the execution scheme and can be marked to execute anyway.

With input terminal sets, it is far more difficult to guarantee the execution of a node, as there are several conditions that must be met before a node is even considered by the execution scheme. Although propagating the demand only to nodes that are no longer valid looks like a very simple improvement, it actually completely changes the execution scheme. By introducing input terminal lists, nodes in the ‘valid’ part of the graph are simply ignored. Without this improvement, every node leading to the requested output is at least visited by the validity check, so nodes that have to execute although their arguments are unchanged, can simply be adapted, on the node level, to do so. This flexibility to locally handle situations that defy pure dataflow calculation is lost when input terminal sets are used, since not every node the output depends on will be visited during execution.

Figure 32 Virtual input terminal

However, such special nodes are probably rare. A possible example would be a node that has to record, save or print something each time an output is requested. A possible extension to deal with such nodes in the demand driven execution scheme with input terminal sets would be a time stamp value that is always interpreted as “more recent” (but would not necessarily be passed downstream), and a virtual input terminal that always returns this time stamp (see Figure 32). The suitability of this method will depend on the type of applications that are to be created in the environment.
3.3.5 Distributed or parallel execution

A distributed dataflow based environment poses additional motivation to utilise the available dataflow information for caching. The intermediate results are naturally stored at the same location where the operations are executed, rather than in a central cache.

Under distributed execution, factors such as propagation overhead due to communication latency, that hardly play a role otherwise, will become more important. In a distributed implementation, it seems reasonable to cache the results of a node at each of its children as well, so that when a result is unchanged, it does not have to be sent over the network along with the activation token. This saves transfer cost even if the child node has to execute anyway due to the change of other argument values.

Not all distributed computing uses the dataflow model, interactions among distributed objects are certainly not constrained to follow data dependency relations. While distributed computation in general might benefit from caching, the specific schemes presented here only apply to environments where interactions between nodes follow dataflow relations.
3.4 Summary and presentation of caching schemes

If the dataflow information of the algorithm is maintained by the environment, a cache can be associated with each node of the dataflow graph, instead of with each function used in the algorithm. This allows a reduction of the number of entries in the cache. In order to avoid recalculating results from the last execution, it is sufficient to store only the previous result, making cache lookup unnecessary. Without dataflow information, the caching scheme will not know how many different instances of a function have their results stored in the same caching table, hence the number of cache entries required to avoid recalculating results from the last execution is unknown.

The validity of the cached value can be determined from the validity of the upstream nodes, by passing cache validity information along the arcs of the dataflow graph. A node’s cache contains a valid result if all the node’s arguments are unchanged since the cached result was computed. As all nodes ultimately depend on the input settings, and these dependencies are readily available from the dataflow graph itself, the validity of every cached value in the graph after a change to the settings can be deduced from comparison at the input terminals.

Dataflow information can be exploited to reduce the number of caches in a justifiable fashion. 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. Once the program graph becomes fixed, only the ‘end’ nodes of the thread (which deliver arguments not belonging to the thread) benefit from caching. Their downstream neighbours may need the last value again to recalculate, due to a change to one of those other input terminals. 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.

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.

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.

During the development of an algorithm, changes are made not only to the input values, but also to the graph itself. A node whose update method is edited by the developer, through a change of its operation or its argument nodes, must naturally be forced to recalculate on the next execution of the edited graph. This requires that all nodes cache their intermediate results, otherwise arguments needed for the edited node must be recalculated as well. All nodes that depend on the edited node will be treated according to the caching scheme. Nodes unaffected by the change will not be executed. 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.

A change in the graph structure can only be detected if the execution scheme visits the changed node(s) during graph execution. If the demand driven scheme uses input terminal sets, the changes may go unnoticed. Therefore input terminal sets should not be used before the design of the system is fixed.

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.

Dataflow was never intended to represent program control, and all dataflow extensions to implement control deviate fundamentally from the dataflow paradigm. Caching makes this fact even more apparent. Caching schemes may not be able to extract dataflow information from graphs with dataflow extensions.

Note that several simple environments do not use control structures at all. The environment could always resort to caching without dataflow information for some parts of the graph.
3.5 Use of dataflow information in other environments

The presented schemes are most useful in the implementation of a dataflow environment, but other environments may be sufficiently similar to dataflow execution to make the caching schemes applicable. Messages in object-oriented or discrete-event systems are sometimes passed one-directionally between objects in a dataflow-like fashion. Caching can also reduce communication overhead in distributed applications ([42] for one example).

An environment is not necessarily dataflow-based in order to collect data dependency information that can be used by a caching mechanism. There are other ways in which this information can be maintained by the environment.

The symbolic manipulation system Maple for example [5] manages storage using pointers in such a way that intermediate results exist uniquely in memory. Each Maple function has an associated remember table that contains handles to previous arguments and corresponding results. This sharing gives the remember option some of the properties of caching with dataflow information, as the data dependencies can be traced through the various tables. This results in a fairly efficient yet simple caching scheme. In such a representation, cache overhead is limited to lookup in a table of handles. A function will receive handles to its arguments, which are either newly calculated values or values already existing in memory. In the latter case, the argument handles will match some of those stored in the function’s remember table.

If the caching mechanism must look up actual values of arguments in the cache, there would be more overhead. In this scheme, only the tables associated with input identifiers will have to look up a value, and create a new handle in the table if the value is not yet available in memory. This corresponds to only comparing the input values with previous cached values in a dataflow environment (yet using more cache space than the MinCache or even the MaxCache scheme). To implement the various other caching schemes proposed for dataflow environments would require some very tricky manipulation of the dependency information that can be gathered from the handles, and is likely impossible for anything but a very simple symbolic manipulation environment.

In Maple, a symbolic manipulation environment, the operations are executed in a fixed sequence. This eliminates some of the problems with side effects, sequential execution and forced execution of nodes which trouble caching in a genuine dataflow environment. Results from different invocations, iteration steps etc. are stored in the same cache without distinction, so the problem of transferring dataflow information from one execution to the next does not occur.

A strategy similar to remember tables can be used to expand the proposed caching schemes by caching a number of previous results for each node. If sufficient memory is available, more than just the last result could be stored in a node’s cache. Some kind of unique identifier (such as the handle to this data in memory) can be associated with each value in the cache. The cache could be indexed by a combination of such identifiers instead of the argument values themselves (Figure 33). This will avoid comparing the data values in most cases; comparison will still be required at the input terminals. The identifiers could replace both time stamps and change tags; a new value is signalled by a new identifier.
Figure 33 Indexed tables of cached results

In the case of a dataflow environment that supports multiple (non-recursive) instances of a procedure, as well as for static iteration abstractions, caching multiple previous results for a subgraph would allow sharing results from different iterations or subgraph invocations. However, keeping track of the identifiers across invocations would become a complex task.
3.6 Manual integration of dependency information

A developer could naively attempt a manual approach towards integrating dependency information and caching, making 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 dependencies between the parts and the measurement settings. However, any change to the program is liable to change these relations. Keeping the conditional structures 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 displayed results correspond to the latest settings, he will be tempted to run the program more often than needed, possibly nullifying the intended time savings. Because of this, it is preferable by far to adapt the environment to automatically integrate data dependency information with caching.

The following example will demonstrate the complexity of manual caching. This example uses the LabView representation and terminology, but the same conclusions would apply to a completely text-based program.

Consider the simple network analyser system in Figure 34. The first component is a signal calculator that can create different types of signals. The blocks Generate and Setup respectively load the signal into the generator and set up the generator and measurement equipment. The results of the measurement block Measure are windowed and their FFT is calculated. From these the transfer function block is calculated using either of two formulae. In an automatic caching environment, this could be the entire program. The lines connecting Generate and Setup with Measure are ‘artificial’ dependency links, to make sure Setup is complete before measuring. These lines do not carry data but synchronising tokens. On a change in the signal or the set-up, a ‘changed’ token is passed along these lines to trigger a new measurement.

![Figure 34 A simple Network Analyser](Image)
The developer has a bit of extra responsibility here, in the presence of I/O or side-effects he must know which operations can safely be skipped, and must group operations and arrange the links accordingly. Making the environment determine this automatically, based on models of the instruments used, would be very complicated. This limits the possibility of transparent integration in existing dataflow environments, because programs would require some rewriting to be compatible with the new execution scheme.

To re-use previous values in a non-caching environment, the entire graph could be placed inside a loop. After comparing the new control values with the previous ones, the program decides whether to execute a part of the graph or to use the previous value. Although in this example the data dependency information is implicit in the dataflow representation, the programmer has to manually separate the graph into portions, according to their dependence on the control values, and route the intermediate values to the next iteration. This has to be repeated for each change to the algorithm.

Only part of the resulting structure is displayed here, yet it clearly demonstrates that clarity and modifiability are greatly reduced (Figure 35 shows the clause where Beta is the only changed input). This example only takes changes to the controls into account (the MinCache scheme), otherwise every single node would require a comparator and a link to the next iteration.

![Figure 35 Part of the code to avoid redundancy](image)
4. Performance and generic graph properties

This and the following chapter will identify some generalised graph properties that can be used to predict and explain the performance of caching schemes, and that would allow a developer to estimate which caching scheme, if any, would be most likely to benefit his application.

An evaluation of the performance of the caching schemes introduced in chapter 3 is presented, averaged for a large number of graphs and as a function of selected generalised graph properties. Little data is available on the performance increase achievable through caching as related to dataflow. The lack of libraries of suitable characteristic example algorithms for analysis was resolved by the use of simulations on random generated graphs. The method put forward in this chapter was inspired by [51]. The simulator is discussed, and a highly condensed overview of the simulation results is presented. A detailed discussion of this performance evaluation is offered in chapter 5.

The simulations show that no caching scheme is consistently superior, but nevertheless some rather well defined areas of method preference based on the generalised properties can be determined for most caching schemes. The simulations also show that certain ranges of graph architecture benefit little or not at all from caching.

A discussion of caching performance for a specific algorithm rather than in terms of general properties, follows in chapter 6.

4.1 Overview

4.1.1 Use of simulations

The lack of sample algorithms in a suitable dataflow format, and of representative user interaction profiles, precludes the possibility of arriving at valid generalisations from studying actual algorithms.

Related research [43, 44, 45] suggests the use of simulations, based on random generated graphs and user interactions, to obtain performance measures. Scheduling policies for the execution of parallel programs for example are also optimised on random
program graphs, although practical programs have non-random program graphs and execution patterns (it is often not feasible to show that a schedule is actually optimal). This method was adapted to the dataflow execution schemes presented in chapter 3 (also described in [46, 47, 48]).

The simulator was created to compare the performance and overhead of several caching schemes:

1. The **NoCache** scheme uses neither caching, nor comparisons of intermediate results to previous values.
2. In the **MinCache** scheme, the end nodes of threads are cached, and values assigned to input terminals are compared to previous values.
3. The **MedCache** scheme caches and compares values at the end nodes of threads.
4. In the **MaxCache** scheme, caching and comparing new values with their previous value is performed at every node.

The details of the simulations are discussed later in this chapter. First, the parameters selected to predict caching performance will be presented.

### 4.1.2 Choosing generalised parameters

To compare the performance of a caching scheme for different graphs, and to draw generalised, comprehensive conclusions, it is necessary to decide some quantitative properties that are representative enough to characterise each graph. These properties should ideally be easy to determine, yet possess a power to explain variations in performance between different graphs, and promote understanding of the factors affecting performance.

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 look up this node’s result in the cache (classed as a node property because it depends on the node’s data size).

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. To be useful as performance indicators, the parameters should be easy to derive (preferably easier than calculating the exact performance!). Some obvious parameters are the graph size (the number of nodes in the graph), the number of input terminals (the **InputFraction** or number of inputs relative to the number of nodes will be used from now on), the number of output terminals (**OutputNumber**), and the **BranchFactor** (the average number of output arcs per node).
The **ThreadFraction** (number of threads relative to number of nodes, including input terminals) is a parameter that was initially introduced to offer an explanation for the dependency of the performance of the caching scheme on the InputFraction, but it also happens to be useful to understand its dependency on the BranchFactor. The minimum number of threads in a graph equals the number of input terminals. The maximum number equals the number of nodes in the graph. This parameter appears to be a good indicator for caching performance, especially for the MaxCache and MedCache schemes.

Many other graph parameters could have been chosen. Topological graph theory literature ([18] for example) introduces many graph properties, such as graph connectivity, radius and diameter, but this field of study deals primarily with ways to represent geometric realisations of mostly undirected graphs, and there is little reason to suspect a connection between the performance of caching schemes and the intricacies of graph depiction on various surfaces. Due to the time and storage requirements involved in the simulations, only those graph parameters directly influenced or needed by the simulator were recorded.

The user 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**.

Representing user interactions by the ChangedFraction assumes no changes are made to the algorithm. An important reason for the use of caching is to ease the design and development phase of a system. However, the effect of caching on the time to recalculate an edited graph is hard to quantify (since the graph itself is not fixed), and this is not attempted in the simulator. The effect of changes to the input settings can be used as a coarse estimation instead.

A last factor that will influence performance is the **overhead importance**. This is the average effort for looking up a value in the cache, relative to the average effort to obtain this value by calculation (the simulator calculates the overhead as proportional to the data size). The overhead importance will depend on the average complexity of the calculations, the average data size, and the environment-dependent efficiency of cache lookup.

### 4.1.3 Effects of execution method and caching scheme on performance

Chapter 5 discusses the simulation results in some detail. A summary of the conclusions is presented here in advance to assist the reader through chapter 5 with an idea of where it is heading. Naturally, not all can be explained in a short summary.
Figure 36 Comparison of average cost and overhead for data driven and demand driven execution

Figure 36 shows the cost relative to the total cost for calculation without caching, and the overhead relative to the minimum overhead for caching without using dataflow information. Averaging all simulations, demand driven execution displays a lower execution cost on average than data driven execution, for all studied modes of caching.

The dataflow execution method does not appear to influence the relative merits of the various proposed caching schemes. For graphs with multiple output terminals, demand driven execution requires less calculation on average than data driven execution. The demand driven calculation cost is directly proportional to the number of nodes required to calculate the requested result (the graph fraction visited by the execution scheme), which is mostly determined by the number of output terminals (see Figure 37). Note that the simulations do not include dataflow extensions, which might otherwise also lead to the skipping of graph parts (see section 2.4.7).

Figure 37 Visited fraction vs. OutputNumber (independent of graph size)
Without any caching, the demand driven cost averages approximately 60% of the data driven cost (not shown), which corresponds to the average fraction of the graph visited during demand driven execution for the simulations.

Figure 38 compares the data driven calculation cost with the demand driven calculation cost, relative to the total calculation cost of the graph part visited during demand driven execution. This comparison shows that the savings achieved through caching are comparable for both schemes. The costs appear to be slightly higher under the demand driven scheme. This might well be due to a higher relative InputFraction of the graph parts visited by the demand driven execution scheme. As, due to the graph construction method, input terminals are the least likely nodes to be excluded from such a graph part, the ratio of input terminals to other nodes for the graph parts will on average be higher than for the complete graphs. The effects of the InputFraction on the cost are discussed in detail in chapter 5.

Since the dataflow execution scheme does not appear to influence the relative merits of the various proposed caching schemes, conclusions concerning the caching schemes can be derived from just the data driven case.
Figure 39 Average fraction of nodes which compare new values to the cache, for the 3 caching schemes

Figure 39 shows the average fraction of nodes that compare new values to previously cached ones for the different caching schemes. There is no exact correspondence between these values and the proportional overhead differences of the various caching schemes, as each overhead is influenced differently by other factors. (It may be confusing that the fraction of nodes comparing values is less than 100% for the MaxCache scheme, this is because output nodes do not compare their values to previous results in the simulator.)

Figure 40 Effect of ChangedFraction on Cost and Overhead averages

The average ChangedFraction is 50%, since all interactions are considered equally likely. Figure 40 shows the effect of a variation in the average ChangedFraction on the cost and the overhead.

The cost difference among the three caching schemes is mainly due to the effect of nodes producing the same result for different arguments. For a low ChanceFactor the costs will be almost equal to the MinCache cost (see Figure 41), whereas the difference in overhead remains.
By comparing only new values assigned to input terminals with the previous values, or new results with a previous one only for nodes at the end of threads, the required memory for caching is reduced as compared to a scheme where all results are cached. The required cache space, relative to the cache space needed in the MaxCache scheme, is proportional to the ThreadFraction of the graph. The reduction in required cache space can thus be considerable.

When overhead and cache size are ignored, the MaxCache scheme always performs equally or better than any other scheme. However, when both cost and overhead are considered, no single caching scheme is consistently superior.

### 4.1.4 Areas of caching scheme effectiveness

To determine which caching scheme is best in a given situation, both node properties, structural graph properties, interaction properties, and overhead importance must be taken into consideration, since no single caching scheme is consistently superior.

This section proposes that in this multi-dimensional space, regions can be identified where a given caching scheme is superior to all others.

The simulation parameters chosen to describe graph properties are: the number of nodes including input terminals (graph size), the relative number of input nodes (InputFraction), the average number of arcs for each node (BranchFactor), the number of output terminals (OutputNumber), and the number of threads relative to graph size (ThreadFraction). Other parameters like calculation cost distribution are ignored in the simulations. As will be demonstrated in chapter 5, graph size has little effect on the relative cost and overhead value. The OutputNumber only matters under demand driven execution, and the dataflow execution scheme does not influence which caching method is most effective.

The InputFraction, ThreadFraction and BranchFactor are not independent but interrelated. From these last three the ThreadFraction is more general and most capable of explaining the variations in cost and overhead, however the InputFraction is easier to grasp and produces somewhat more easily readable figures.

When the total cost (calculation cost + caching overhead) is considered, a single parameter is insufficient to fully describe the performance of all caching schemes. The
MinCache cost for example depends on the graph structure, while the MinCache overhead is determined only by the number of input terminals and does not depend on other aspects of the graph structure at all.

The sum of the calculation cost and caching overhead is compared for the three caching methods, and for total recalculation without caching.

Areas of caching scheme efficiency are represented on InputFraction - ChanceFactor and ThreadFraction - ChanceFactor figures (these graph parameters have the advantage of being restricted in value to the 0-1 interval).

The selection of the ChanceFactor as second parameter is somewhat arbitrary and the ChangedFraction or overhead importance could have been chosen just as well. The figures are shown for different overhead importance values (1%, 50%, and 100%). The average ChangedFraction is 0.5 (50%) in Figure 42 (all input states are considered equally likely). The influence of varying the ChangedFraction is presented in Figure 43 and Figure 44.

The areas on Figure 42 show where the various methods are more effective for a larger percentage of the graphs than any other methods. These figures are interpolated so the grey values are only an indication. The darkest area means MaxCache is the most effective scheme, lightest means caching is ineffective. Dark grey and light grey represent the MedCache and MinCache scheme, respectively.
It is clear that fairly distinct areas can be identified as being most effective for a particular caching method. It seems that the MedCache scheme is never more effective for the majority of the simulated graphs (however, other factors, such as the required cache space, are not taken into account on these areas of method effectiveness).

An overhead importance of 100% corresponds to a total caching overhead that equals the total calculation cost. In an environment that does not make use of dataflow information, such conditions would completely exclude caching as a measure to increase performance. However, Figure 42 indicates that a caching scheme assisted by dataflow information can still perform effectively under these conditions.
In Figure 43 and Figure 44, the effect of choosing a lower or higher average ChangedFraction is represented. Changing only a small fraction of the input terminals during each interaction will benefit caching, while changing most input terminals during each interaction will have an adverse effect on the effectiveness of caching.

Figure 43 and Figure 44 show the (interpolated) areas of method effectiveness for average ChangedFractions of 25%, 50%, and 75%, for an overhead importance of 1% and 50%. The effects of the ChangedFraction are more apparent for higher overhead importance (Figure 44).

*Figure 43 Influence of the ChangedFraction, overhead importance 1%*
Figure 44 Influence of the ChangedFraction, overhead importance 50%

Note that the area of best performance for the MaxCache scheme changes little with the ChangedFraction. The performance of the MaxCache scheme depends more on the ChanceFactor and less on the number of changed inputs. The ChangedFraction has considerably more effect on the other schemes.

The area of effectiveness for the NoCache scheme is almost identical for the medium and high ChangedFraction, but for a low ChangedFraction, the area where some form of caching is not effective reduces dramatically. This is true with both low and high importance accorded to caching overhead.

For higher ChangedFraction values, not caching is the most effective scheme in areas where the MedCache and MinCache scheme are effective for a lower ChangedFraction. This can be explained as follows: for a low ChangedFraction, the calculation cost is almost identical for the three caching schemes, but the lower overhead of the MedCache

58
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 generate 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 any caching profitable.

4.1.5 Summary of caching scheme performance results

4.1.5.1 NoCache scheme

Under the data driven execution scheme, the entire graph is recalculated during each user interaction. In the demand driven execution scheme, only operations that lead to the requested output are executed. There is no caching overhead.

The calculation cost in the NoCache scheme is never lower than in the schemes that use caching. Therefore caching can only be less effective than the NoCache scheme when the combination of calculation cost and caching overhead exceeds the NoCache cost. This can occur for two reasons: the caching overhead is large or the calculation cost is hardly reduced by caching. A high InputFraction or ThreadFraction in addition to a low ChanceFactor corresponds to both a high calculation cost and a high caching overhead for the MedCache and MaxCache schemes, so caching is less effective then.

4.1.5.2 MinCache scheme

In the MinCache scheme, new values assigned to input terminals are compared to the previous values. Only operations that depend on a changed input are recalculated. Only values at the end nodes of threads must be stored in a cache. The comparison of new input values is the only caching overhead. The cost depends mostly on the number of changed input values, and is equal to the NoCache cost when all input terminals changed. So unless the overhead can be ignored, the NoCache scheme will be more effective than the MinCache scheme when all input terminals always change during a user interaction.

The MinCache cost also depends on the graph structure, and increases for a higher ThreadFraction, but not as much as the other caching schemes. It is independent of the ChanceFactor.

4.1.5.3 MaxCache scheme

In the MaxCache scheme, all nodes store their result in a cache, and compare a newly calculated result to a previous one (in addition to verifying the newness of input values). The overhead will be proportional to the number of updated input terminals, plus the number of recalculated nodes. Properties that increase the number of recalculated nodes will increase cost and overhead equally, so conditions where caching hardly reduces the calculation cost will also lead to the highest caching overhead.

Unlike the MinCache scheme, the MaxCache scheme can still benefit from caching when all the input values are changed, as long as the ChanceFactor is nonzero. From all caching schemes, the MaxCache scheme always has the lowest calculation cost, but also
the highest overhead. This scheme is most effective for low ThreadFraction values and a high ChanceFactor.

4.1.5.4 MedCache scheme

The MedCache scheme caches the same intermediate results as the MinCache scheme. Since the end nodes of threads cache their values in the MinCache scheme, these end nodes might as well compare new values to their cache. Nodes that get only unchanged values for arguments are not recalculated. This scheme is less effective than the MaxCache scheme in avoiding unneeded operations, but the overhead is lower, especially for low ChanceFactor values. It is the most effective scheme only in a minority fraction of cases, but spread over a large area of parameter values. It seems that generalised graph properties are not entirely sufficient to distinguish the graphs for which the MedCache scheme is most effective. Presumably the distribution of the thread end nodes has some influence.

The MedCache scheme’s main advantage is that it requires only the same amount of cache space as the MinCache scheme, while being closer to the MaxCache scheme in performance. This scheme is therefore most useful as a replacement for the MaxCache scheme when memory constraints prohibit its use, and the ChanceFactor is sufficiently high to make the MinCache scheme less attractive.

4.1.5.5 Effects of dataflow information

Assuming unlimited cache space is available, caching without using dataflow information will be at least as effective in avoiding unneeded operations as the MaxCache scheme. When several nodes execute the same operation, it is possible they coincidentally execute with the same argument values. The MaxCache scheme will not be able to detect this. Also, values that were used several interactions ago might turn up again. By storing values from all previous interactions together in one cache, an operation will only execute once for each occurring set of argument values.

The caching overhead in this dataflow-less scheme is caused by searching the cache for each combination of operation and arguments that is encountered during execution. This overhead will depend on the cache structure and organisation, and on the number of entries in the cache. During the search, an argument will be compared at least once to a cache entry, so the total data size can be taken as an approximate minimum value for the caching overhead generated when no data dependency information is used (however, the real value will also depend on the result of the comparisons).

When unlimited cache space is available and caching overhead is insignificant, the use of dataflow information offers no advantages. However, when available cache memory is limited or the overhead is important (possibly even larger than the calculation cost), caching with the use of dataflow information can increase performance for systems where ordinary caching would be useless.

4.1.6 Real-time identification of the optimum caching strategy during execution

If performance data can be collected during execution, the caching scheme could be adjusted to optimise the caching performance based on the actual execution costs, user interactions, available memory, etc. The caching strategy could even be individually
tailored to each node, rather than on the graph level (further study would be required to identify the benefits of this).

It would be preferable if each node could determine the best strategy for itself, based on its own performance measurements. An individual node can gather a lot of information to base a caching strategy on. The overall graph structure, and the interaction properties cannot be determined locally by an individual node. However, relevant information on these properties can be derived from the frequency of changed arguments versus unchanged arguments arriving at the node, and on how often the node has to execute versus how often execution is avoided. By comparing its input terminal set with the input terminal sets of its children, the node can determine whether or not it is a thread end node.

The node, if it starts out by caching its result and comparing a new result with the previous value in the cache, can determine its own local ChanceFactor. If sufficient memory is available to cache all results during a number of executions, and if the resulting overhead is acceptable, measured ChanceFactors can readily be obtained for every node in the graph. If neither condition is achieved, an optimisation mechanism can only gather data on a few nodes in turn.

If a node can access system performance information, it holds accurate data on the processing cost to execute its operation and to compare its result with a cached value. However, it is not correct to base the caching strategy of the node on these two values, because comparing the node’s result with the node’s cache never benefits the node itself, only (possibly) its children. The cost reduction possible by finding the result to be unchanged, will depend on all downstream execution costs, and on the participation of this particular result in the current downstream execution behaviour. A node could make only heuristic decisions based on these local performance measures.

Thus, nodes cannot conclusively determine their best caching strategy from performance information gathered locally by themselves. For example, we have seen that in some circumstances the MinCache scheme is more costly than not caching at all, due to the overhead of comparing new input terminal values with previous ones. It is not possible for an input terminal to determine this independently, based solely on the input terminal’s own history. A performance evaluation on the graph level is required.

Fortunately, the number of choices for each node is limited. If the node maintains a cache, it can decide whether to compare a newly calculated result with the previously cached value, or instead determine the result’s ‘changed’ tag from the ‘changed’ status of its arguments. If a node is not a thread end node, maintaining the cache is useless without comparing it to new results. For such a node, the decision to cache can be based on the probability of the node to return the same results for changed argument values, and on the comparison cost relative to the possible downstream cost savings.

The benefit of caching thread end nodes does not solely depend on the probability of the node to return unchanged results for changed argument values. If global performance information is unavailable, the proportion of unchanged arguments in the total number of arguments received by the node can help determine locally whether caching is an effective strategy.

In summary, the caching strategy can be adjusted at runtime based on actual performance measures gathered during execution, if only to reconsider the assumptions made previously. Reconfiguration of the caching strategy should preferably take the entire program into account.
4.2 Establishing performance through simulations

4.2.1 Overview

The performance and overhead of a specific scheme will not only depend on the properties of the dataflow graphs (such as the distribution of computation times and data sizes, the number of arcs and the graph connectivity), but also on the unpredictable sequence of user interactions. This hinders the evaluation of caching performance.

Related research [49, 50, 51] suggests the use of simulations, based on random generated graphs and user interactions, to obtain performance measures. This method was adapted to several dataflow execution schemes, including those from [52, 53, 54].

In this study, a simulator was created to compare the performance and overhead of several caching schemes:

1. The **NoCache** scheme uses neither caching, nor comparisons of intermediate results to previous values.
2. In the **MinCache** scheme, the end nodes of threads are cached, and values assigned to input terminals are compared to previous values.
3. The **MedCache** scheme caches and compares values at the end nodes of threads.
4. In the **MaxCache** scheme, caching and comparing new values with their previous value is performed at every node.

A node could cache more than just the last value (as in [5]), thus increasing the probability of finding a result in the cache, but this possibility was not explicitly considered in the simulation, because simulating this would be impractical: the probability of finding the current arguments of an operation among the cached values of previous executions depends strongly on the implementation details of the caching environment. Suppose for example that each node caches the same number of values. The number of previous executions recorded in a cache depends on previous cache hits (more hits mean more previous executions are recorded in the same cache), and will differ for each node. One of the advantages of using dataflow information is reducing the number of entries in the cache anyway. For these reasons, caching multiple results for each node was not simulated.

4.2.2 Creating dataflow graphs

Graphs are created randomly with a predetermined number of nodes and predetermined number of input terminals. For each node a data size and a calculation cost are chosen randomly from a given range. Conditional and iterative functions are not modelled. Each node has a randomly associated probability of yielding the same result on execution (this could also be interpreted as a simple approximation of multiple-value cache hits). The average value of this probability, the **ChanceFactor**, is chosen at the graph level.

From the input terminals, links are made to randomly selected nodes. The exact number of arcs from each node is random, each node has at least one output arc, and the probability of extra arcs is determined to average to a predetermined graph parameter, the
BranchFactor. Each node has exactly one output result in the simulations (in reality, a node might have several output results).

The 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 output arcs per node is again determined randomly based on the BranchFactor. The last remaining node is removed, and its arguments are chosen as the output nodes of the graph. Due to the construction process, the number of output nodes is not predetermined. Note that not only nodes without children can be output nodes. Since their results are not used by other nodes, nodes without children have no purpose but as output nodes.

4.2.2.1 Limitations of the graphs created by the simulator

It is possible that more specific results could be obtained from simulations with a graph structure which matches more closely the algorithms specific to the field under study. Considering measurement, data processing and data representation as the main parts of the algorithm, the following points could be considered (but were not) in the simulations:

- Distribution of data size in the graph.
  
  Most probably input settings are scalar values, not large data structures. Generated signals and measured data, on the other hand, tend to be larger arrays. Presumably, data processing operations will transform the measured data into more or less equally sized or smaller arrays. The data size in the data representation part varies depending on the type of representation.

  In the simulations performed here, data size was distributed at random, because the previous arguments are application-dependent and too fuzzy to incorporate. The difference between the smallest and largest data size is limited to a factor of 100 in the simulations, to make the influence of other factors more apparent.

- Complexity of the operations.

  Duration of measurements depends on the amount of data and on the sampling rate. The calculation time for data processing operations also depends on the data size. Instead, in the simulations, data size and calculation cost are assigned as independent node properties.

  For the cost distribution, the 90/10 rule could be assumed (90% of the operation cost is concentrated in 10% of the nodes), however, applicability of this rule on dataflow diagrams is unknown and somewhat doubtful. An uneven distribution of node cost could greatly reduce the effect of caching, if only low-cost operations can be avoided by caching. The popularity of caching in several application domains [4,44] seems to indicate that such distributions are rare. In the simulator, the costs are assigned at random from a range with a factor of 100 as difference between the smallest and largest cost.

- Distribution of input terminals.

  Depending on whether the process is more measurement-oriented or more representation-oriented, the majority of input terminals will be located near the
sources or the sinks of the dataflow graph, or more or less evenly distributed between them. This is likely to considerably affect the impact of a change to one of the settings. In the simulator, the input terminals are randomly distributed over the graph.

- Blocks in the dataflow graph.

In real applications, the dataflow graph is likely to consist of several blocks of highly interconnected nodes, but with relatively few connections in between the blocks (for example; a measurement block, a data processing block and a data representation block). It is not obvious what the effect of such a distribution would be on the performance, a graph consisting of such blocks might behave similarly to a graph with single nodes substituted for the blocks.

- Distribution of node arguments.

The average number of node arguments will equal the average number of outputs, but due to the graph construction process, nodes closer to the outputs of the graph will, on average, have slightly more arguments than nodes closer to the input terminals. There is no reason to assume this has any significant effect on the results.

These application-specific factors were not implemented in the simulations to avoid influencing the results by any unfounded preconceptions about the application domain would influence the results. Attempting to incorporate them would have been mostly arbitrary since no data on representative distributions for a specific application domain is available. No attempts were made to quantify the distribution of properties such as data size or calculation cost.

Specific algorithms are analysed in chapter 6, and compared to the simulation results. The comparison indicates that the simulated results are equivalent to actual ones, but also that performance is affected by an uneven distribution of properties.
4.2.3 Creating user interactions

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 data driven scheme, these are simply the nodes whose arguments are unchanged.

The simulator compares data driven and demand driven execution of the same graphs, with the same pattern of changed input terminals. It can be argued (see section 2.4.7) 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: comparison should involve the same graphs. Since the simulator does not generate dataflow extensions, the demand driven scheme is only relevant when the system has more than one output node. Consequently, graphs with several outputs had to be used, even though these are unlikely to be used in a data driven environment. This “unfair advantage” of the demand driven scheme should be kept in mind when comparing performance of the data driven and demand driven scheme.

For each interaction, one of the outputs is selected at random, and all its upstream arcs are ‘primed’. Only activation tokens passing through primed arcs will add to the calculation cost and caching overhead data collected on demand driven execution. Arcs are ‘unprimed’ once an activation token has passed through them.

In the demand driven mode, only a 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 and each node is visited only once during each run). 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).

The minimum number of nodes that must cache their value are the outputs of a ‘thread’ of nodes depending on the same inputs, these nodes are marked during the graph generation phase by compiling and comparing the set of input terminals each node depends on. Caching is optional for other nodes, as long as thread output nodes are cached, nodes that depend on unchanged input terminals will not have to be recalculated. Caching the other nodes is useful only when operations produce unchanged results for changed arguments.

4.2.3.1 Cost determination

The calculation cost is subdivided according to caching scheme in:

- NoCache Cost (the cost for executing the entire graph without caching),
- MinCache Cost (the cost for executing the graph when taking into account which input terminals have new values, and caching only the necessary minimum number of nodes),
• MaxCache Cost (the cost for executing the graph when each intermediate result is compared with its previous value), and
• MedCache Cost (the cost for executing the graph when intermediate results at the end of a thread are compared with their previous values).

The simulated costs are normalised for graph size and average node calculation cost.

### 4.2.3.2 Overhead determination

The exact overhead is difficult to determine, as it will depend on how data comparison is implemented in the specific environment. A full element by element comparison is usually not required to determine whether two arrays are different. A caching overhead proportional to the data size is taken as a worst case estimate by the simulator, and whether the compared data was identical or different is not taken into account.

The overhead generated by the fetching and writing of data from and to the cache is not considered separately from the comparison overhead. In a dataflow environment, intermediate results have to be stored and retrieved even when no explicit caching is used. Results need to be stored at least until the downstream nodes are ready to accept the data. In the proposed schemes, the extra overhead is due primarily to the comparisons with a previous result.

In data driven mode, the MinCache overhead is the accumulated data size of the input terminals, the MaxCache overhead is the accumulated data size of all nodes that are executed, and the MedCache overhead is the data size for results on the end of threads. The overhead is equivalent in demand driven mode, but limited to the “visited” part of the graph.

To determine the benefits of using dataflow information, a measure would be needed for the caching overhead when dataflow information is not used. This overhead is impossible to calculate, as it will depend not only on the number of cache entries, the number of calls for each procedure, the cache management and lookup method, but also on the user interaction history.

Consider for example a node whose result is used by several other nodes, as in Figure 45. If the caching scheme uses none of the available dataflow information, then before node B is executed, the cache will have to be searched for an entry with the result of node A and B’s other argument together with the operation for B. Another search concerning the argument of A will occur before the execution of node C. Thus, the cache is searched twice for the value of A.

![Figure 45 Node used as argument by several others](image-url)
By using dataflow information, one comparison of A’s result with its cached value suffices to settle the validity of B and C’s cache as far as A is concerned. Even this one comparison is unnecessary if A’s arguments were unchanged.

Without data dependency information, the overhead will at least consist of comparing argument values with the cached arguments for each operation (though this depends on the result of the comparison, comparing one argument may suffice to determine the absence of the result in the cache). The very minimum will be the cost of comparing the input terminal values with the previous ones (assuming the rest can be derived through the collected dependency information).

In the simulations, we use the sum of the data sizes of all nodes as a reference (the cost of comparing each value with a previous one exactly once).

Overhead for comparing validity tags is considered insignificant next to the overhead of comparing the data, and is ignored.

4.2.4 Detailed description of the simulation process

4.2.4.1 Steps for creating a graph

The detailed procedure followed to create graphs for the simulator:

Create size+1 nodes.
Assign random calculation cost, data size and ChanceFactor to each node.
Take the first inputnumber nodes as input terminals (calculation cost is set to 0).

For (each input terminal)

a link from the input to a random non-input node is added to the linktable.
a random number of additional links to random non-input nodes are added to the linktable. The average number of links is determined by the BranchFactor.

Repeat

A node with input links but without output links is selected at random.
A random, nonzero number (its average determined by the BranchFactor) of links from this node to random nodes without output links is added to the linktable. This ensures the generation of an acyclic graph.
Until (only one node without output links remains).

Remove the last node. Its arguments become the graph’s output terminals.
Determine the set of input terminals each node depends on, and identify the different threads.
Calculate average calculation cost, data size and ChanceFactor, and the number of threads, outputs and the BranchFactor for the graph.

4.2.4.2 Steps for creating an interaction

The simulator sets up each interaction as follows:

Each node has three “changed” tags, one for each caching scheme: changeMax, changeMin, and changeMed

The number of inputs to be changed and the changed inputs are chosen randomly. The “changed” tags of these inputs are set to true, and to false for the other inputs. The links from the changed nodes are marked as inconsistent (for demand driven execution).
One of the output nodes is selected at random and primed. This primes all nodes leading to this output to be ‘executed’ in the demand driven scheme.

The data size of all changed inputs is added to the overhead counters. There are six overhead counters, one for each caching scheme and execution mechanism; OverheadMax, OverheadMin, and OverheadMed, and demandOverheadMax, demandOverheadMin, and demandOverheadMed. The demand driven counters are only incremented with the data size of primed nodes.

Run the interaction.

Eight counters collect the data driven and demand driven execution cost for the three caching schemes and for no caching. Additionally, the numbers of changed inputs, and the number of nodes delivering an identical result for a changed argument are counted. Some data describing the graph is stored with the interactions, consisting of the graph size, inputnumber, outputnumber, branchfactor, and threadnumber, the average calculation cost, data size and chancefactor, minimum required cache size and the experiment name.

Running the interaction:

Put an activation token on the output arcs of each input terminal.
Repeat
   Find a node that has a token on each input arc.
   Determine the status of the “changed” tags, the various calculation costs and overheads for this node and update the counters for relevant execution mechanisms and caching schemes.
   Remove the tokens from the node’s input arcs.
   Place tokens on the node’s output arcs.
Until no more executable nodes are found.

Determining the costs, overheads and “changed” tags of a node:

Each node has three “changed” tags; changedMax, changedMed and changedMin. The status of these tags is determined by the status of the corresponding tags of the node’s arguments. If one of the node’s arguments has an active “changed” tag, the node becomes “changed” for that caching scheme as well. Only if all arguments are unchanged for the caching scheme, is the node marked as unchanged.

CostNoCache is incremented by the node’s calculation cost.
If (changedMin) CostMinCache is incremented by the same amount.

If (changedMax)
   Increment CostMaxCache, and increment OverheadMaxCache by the node’s data size. A random function determines whether the new result will be identical to a previous one, based on the node’s chancefactor. ChangedMax is adjusted accordingly. If the node’s value really changed, the children of this node are no longer consistent with the input settings, and are so marked (this is important for demand driven execution).

If (changedMed)
   Increment OverheadMedCache only for nodes at the end of a thread, where a new result is compared to the cache in this scheme. Only in these nodes can an active changedMed tag become inactive, based on the result of the random function used for the MaxCache scheme.

If (the node is primed) it will be executed by the demand driven scheme. The node’s calculation cost is added to the demand driven NoCache cost counter.
If (the node has been marked as inconsistent)
One of its arguments changed since the latest demand driven execution of the node, and it must be recalculated. The node’s calculation cost is added to the demand driven MaxCache, MedCache and MinCache cost counter. The data size is added to the demand driven MaxCache overhead counter, and to the MedCache overhead counter if the node is an end node of a thread.

If (the node has not yet been marked as inconsistent)

The cost and overhead counters are updated according to the status of the "changed" tags, as in the data driven case. The node is unprimed, and marked as consistent with its arguments.

To determine whether a node is an end node of a thread, the program determines for each node the set of input terminals it depends on. A node is an end node of a thread if one of its children depends on a different (larger) set of input terminals.
5. Comprehensive performance evaluation

This chapter offers an in-depth analysis of the simulation results. It is proposed that performance will depend on four distinct types of properties; node properties, graph properties, interaction properties and relative overhead importance.

The graph properties are the most difficult to express by generalised parameters. A histogram analysis suggests that the proposed parameters cover most variations of the graph structure that affect performance. It was hoped that a single graph parameter could have been found to adequately describe the influence of the graph structure. Unfortunately, the ThreadFraction parameter introduced in this chapter, while useful to explain caching performance, is unable to predict all performance aspects (for example the MinCache overhead is only dependent on the InputFraction). It also behaves rather peculiarly and produces figures that are difficult to interpret.

Areas of caching scheme effectiveness can be determined in the properties space.
5.1 Introduction

The figures in this section show values based on a set of some $10^4$ (actually 30.986) simulations. Some figures are based on smaller subsets that satisfy certain properties. For each graph, a sequence of at least 20 interactions with random input changes is recorded. Graph properties are varied in several experiments to determine their influence on the usefulness of caching.

The InputFraction, the ChanceFactor, the graph size, and the BranchFactor can be controlled. Other parameters vary randomly due to the graph and interaction generation methods.

Remember that the simulated caching schemes were called:
1. NoCache: no caching is performed, no comparisons to previous values are made
2. MinCache: caches the end nodes of threads, and compares the values assigned to input terminals only
3. MedCache: caches and compares values at the input terminals and at the end nodes of threads
4. MaxCache: caches and compares new values with their previous value at every node.
5.2 Data driven scheme

5.2.1 Calculation cost

5.2.1.1 Cost dependency on the ChanceFactor

After normalising the cost by average node calculation cost and graph size, the only node property influencing the caching scheme is the ChanceFactor, the probability level that nodes produce the same results for changed inputs. If this probability is zero, the three caching schemes should result in identical calculation costs. A low ChanceFactor (corresponding to most executed nodes producing changed values) will yield a MaxCache cost almost as high as the MinCache cost, as shown in Figure 46. The difference between MaxCache cost and MinCache cost increases with increasing ChanceFactor, since the MaxCache scheme detects the nodes that produce the same results for changed inputs. The MedCache scheme results in a calculation cost somewhere in between those of MinCache and MaxCache.

![Figure 46 Cost vs. ChanceFactor](image-url)
5.2.1.2 Cost dependency on the InputFraction

Contrary to our initial expectations, a graph with only a few inputs benefits the most from the caching schemes. This can be attributed to the random distribution of the input terminals in the simulated graphs, which is not how one would usually imagine a graph with few input terminals. Figure 47 shows that the higher the InputFraction (number of input terminals relative to the number of nodes and input terminals in the graph), the higher the cost. 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. Figure 47 shows the cost for graphs with an average BranchFactor fixed at 1.2 (the actual average number of arcs may naturally vary slightly from the reference value used during graph construction).

The calculation cost for the MaxCache scheme is the most sensitive to variations of the InputFraction (as it was to the ChanceFactor). Detecting nodes that produce the same results for changed inputs appears more effective for graphs with long threads.

For high InputFractions, the majority of nodes are input terminals and the costs are virtually identical for all caching schemes.
5.2.1.3 Cost dependency on the BranchFactor

Increasing the BranchFactor (with constant InputFraction) decreases the benefits from caching, as expected. Increasing the BranchFactor will cause an increase in graph interconnection, and the graph will consist of more, shorter threads (in Figure 48 the InputFraction is held constant at 25%).

In the simulator, the BranchFactor has a maximum value for each given number of nodes and input terminals, in which all nodes are interconnected and each node depends on all input terminals. All nodes then belong to one single thread.

If $i$ is the number of input terminals and $n$ the number of other nodes in a graph (excluding input terminals), the maximum BranchFactor value can be calculated as $\frac{n}{n+i} \left( i + \frac{n-1}{2} \right)$ assuming each node produces only one result, which is used at most once by each other node as an argument (as is the case in the simulator).

This result is easily obtained: in a maximally interconnected graph, each input terminal will have $n$ children. Since there are no loops, the $k^{th}$ node will have $n-k$ children. Hence the total number of arcs is $n* \sum_{k=1}^{n-1} k$. In Figure 49, there are 2 input terminals and 3 other nodes, the maximum number of arcs is 9, hence the maximum BranchFactor is $\frac{9}{5}$.
In reality, nodes can have multiple output results, and a result could be used as argument more than once by the same node. For the subset of graphs in Figure 48 and Figure 50, with 10 input terminals and 30 other nodes, this maximum value is 18.4.

Caching is almost useless in a graph with maximum BranchFactor, since a change to any one input always requires recalculation of the entire graph. Figure 48 shows the calculation costs for the caching schemes approaching the NoCache cost for a high BranchFactor. However, these unrealistically high BranchFactors are of little relevance to measurement environments.

Figure 50 shows how the ThreadFraction (the number of threads relative to graph size) increases with the BranchFactor, then reaches a maximum where there are almost as many threads as there are nodes in the graph, and then decreases as larger portions of the graph depend on all input terminals (this figure compares graphs with 10 input terminals and 30 other nodes and graphs with 25 input terminals and 75 other nodes).

It seems the BranchFactor is an almost size-independent parameter. Normalising the BranchFactor, by expressing it relative to its maximum value, does not appear to improve the generalising power of the parameter, quite the contrary in fact, as indicated by Figure 51. Additionally, the maximum BranchFactor value depends on the number of outputs of each node, and can only be determined when this number is fixed and known (as in the case of the simulations, where it is fixed to one).
Figure 51 Effect of normalising the BranchFactor, for different graph sizes

5.2.1.4 Cost dependency on ThreadFraction

No other property predicts the MaxCache cost as well as the ThreadFraction. Except for a very high BranchFactor, the calculation cost increases and caching efficiency decreases with increasing ThreadFraction (Figure 52). The MaxCache scheme is most effective in graphs consisting of few, large threads, where each thread depends on a small fraction of the input terminals. For the other caching schemes a minimum is observed in the graph. A graph with a single input terminal for example will have a very low ThreadFraction value. Such a graph consists of one thread, so the MinCache and MedCache scheme will both result in complete recalculation for every input change. A graph with few threads will have few thread end nodes besides the input terminals, so for low ThreadFraction values the MedCache cost is very close to the MinCache cost. For high ThreadFraction values, thread end nodes will be numerous and the MedCache cost will be closer to the MaxCache cost.

For very high BranchFactor values, the ThreadFraction decreases with the BranchFactor while the cost continues to increase (see Figure 53 for an impression of this behaviour).
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. The ThreadFraction thus appears to be a useful graph structure parameter. It covers variations in both InputFraction and BranchFactor (see Figure 50 and Figure 54). In addition, it offers an intuitive understanding of the influence of graph structure on the effectiveness of caching. Unfortunately, one ThreadFraction value corresponds to different costs for low and high BranchFactor values respectively. Also, as we shall see, it is unsuitable for predicting the MinCache overhead.
Figure 54 ThreadFraction vs. InputFraction

Figure 55 and Figure 56 show the MaxCache cost (which, from all costs, is the most sensitive to graph structure variations) as a function of ThreadFraction. These figures demonstrate that the cost varies almost linearly with the ThreadFraction in most of the range, and a different number of output terminals, a variation of the InputFraction, of the BranchFactor (not too high), and a variation of the graph size result in almost identical MaxCache behaviour relative to the ThreadFraction. Note that the rate of change will still depend on the node property ChanceFactor and on the interaction property ChangedFraction.

Figure 55 MaxCache cost vs. ThreadFraction for 3 experiments
5.2.1.5 Limitations of generic graph structure parameters

The graph structure parameters, even the ThreadFraction, can give only an average indication of performance. For example, the two graphs in Figure 57 have the same size, InputFraction, OutputNumber, BranchFactor and ThreadFraction. Suppose all nodes have the same calculation cost. The MaxCache performance is completely different for the two graphs. Suppose the input value has changed. If node E were to return an unchanged result in the graph on the left, nodes A, B and C and D could be skipped, saving 80% of the cost, whereas in the graph on the right, it would require three simultaneously unchanged results, A, B and C, to skip node D and E, and save only 40%. An unchanged node D would save 20%. The performance difference between the two graphs in Figure 57 is caused by the uneven distribution of the number of branches.

5.2.1.6 Cost dependency on ChangedFraction

The fraction of changed input values (Figure 58) is the only interaction parameter affecting the simulated cost. MinCache cost varies from 0% of the NoCache cost when no inputs have changed to 100% when all inputs have changed. The MaxCache cost is usually smaller than the MinCache cost, and is more dependent on the graph parameters. Note that MaxCache and MedCache cost can be smaller than NoCache cost even when all inputs
have changed. This can be attributed to the skipping of part of the graph due to nodes producing unchanged results upon execution.

![Figure 58 Cost vs. Changed Inputs](image)

*Figure 58 Cost vs. Changed Inputs*

The cost increases faster than linearly with the ChangedFraction. The costs for the various caching schemes differ most for high ChangedFraction values. Lower values result in barely different costs for the different schemes.

The costs as a function of the ChangedFraction will depend on the node and graph parameters as described earlier.

![Figure 59 MaxCache cost vs. ChangedFraction vs. InputFraction](image)

*Figure 59 MaxCache cost vs. ChangedFraction vs. InputFraction*

Figure 59 illustrates how the InputFraction influences the MaxCache cost behaviour as a function of the ChangedFraction. The nature of this influence was already depicted in Figure 47. The effect of variation of other graph parameters is similar.

Note that all the above figures show average values for a multitude of graphs, and that specific individual graphs can be expected to behave differently.
5.2.1.7 Cost dependency on graph size

The calculation cost will of course increase with the number of nodes (input terminals excluded), but if all size-dependent values are normalised for the number of nodes, normalised performance measures appear to be almost size-independent (see Figure 60 and Figure 61).

![Figure 60 MaxCache cost vs. ChangedFraction and graph size](image1)

*Figure 60 MaxCache cost vs. ChangedFraction and graph size*

The MaxCache cost is shown here because it was found to be the most sensitive to other graph structure variations. Results for other costs are comparable.

![Figure 61 MaxCache cost vs. InputFraction and graph size](image2)

*Figure 61 MaxCache cost vs. InputFraction and graph size*

5.2.1.8 Cost dependency conclusions

The effect of caching on the data driven calculation cost is influenced by the properties of the dataflow graph.

81
Concerning the node properties, the simulations confirm that comparing intermediate results beyond the input terminals will reduce the calculation cost only when nodes will sometimes produce the same result for changed arguments. The extra calculation cost reductions for MedCache and MaxCache appear to be proportional to this probability.

The graph property that determines cache efficiency appears to be the number and size of threads in the graph, and the relative number of input terminals each thread depends on. If each node depends on most input terminals, the concept of threads loses its significance. For realistic graph structures however, the ThreadFraction proves to be a reasonable indicator for cache efficiency, particularly for the MaxCache scheme.

### 5.2.2 Required cache space

So far only calculation cost was considered. Another important performance factor is the cache space required by the caching scheme. The MaxCache scheme requires all intermediate results to be cached. The MedCache and MinCache scheme only cache the results of the end nodes of a thread, and thus require less cache space.

The amount of cache space required for the MedCache and MinCache scheme depends on the graph structure: as the MedCache and MinCache scheme cache only the end nodes of threads, the cache space required will be proportional to the number of thread end nodes in the graph. Figure 63 shows the cache space required as a function of the ThreadFraction.
The required cache space is not influenced by the execution scheme, and is the same for demand driven execution. It behaves just like the ThreadFraction to variations of InputFraction or BranchFactor.

### 5.2.3 Caching Overhead

The caching overhead is expressed relative to the overhead for comparing all intermediate results with their previous values. This reference value is used because the caching overhead - when dataflow information is not used - cannot be determined. The chosen reference assumes each intermediate result is compared exactly once to a cached value, the minimum 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 63 MinCache and MedCache space requirements vs. ThreadFraction**
5.2.3.1 Overhead dependency on InputFraction

![Graph showing overhead vs. InputFraction]

Figure 64 Overhead vs. InputFraction

The MinCache overhead is caused by comparing values for the input terminals, so naturally this overhead is proportional to the InputFraction (see Figure 64). The relative overhead is about $\frac{1}{2}$ of the InputFraction in Figure 64, because the random number of changed input terminals averages to 50%. The MedCache and MaxCache schemes compare values for executed nodes as well as for the input terminals, and these overheads will be larger as more nodes are executed. Since the number of executed nodes (proportional to the calculation cost) increases with the InputFraction, so does the overhead difference between MinCache and the other schemes. The appearance of a “maximum” in the overhead for the MedCache and MaxCache scheme is due to the graph construction algorithm used by the simulator: graphs with higher InputFraction will have relatively more nodes without children (output nodes), for which no comparison is required, and this lowers the overhead.

5.2.3.2 Overhead dependency on BranchFactor

![Graph showing overhead vs. BranchFactor]

Figure 65 Overhead vs. BranchFactor
The MinCache overhead is expected not to change with the BranchFactor (see Figure 65) since the number of input terminals remains unaffected by variation of the BranchFactor. MaxCache overhead increases with the BranchFactor for the same reason the calculation cost does; more executed nodes require more comparisons. The overhead for the MedCache scheme displays a maximum. This can be understood as follows: Figure 50 shows that the ThreadFraction reaches a maximum for a BranchFactor between 2 and 3 and decreases for higher values of the BranchFactor. 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 overhead is expected. At the same time, the number of executed nodes increases with the BranchFactor, so the maximum and decrease are not as pronounced as for the ThreadFraction relative to the BranchFactor (Figure 50).

5.2.3.3 Overhead dependency on ThreadFraction

The overhead for the MaxCache and MedCache schemes increases with the ThreadFraction about the same way as the calculation cost, for reasonable values of the BranchFactor. 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. Figure 67 offers an impression of this (there is no obvious reason why the MedCache overhead for high BranchFactor stays below the MaxCache overhead for low BranchFactor, as is the case for these simulations). Plotting MinCache overhead as a function of ThreadFraction does not make much sense, as it depends only on the number of input terminals. If the ThreadFraction variation is due to a change in the InputFraction, the MinCache overhead will increase with the ThreadFraction, but if the variation is due to a change in the BranchFactor, the MinCache overhead will be unaffected by it.

Figure 66 Overhead vs. ThreadFraction

The overhead for the MaxCache and MedCache schemes increases with the ThreadFraction about the same way as the calculation cost, for reasonable values of the BranchFactor. 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. Figure 67 offers an impression of this (there is no obvious reason why the MedCache overhead for high BranchFactor stays below the MaxCache overhead for low BranchFactor, as is the case for these simulations). Plotting MinCache overhead as a function of ThreadFraction does not make much sense, as it depends only on the number of input terminals. If the ThreadFraction variation is due to a change in the InputFraction, the MinCache overhead will increase with the ThreadFraction, but if the variation is due to a change in the BranchFactor, the MinCache overhead will be unaffected by it.
5.2.3.4 Overhead dependency on ChanceFactor

The effect of the ChanceFactor on the overhead is visualised in Figure 68. The MinCache overhead is independent of the ChanceFactor. The other overheads decrease with increasing 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. The MaxCache overhead is initially higher as it compares results for all executed nodes, not just those at the end of a thread. As the ChanceFactor increases, so does the effectiveness of comparing extra results, and for very high ChanceFactor values the MaxCache scheme may actually perform fewer comparisons than the MedCache scheme.

5.2.3.5 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. Under this assumption the MinCache overhead is
proportional to the changed fraction as in Figure 69. The average InputFraction in Figure 69 is 25%, which explains the maximum value for the MinCache overhead.

If, on the other hand, the system must always compare all input terminal values, the MinCache overhead would be a constant. The additional overhead in the MedCache and MaxCache schemes is proportional to the number of executed nodes, and so it will increase with increasing ChangedFraction. The rate of increase will depend on the graph properties as discussed previously.

5.2.3.6 Overhead dependency Conclusions

There is no single graph property that determines cache overhead for all caching schemes. The MinCache scheme overhead only depends on the InputFraction. The MaxCache overhead is proportional to the number of executed nodes in this scheme, and behaves in much the same way as the MaxCache calculation cost. The MedCache scheme overhead behaves differently for high BranchFactor values, but acts similar to the MaxCache overhead for reasonable values of the BranchFactor.
5.3 Demand driven scheme

5.3.1 Calculation cost

5.3.1.1 Cost dependency on InputFraction and OutputNumber

In the demand driven scheme, the NoCache cost is no longer constant for variations of InputFraction (Figure 71). The cost for each of the caching schemes increases with the InputFraction for low InputFractions as in the data driven scheme, but follows the decreasing trend of the NoCache cost for higher InputFractions. To understand why the NoCache cost decreases with increasing InputFraction, one must look at the fraction of the graph executed by the demand driven scheme (Figure 72).

![Figure 71 Demand driven Cost vs. InputFraction](image)

In the demand driven scheme, only the nodes leading to the requested output will be calculated. The calculation cost is proportional to the size of the fraction of the graph that leads to the requested output, because only that fraction is visited by the demand driven execution scheme. The size of the graph fraction leading to an output will be strongly influenced by the number of output terminals (the OutputNumber), as can be seen in Figure 73. For a graph with a single output, the demand driven scheme executes exactly the same operations as the data driven scheme. For higher OutputNumber values the visited fraction also decreases with the InputFraction.

Consider the case of 2 output nodes on Figure 72. One would assume each node to depend on about \( \frac{1}{2} \) of the graph, but this condition is only approached for the highest values of the InputFraction. This indicates that (especially for lower InputFractions) a considerable portion of the graph is shared in common by the output terminals.
Due to the graph construction method, the average OutputNumber increases with the InputFraction. Because the BranchFactor is constant, more input terminals will tend to cause a “wider” graph with more output terminals (see Figure 73). This explains why the decrease of the cost with increasing InputFraction in Figure 71 is more dramatic than what is suggested by Figure 72.

If the costs are corrected for the executed graph fraction by expressing them relative to the demand driven NoCache cost (Figure 74), the demand driven scheme behaves almost exactly like the data driven scheme in relation to the InputFraction. It seems the only difference between both execution schemes lies in the smaller executed fraction of the graph. Apart from this difference, conclusions reached in the section on data driven execution will also hold for demand driven execution.
Figure 74 Costs relative to NoCache cost (corrected for visited graph part)

Figure 75 shows how the demand driven costs strongly depend on the number of outputs.

Figure 75 Cost vs. OutputNumber

Because the graph is more or less “divided” among the output nodes, it is the absolute number of output terminals, and not the number of outputs relative to graph size, which determines the fraction of the graph that is visited during demand driven execution (see Figure 76).
5.3.1.2 Cost dependency on BranchFactor

In data driven execution, the costs increased with increasing BranchFactor. But a higher BranchFactor also increases the number of output terminals of a graph (Figure 77), and an increased number of outputs reduces the executed graph fraction, and hence the cost.

For very high BranchFactor values, the number of outputs would equal the number of nodes as the BranchFactor approaches the maximum value. Figure 77 shows the number of output nodes relative to the BranchFactor for a number of graphs with the same size and InputFraction (graphs have 10 input terminals and 30 other nodes).

The NoCache cost is no longer constant as in the data driven case, but decreases with increasing BranchFactor (due to the increasing number of outputs). For the other caching schemes the opposing influences of increasing ThreadFraction and increasing OutputNumber seem to cancel each other, and the costs seem almost independent of the BranchFactor (Figure 78). For high values of the BranchFactor, there will be only input
terminals and output nodes in the graph, and the average calculation cost would equal the
cost to recalculate the average node. This cost will differ for each graph.

![Figure 78 Cost vs. BranchFactor](image_url)

**Figure 78 Cost vs. BranchFactor**

### 5.3.1.3 Cost dependency on ThreadFraction

There is no clear relation between OutputNumber and ThreadFraction, even though
higher OutputNumbers and higher ThreadFraction values will both coincide with higher
InputFraction values. The NoCache cost is not constant, but decreases with increasing
ThreadFraction. The costs for the other caching schemes approach the NoCache cost for
higher ThreadFraction values (Figure 79). A graph with a single input terminal for
example will have a very low ThreadFraction value. Such a graph consists of one thread,
so the MinCache and MedCache scheme will both result in complete recalculation for
every input change.

![Figure 79 Cost vs. ThreadFraction](image_url)

**Figure 79 Cost vs. ThreadFraction**
For high BranchFactor values, the ThreadFraction will start to decrease with increasing BranchFactor, while the costs will approach the average output node calculation cost. Figure 80 shows an impression of this behaviour.

5.3.1.4 Cost dependency on ChanceFactor

The influence of the ChanceFactor is the same as in the data driven case, with the exception that now all the costs have lower average values (the reduction is proportional to the average executed graph fraction).

5.3.1.5 Cost dependency on ChangedFraction

The effect of the ChangedFraction too is similar to the data driven case, but with reduced average cost values.
Note that unlike in the data driven scheme, a ChangedFraction of zero does not correspond to a zero calculation cost. In the demand driven execution scheme, the nodes not leading to the requested result will not be recalculated, hence they are not consistent with the values on the input terminals. When another result is requested, these inconsistent nodes will have to be recalculated, even though no change was made to the input terminals. This explains the presence of a calculation cost for a zero ChangedFraction. This cost is about 10% of the total calculation cost on average (see Figure 82), and is lower for low InputFractions, probably because then the outputs have a larger part of the graph in common.
5.3.1.6 Cost dependency conclusions

The relative efficiency of the different caching schemes is similar to what was observed for the data driven scheme. The NoCache cost, however, is no longer constant and equal to the total execution cost, but proportional to the fraction of the graph executed to obtain a desired output. This fraction decreases with increasing OutputNumber, InputFraction and BranchFactor. The decrease in the costs for increasing InputFraction, BranchFactor or ThreadFraction is caused by the corresponding increase in OutputNumber.

5.3.2 Caching Overhead

5.3.2.1 Overhead dependency on InputFraction and OutputNumber

The MinCache overhead increases with increasing InputFraction, as in the data driven case, as more input terminals increase the number of values to compare. For higher InputFraction values, however, the relation between InputFraction and OutputNumber will result in comparisons for a smaller fraction of the input terminals. As a consequence, the MinCache overhead reaches a maximum and decreases with increasing InputFraction (Figure 84).
The MedCache and MaxCache overhead also have maxima caused by the opposing effects of increasing InputFraction. A higher InputFraction will increase the number of executed nodes in the graph, but will also reduce the graph fraction visited by demand driven execution (as was explained for Figure 71).

Overhead decreases with increasing OutputNumber, just like the calculation cost does (Figure 85). A higher OutputNumber reduces the number of nodes visited by the demand driven scheme, and as such reduces the number of comparisons.
5.3.2.2 Overhead dependency on BranchFactor

![Figure 86 Overhead vs. BranchFactor](image)

The BranchFactor too has opposing effects on the overhead (Figure 86). An increase in BranchFactor will cause more nodes to be executed, but also increases the OutputNumber. The MaxCache overhead increases with increasing BranchFactor. For high BranchFactor values, caching is ineffective, the decrease in the MedCache overhead is due to the decreasing number of threads, and hence lower number of nodes that cache their results in the MedCache scheme. The MinCache overhead is independent of the BranchFactor.

5.3.2.3 Overhead dependency on ThreadFraction

![Figure 87 Overhead vs. ThreadFraction](image)

The MedCache and MaxCache overhead increase, reach a maximum and decrease again with increasing ThreadFraction. The MinCache overhead is not directly influenced by the ThreadFraction, and is therefore not included in Figure 87.
5.3.2.4 Overhead dependency on ChanceFactor

The effect of the ChanceFactor on the overhead (Figure 88) is very much similar to its effect in the data driven case, except that the average values are a bit lower. The MedCache overhead remains lower than the MaxCache overhead.

5.3.2.5 Overhead dependency on ChangedFraction

The overhead is proportional with the ChangedFraction (Figure 89) as in the data driven scheme, except that a zero ChangedFraction results in a non-zero overhead for the MedCache and MaxCache scheme.
5.3.2.6 Conclusion

Figure 90 Effect of node and graph properties on cache overhead

The demand driven overhead, like the demand driven calculation cost, depends strongly on the graph fraction visited by the demand driven execution. The size of the graph fraction an output depends on is affected the most by the OutputNumber, and also by the InputFraction, BranchFactor and ThreadFraction. All caching schemes are similarly affected, demand driven execution does not seem to give an advantage to any specific scheme.
5.4 The effects of overhead importance

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 and 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 91, the overhead/cost proportion is varied linearly. As can be seen, no scheme is always superior to all others, and general conclusions are not straightforward. It seems that the MedCache scheme is never cheaper than the MaxCache scheme for demand driven execution, while for data driven execution it is only cheaper when the MinCache scheme outperforms both.

The overhead will most likely be lower than the calculation cost, depending on the complexity of the operations and the data size. For a matrix of size N, the comparison overhead will be proportional to N, whereas the cost of an operation like a matrix multiplication for example will be proportional to $N^2$. Assuming a floating point operation and comparison to be of equal complexity, for a matrix of size 1,000 for example, the calculation cost for a multiplication will be $10^3$ times the overhead for element-by-element comparison. The average proportion between caching overhead and calculation cost depends on the type of application that is implemented in the environment.

The curve for caching without the use of dataflow information uses the minimum overhead required for this scheme (as the exact value depends on the execution history which is unknown), and the same cost as the MaxCache scheme is assumed.

![Figure 91 Total Cost (calculation + overhead) for increasing relative comparison cost.](image)

This result is only valid on the simulated population of graphs. By considering a subset of low ChanceFactor, high InputFraction (Figure 92) or high ChanceFactor, low InputFraction (Figure 93), very different results are obtained.
In Figure 92, 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 when the overhead rises above 30% of the cost in the demand driven scheme, or above 70% in the data driven scheme.

\[
\text{Figure 92 Total cost (low ChanceFactor, high InputFraction)}
\]

In Figure 93, caching is always effective, and there is no question that the MaxCache scheme is superior, and that it is highly effective even when the cost of caching without using dataflow information exceeds the total calculation cost without caching. On the other hand, the demand driven calculation scheme is barely more effective than the data driven scheme in Figure 93, whereas in Figure 92, the difference between data driven and demand driven execution is quite dramatic.

\[
\text{Figure 93 Total cost (high ChanceFactor, low InputFraction)}
\]

There is no single caching scheme that is consistently superior, when both cost and overhead are considered. To determine which caching scheme is best in a given situation, both node properties (ChanceFactor), structural graph properties (ThreadFraction, InputFraction, BranchFactor), interaction properties (ChangedFraction) and overhead importance must be taken into consideration.
5.5 Areas of caching scheme effectiveness

In this section, regions in the parameter space are identified, where a given method is superior to all others.

In the following figures, the elevation in the 3D mesh represents the number of graphs for which a given caching scheme produces the lowest total cost, relative to the total number of samples in the local area. The relative importance of the cache overhead is set to 1% of the calculation cost (total cost = calculation cost + 0.01*cache overhead). The parameter space is divided in equal size, square areas by a 50X50 grid.

To improve clarity in the following figures, a surface was obtained by mapping the data to a regular grid. It must be noted that the very high InputFractions, where the graph contains far more input terminals than nodes, 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, those values are obtained through extrapolation by the grid algorithm. Some figures showed strange extrapolation effects on the borders, for these figures the surface is truncated to the area for which simulation data is available.

Note that the average ChangedFraction is 0.5 (50%) in the following figures, since all input states are considered equally likely.

The following sections will show areas of method effectiveness for:

- Overhead importance 1%:
  - ChanceFactor / InputFraction
  - ChanceFactor / ThreadFraction

- Overhead importance 50%:
  - ChanceFactor / InputFraction
  - ChanceFactor / ThreadFraction

5.5.1 Areas of method effectiveness on ChanceFactor/InputFraction

Graph

Figure 94 Sample density
Figure 94 shows the sample density in the ChanceFactor/InputFraction space (elevation corresponds to number of samples). The total number of samples is $4 \times 10^4$. The lower density areas mostly represent unlikely configurations (high InputFractions, high ChanceFactors) that received less attention during simulation, whereas the higher density areas correspond to more representative configurations.

Figure 95 No Cache

Figure 95 shows the graph parameters for which not caching is the most economical solution. The elevation in the 3D mesh represents the number of graphs for which no caching produces the lowest total cost, relative to the total number of samples in the local area.

The figure shows clearly that caching will not be appropriate for graphs with a high InputFraction. A small fraction of graphs with very low ChanceFactor does not benefit from caching either.

A ChanceFactor of 1 means every node will always produce the same result, regardless of its arguments. Only a node with a changed input terminal as argument is executed, so the more input terminals, the more executions. This explains why even for this ChanceFactor, the NoCache scheme is the most effective for a high enough InputFraction.

Figure 96 MinCache
Figure 96 shows the zone of best performance for the MinCache scheme (extrapolation omitted). For a high InputFraction and a low ChanceFactor, the difference in overhead between the MinCache scheme and the other caching schemes is the greatest. For a higher ChanceFactor, the other caching schemes become more effective, whereas the performance of the MinCache scheme does not depend on the ChanceFactor. It seems that even in its peak areas this scheme performs better than the others for less than 25% of the graphs in the area.

![Figure 96 MinCache](image)

**Figure 96 MinCache**

Figure 97 displays where cache comparison at the end of a thread is most effective. This scheme seems to be effective for a minority of graphs only. For a high InputFraction, the MedCache overhead will become important, while the MaxCache scheme will be more effective for a high ChanceFactor.

![Figure 97 MedCache](image)

**Figure 97 MedCache**

Figure 98 displays where caching all intermediate results and comparing values after each executed operation is the preferred method (extrapolation omitted). This appears to be the case 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.

![Figure 98 MaxCache](image)

**Figure 98 MaxCache**

104
5.5.2 Areas of method effectiveness on ChanceFactor/ThreadFraction Graph

The ThreadFraction increases with increasing InputFraction, but is a more general parameter as it can also describe variations in BranchFactor. However, the ThreadFraction figures are slightly harder to interpret. Depending on the BranchFactor, one ThreadFraction value corresponds to different performance values. Extrapolation is omitted in all ChanceFactor/ThreadFraction figures.

![Figure 99 Sample density](image)

The sample density (Figure 99) is spread more evenly for the ThreadFraction. Unlike the InputFraction, the ThreadFraction value of a graph is not set directly by the user, so the variety of values occurring is greater.

![Figure 100 NoCache](image)

The NoCache scheme is the most effective only for low ChanceFactor and high ThreadFraction. Even then no more than 50% of the graphs are most effective with this scheme (the vertical axis on the 3D plot shows the local fraction of graphs for which the scheme is most effective).
The effectiveness of the MinCache scheme does not directly depend on the ThreadFraction (only insofar as the ThreadFraction is influenced by the number of input terminals), and is independent of the ChanceFactor. For a high ChanceFactor, it will be outperformed by schemes that increase in performance with an increasing ChanceFactor.

Comparing more node results to a cache shifts the area of maximum effectiveness towards the lower ThreadFraction, higher ChanceFactor region.
The MaxCache scheme seems to be the only one that is superior for a majority of the graphs in a large part of the ChanceFactor-ThreadFraction space.

5.5.3 Areas of method effectiveness on ChanceFactor/InputFraction Graph for high overhead cost

In the following figures, the importance of the cache overhead is taken to be 50% of the calculation cost. The samples distribution is the same as in the previous sections. It can be expected that an increase in overhead will make caching less attractive.

As expected, a larger section of the InputFraction/ChanceFactor space does not benefit a lot from caching.

The MinCache scheme gains somewhat in importance, as it becomes the most efficient scheme for more than 60% of the graphs in certain areas. For a high InputFraction, the costs are almost identical for the three caching schemes, but the MinCache overhead is lower than the overhead for the MedCache and MaxCache schemes. For a lower InputFraction, the difference between the costs increases.
The medium caching scheme gains some importance too. When caching overhead was less expensive, it was effective for only an insignificant fraction of the graphs, here it is the most effective scheme for some 10% of the graphs in a fairly large area.

The Maximum caching scheme still dominates the low InputFraction/high ChanceFactor region (Figure 107).

### 5.5.4 Areas of method effectiveness on ChanceFactor/ThreadFraction Graph for high overhead cost

In the following figures, the importance of the cache overhead is taken to be 50% of the calculation cost. The samples distribution is the same as in the previous sections.
Due to the higher overhead, caching is less attractive for a larger part of the parameter space.

The MinCache scheme gains considerably in importance, being most effective for a majority of the graphs in some areas.
The MedCache scheme is most effective on the border between the MinCache and MaxCache areas of effectiveness. It seems that for the assumed importance of the caching overhead, the MedCache scheme is never more effective for the majority of the simulated graphs. Note that there are other factors, such as the required cache space, which were not taken into account to determine these areas of method effectiveness.

![Image showing the effectiveness of MedCache](image.png)

**Figure 111 MaxCache**

The MaxCache scheme still dominates the high ChanceFactor, low ThreadFraction corner.

The ThreadFraction figures naturally show a larger area of effectiveness for caching than the InputFraction figures, as can be understood by checking Figure 54, which shows the relation between InputFraction and ThreadFraction.
5.6 Cost variation due to undetermined factors

This section will briefly consider the impact on the caching performance of aspects of the graph topology that are not covered by the generic graph parameters, and determine the uncertainty on the averages in the simulation results. A cost histogram can be used as a measure for the appropriateness of the chosen graph structural parameters. A histogram shows the number of times each cost value occurred, and can show how representative the average cost values are.

A cost histogram for the various data driven schemes is represented in Figure 112. The cost average and standard deviation are also given for each caching scheme (the distributions are not quite Gaussian though).

![Figure 112 Data driven Cost Histogram](image)

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Avg (%)</th>
<th>Std (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoCache</td>
<td>95</td>
<td>8</td>
</tr>
<tr>
<td>MinCache</td>
<td>64</td>
<td>31</td>
</tr>
<tr>
<td>MaxCache</td>
<td>49</td>
<td>31</td>
</tr>
<tr>
<td>MedCache</td>
<td>56</td>
<td>32</td>
</tr>
</tbody>
</table>

*Table 1 Cost averages and standard deviations*

![Figure 113 Cost Averages and Standard Deviation](image)

The Cost histogram (Figure 112) shows that the cost can vary considerably for a given caching scheme. The average and standard deviation (Figure 113) are used here only as an indication, since the distributions are evidently non-Gaussian.
The large variation in cost values is mostly the result of different fractions of changed inputs, with low ChangedFractions explaining the occurrences of low costs for any caching scheme (see Figure 114). A high ChangedFraction will cause the highest cost, which is almost equal to the NoCache cost (100%) for the MinCache scheme. The other schemes avoid redundant operations even when all inputs changed, hence their lower maximum costs. Compare this with the effect of changing the InputFraction on the maximum cost as seen on Figure 115.

**Figure 114 MaxCache Cost Histogram for various ChangedFractions**

**Figure 115 MinCache Cost Histogram for various InputFractions**

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>Cost Avg</th>
<th>Cost Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoCache</td>
<td>96%</td>
<td>5%</td>
</tr>
<tr>
<td>MinCache</td>
<td>71%</td>
<td>12%</td>
</tr>
<tr>
<td>MaxCache</td>
<td>50%</td>
<td>15%</td>
</tr>
<tr>
<td>MedCache</td>
<td>60%</td>
<td>16%</td>
</tr>
</tbody>
</table>

*Table 2 Cost averages and standard deviations for fixed parameters*
Figure 116 gives the standard deviations for a fixed ChangedFraction (50%) and fixed values for all enforceable graph parameters. The remaining variation, caused by the random and unrecorded variations in graph structure, is much smaller. Note how in Figure 116, the variation is lowest on the MinCache scheme. The other two caching schemes have an additional random factor: intermediate nodes may produce the same results for changed inputs. This explains why, in Figure 114, the cost histogram shows a considerable amount of low cost values even for 100% changed inputs.

5.6.1 Conclusion

When the interaction parameters are kept constant, the variation on the cost values is relatively small (standard deviation around 15%) for fixed values of the generalised graph parameters (InputFraction, BranchFactor, etc.). This suggests that, at least for the simulations, the chosen graph parameters are fairly representative on average of the actual graph structural properties.

Note that the standard deviations plotted in Figure 116 do not represent the uncertainty on the averages. According to the Central Limit theorem [55], the variance on the averages is \( \frac{\sigma^2}{n} \) with \( n \) the number of individuals in the average.
6. Performance evaluation for individual graphs

This chapter discusses caching performance evaluation for a specific, individual graph. Methods are presented to calculate cost and overhead for the various caching schemes, given a specific graph. The results for two example graphs are compared to the generalised performance evaluation based on the simulations.

The two sample algorithms demonstrate that a real graph’s performance can deviate considerably from the average graph structure generated by the simulator. The behaviour of the cost and overhead as a function of the ChanceFactor can differ considerably for almost identical graphs. Even so, the performance calculated for two sample algorithms approximately matches the areas of method effectiveness derived from the simulations.

This chapter also presents a discussion of caching overhead importance relative to calculation cost, and whether caching and the use of dataflow information are profitable in this regard.
6.1 First algorithm

A simple network analyser is used as sample algorithm (Figure 117).

The network analyser will calculate the transfer function or power spectrum of a device under test. This example immediately illustrates the difficulty of comparing data driven and demand driven execution. The use of two output terminals is only sensible in a demand driven environment. For data driven execution, the programmer will most likely insert a condition immediately following the FFT nodes to select one of the outputs.

![Figure 117 Simple network analyser](image)

This 18 node graph has 5 input terminals (InputFraction 28%) and 13 other nodes, two of which are output nodes (connected to an output terminal). There are 9 threads, which makes the ThreadFraction 50% and the average ThreadLength 2. If we break up this total, we find 7 1-node threads (input terminals 1 to 5, and nodes B and D in Figure 118), 1 2-node thread (nodes A and C) and 1 9-node thread (nodes E-M). Due to the simplicity of this algorithm and the absence of display or data processing settings, there is a distinct disproportion of ThreadLength before and after the measurement. The BranchFactor of the graph is 1.28. The cost $s$ for set-up of the measurement equipment, $G_n$ for loading the signal into the generator and $M_n$ for the measurement will depend on such factors as the hardware and the sample period.

115
6.1.1 Determining the Cost

The calculation costs and cache sizes are indicated on Figure 117.

If the input terminals and other nodes are numbered as in Figure 118, the following table can be constructed:

<table>
<thead>
<tr>
<th>Inputs changed</th>
<th>data driven</th>
<th>Nodes executed</th>
<th>demand driven O1</th>
<th>demand driven O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EFGHIJKLM</td>
<td>EFGHIKLML</td>
<td>EFGHJ</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ACDEFGHIJKLM</td>
<td>ACDEFGHIKLML</td>
<td>ACDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BDEFGHIJKLM</td>
<td>BDEFGHIKLML</td>
<td>BDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DEFHGIJKLM</td>
<td>DEFHGIKLML</td>
<td>DEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;2</td>
<td>ACDEFGHIJKLM</td>
<td>ACDEFGHIKLML</td>
<td>ACDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;3</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;4</td>
<td>BDEFGHIJKLM</td>
<td>BDEFGHIKLML</td>
<td>BDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;5</td>
<td>DEFHGIJKLM</td>
<td>DEFHGIKLML</td>
<td>DEFGHJ</td>
<td></td>
</tr>
<tr>
<td>2;3</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>2;4</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>2;5</td>
<td>ACDEFGHIJKLM</td>
<td>ACDEFGHIKLML</td>
<td>ACDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>3;4</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>3;5</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>4;5</td>
<td>BDEFGHIJKLM</td>
<td>BDEFGHIKLML</td>
<td>BDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;2;3</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;2;4</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;2;5</td>
<td>ACDEFGHIJKLM</td>
<td>ACDEFGHIKLML</td>
<td>ACDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;3;4</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
<tr>
<td>1;3;5</td>
<td>ABCDEFGHIJKLM</td>
<td>ABCDEFGHIKLML</td>
<td>ABCDEFGHJ</td>
<td></td>
</tr>
</tbody>
</table>
For all 32 possible combinations of input status, 20 lead to complete recalculation. The total data driven cost (using the node costs of Figure 117) is $2n^2 + Gn + 3n \log_2 n + Mn + s + 5n$, with $n$ the number of samples. One combination leads to a data driven cost of $2n^2 + 2n \log_2 n + 5n$, two to a cost of $2n^2 + 2n \log_2 n + 5n + Mn$, four to a cost of $2n^2 + 2n \log_2 n + 5n + Mn + s$ and four to a cost of $2n^2 + Gn + 3n \log_2 n + Mn + 5n$. One, with no inputs changed, has cost 0.

With demand driven execution and assuming both outputs are equally likely to be requested, the total cost is $2n^2 + Gn + 3n \log_2 n + Mn + s + 2.5n$, with $n$ the number of samples. One input state leads to a cost of $2n^2 + 2n \log_2 n + 2.5n$, two to a cost of $2n^2 + 2n \log_2 n + 2.5n + Mn$, four to a cost of $2n^2 + 2n \log_2 n + 2.5n + Mn + s$ and four to a cost of $2n^2 + Gn + 3n \log_2 n + Mn + 2.5n$.

Note that in this graph, 2/3 of the possible input states require a complete recalculation of the graph.

<table>
<thead>
<tr>
<th>Input states (total 32)</th>
<th>data driven</th>
<th>demand driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>$2n^2 + Gn + 3n \log_2 n + Mn + s + 5n$</td>
<td>$2n^2 + Gn + 3n \log_2 n + Mn + s + 2.5n$</td>
</tr>
<tr>
<td>1</td>
<td>$2n^2 + 2n \log_2 n + 5n$</td>
<td>$2n^2 + 2n \log_2 n + 2.5n$</td>
</tr>
<tr>
<td>2</td>
<td>$2n^2 + 2n \log_2 n + 5n + Mn$</td>
<td>$2n^2 + 2n \log_2 n + 2.5n + Mn$</td>
</tr>
<tr>
<td>4</td>
<td>$2n^2 + 2n \log_2 n + 5n + Mn + s$</td>
<td>$2n^2 + 2n \log_2 n + 2.5n + Mn + s$</td>
</tr>
<tr>
<td>4</td>
<td>$2n^2 + Gn + 3n \log_2 n + Mn + 5n$</td>
<td>$2n^2 + Gn + 3n \log_2 n + Mn + 2.5n$</td>
</tr>
</tbody>
</table>

Assuming all input states are equally likely, the average required calculation cost is $1.9375n^2 + 0.75Gn + 2.6875n \log_2 n + 0.9375Mn + 0.75s + 4.8437n$ data driven, and $1.9375n^2 + 0.75Gn + 2.6875n \log_2 n + 0.9375Mn + 0.75s + 2.4218n$ demand driven.

The difference between data driven and demand driven execution is marginal in this case.

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, and the calculation cost is identical for each caching scheme.
6.1.2 Alternative cost determination

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

This is alternatively proved as follows: 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 possibility that none of the \( p \) inputs changes is \( \sum_{k=0}^{p} \binom{i-p}{k} = 2^{i-p} \). 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 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 (the cost for an input terminal \( C_{\text{input}} \) is zero): \( C_{\text{total}} = \sum_n P_n \cdot C_n \).

For the demand driven scheme the node cost is also multiplied with the fraction of output terminals located on a path from the node. For the example, this gives:

\[
\frac{31}{32} \left( 2n^2 + 2n \log_2 n + 5n \right) + \frac{15}{16} M_n + \frac{3}{4} (n \log_2 n + s + G_n) \quad \text{data driven and}
\]

\[
\frac{31}{32} \left( 2n^2 + 2n \log_2 n + \frac{1}{2} n + \frac{4}{2} n \right) + \frac{15}{16} M_n + \frac{3}{4} (n \log_2 n + s + G_n) \quad \text{demand driven}
\]

which is exactly the cost obtained earlier.

This is a simple (and correct, assuming all input states are equally likely) method to determine the MinCache calculation cost, which allows to quickly guess whether caching is likely to produce a significant reduction in calculation cost for a given graph. The calculation costs for the other caching methods will always be equal or lower than the MinCache cost.

6.1.3 Caching Overhead

The comparison overhead is proportional to the total data size of the input terminals, which is 5 in the minimum caching scheme. For the maximum caching scheme, considering all input changes equally likely (and a ChanceFactor of zero), the overhead can be determined in the same way as the calculation cost. The MaxCache overhead is \( 5.5937 + 12.0937n \) in the data driven case and \( 5.5937 + 9.5937n \) in the demand driven case (the comparison overhead is taken to be proportional to the size of the compared values).

When intermediate results are cached without using dataflow information, with a cache table associated with each function, the comparison overhead (maximum when all input values are unchanged) can be determined, assuming a lookup overhead proportional to the number of entries entered in the cache since the cached value was last used. The comparison overhead (equal to the minimum cache size without using dataflow information) is \( 9 + 27n \), more than twice the total data size (or the maximum cache size when dataflow information is used), which is \( 6 + 13n \). The difference in overhead is caused
by multiple nodes having the same operation. As a result, multiple results are added to this
operation’s cache table during a run, and the previous arguments of a given node will not
always be found on top of the cache.

The minimum cache size (required by the MinCache and MedCache scheme) is $6+5n$, or about 38% of the cache space required by the MaxCache scheme (the total data size). According to the simulations, the minimum cache space needed is proportional to the ThreadFraction (50%) on average. For this actual problem however, more than half of the values to be cached are scalar and only a few are arrays, which explains the difference between the relative number of cached nodes and the relative size of the required cache space. In the generalised results of the simulations, the data size is evenly distributed in the average graph.

6.1.4 Numerical example

For a numerical example, assume $n = 10^5$, $M = 10$, $G = 0.5$ and $s = 10^4$. This yields the following costs:

<table>
<thead>
<tr>
<th></th>
<th>demand driven</th>
<th>data driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoCache cost</td>
<td>20.006.292.892</td>
<td>20.006.542.892</td>
</tr>
<tr>
<td>Cached cost</td>
<td>19.380.688.520</td>
<td>19.380.930.710</td>
</tr>
</tbody>
</table>

Table 5 Numerical example

Note that the execution costs of the most expensive and least expensive nodes differ by a factor of $10^6$. The chosen value of $n$ may appear rather high. This high value is selected to underline the potentially poor performance of the caching scheme for this graph, but it is not totally unrealistic: a practical example would be a measurement on a wide band system such as a synchronous machine [56]. The cost and overhead for smaller values of $n$ is displayed in Figure 121. Figure 120 shows a graphical comparison of the overhead for various caching schemes.

Figure 119 Overview of average costs

Figure 120 Comparison Overhead

119
The obtained expressions can be used to visualise the effect of various properties on the cost and overhead.

**Figure 121 Relative MaxCache cost and overhead vs. n**

The cost with caching and the MaxCache overhead vary very little with the data size $n$, except for “low” ($<1000$) values of $n$ (Figure 121).

**Figure 122 Cached cost vs. $M$, $G$ and $s$**

The influence of the other variables on the cost seems relatively small. The most calculation-intensive nodes are the windowing nodes, which are also executed for every change to the input status. This partly explains the small influence of caching and of the execution scheme on the performance. It also explains the decrease in the relative cost with an increase in the relative importance of nodes that can be skipped because of the caching scheme. In Figure 122, increasing the variables $M$, $G$ and $s$ increases the relative execution cost of nodes that are sometimes skipped. In Figure 121 too, low values for $n$ will decrease the dominating influence of the windowing nodes on the execution cost.

On the areas of effectiveness, this graph lies on the border of the area where caching is effective. For an important caching overhead, the NoCache scheme is more effective, but for a less important overhead, caching is more effective.

### 6.1.5 Introducing the ChanceFactor

As stated before, it is very unlikely that any of the operations in this algorithm will produce unchanged results for changed arguments. However, the calculation cost for each node can be expressed mathematically as a function of the ChanceFactor. This allows the calculation of the MaxCache cost as a function of the ChanceFactor.

To this end, a probability $r_n$ of producing the same value for changed input arguments is associated with each node $n$. The average of this probability on graph level is the ChanceFactor of the graph.
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.

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 \( i \), \( r_i = 0 \), and \( P_i = \frac{1}{2} \).

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:

\[
P_n = 1 - \prod_{\text{arguments}(n)} (P_{\text{argument}} \cdot r_{\text{argument}} + (1 - P_{\text{argument}}))
\]

The calculation cost for the graph is then \( C_{\text{total}} = \sum_n P_n \cdot C_n \) with \( C_n \) the cost of node \( n \).

For simplicity, the probabilities \( r \) will be assumed identical for each node. As before, all input states are considered equally likely. The resulting calculation cost in the data driven MaxCache scheme, relative to the NoCache cost, is plotted as a function of the ChanceFactor (\( n=10^4 \), \( s=10^4 \), \( G=2 \), \( M=25 \)) in Figure 123.

![Figure 123 Data driven MaxCache cost vs. ChanceFactor](image)

Note that the cost decreases more or less linearly with the ChanceFactor. The cost does not reach zero for a unity ChanceFactor (all nodes always produce the same results), because the nodes with changed input terminals as arguments will still have to be recalculated: the environment does not “know” these values will always be identical. Because the nodes with input terminals as arguments are among the most computation-intensive in the example, the resulting cost is relatively high.

The MinCache cost is independent of the ChanceFactor, and equals the maximum MaxCache cost value. The MaxCache overhead can be calculated in a similar way, and is plotted in Figure 124. As in the simulations, the total data size (\( 13n+6 \)) is used as a reference.

121
The total cost (calculation cost + caching overhead) is plotted on Figure 125, for an overhead importance of 0.01 and 0.5. For an overhead importance of 0.01 (1%), the MaxCache total cost is always lower than the NoCache and MinCache cost, but for an overhead importance of 0.5 (50%), the MinCache cost is lower for ChanceFactor values under 0.4 (because all input values are scalar, the MinCache overhead is 5, completely negligible compared to the calculation cost). This transition point corresponds well to the areas of method effectiveness for InputFraction 26%, ThreadFraction 47% (see section 6.1.6).

The cost and overhead for the MedCache scheme can be calculated as well. The probability for a node $n$ to be executed is $P_n = 1 - \prod_{\text{arguments}(n)} (P_{\text{argument}} \cdot r_{\text{argument}} + (1 - P_{\text{argument}}))$ as in the MaxCache scheme, but since only nodes at the end of a thread check their result, the term $P_{\text{argument}} \cdot r_{\text{argument}}$ should be omitted for all other nodes. The only nodes at the end of threads in the example are B, C and D.
For this algorithm, the MedCache cost (Figure 126) is only slightly higher than the MaxCache cost. The overhead, however, is substantially lower (Figure 127) and less sensitive to ChanceFactor variations. This is understandable, because there is only one node in the graph (node D) that will sometimes be skipped in the MedCache scheme.
Comparing all schemes for this problem (Figure 129), the MaxCache scheme is most effective for most ChanceFactor values, when the overhead is small (1%) compared to the calculation cost. For a more important overhead (50%), the MinCache scheme is most effective for a ChanceFactor below 0.2, and the MedCache scheme is the most effective one for all other ChanceFactor values. The superiority of the MedCache scheme over the MaxCache scheme was not predicted by the simulations. It could be due to the asymmetric distribution of input terminals in the graph.

![Figure 129 Comparison of caching schemes (0.01 & 0.5 overhead)](image)

The most cost-expensive nodes are the windowing nodes (E and F), which are also executed for every change to the input status (at least in the MinCache scheme). This partly explains the small influence of caching and of the execution scheme on the performance.

The average ChangedFraction is 50% in the above calculations because all input states are considered equally likely. The cost for the various input states can also be plotted relative to the ChangedFraction (Figure 130).

![Figure 130 MaxCache cost vs. ChangedFraction, ChanceFactor 0](image)

For a ChanceFactor of zero, the calculation cost saved by caching in the MaxCache scheme seems to be negligible. The majority of input conditions (all except unchanged input values) require complete recalculation and only some low-cost operations can be avoided by caching in this algorithm (Figure 130).
For a higher ChanceFactor, the behaviour of the average MaxCache cost relative to the ChangedFraction matches the simulated results better, though it is still higher than the simulated cost (Figure 131). The line represents average values, the dots are the actual cost values. Note that only a limited number of different cost values occur.

6.1.6 Conclusion

This algorithm is small and asymmetric with a very uneven cost distribution. It would be unrealistic to expect a perfect match between the performance for this graph and the results of the simulations.

Figure 132 shows the most effective caching scheme for this algorithm, plotted on the InputFraction-ChanceFactor figures, for low (left) and high (right) overhead importance. Note that the NoCache scheme is never superior in the examples, probably because the MinCache overhead is inconsequential. The effective area for the MaxCache scheme is smaller, and the MedCache scheme is more important than predicted by the simulations.
6.2 A second algorithm

In the previous example, the windowing nodes account for most of the total calculation cost. Without the windowing, the graph becomes as in Figure 133. This graph has 15 nodes, 4 of which are inputs, and an InputFraction of 27%. The graph has 7 threads, or a ThreadFraction of 46.6%. Note that these parameters are hardly different from the previous graph.

6.2.1 Determining the Cost

The costs can be determined as in 6.1.1, the derivation is omitted here. The total cost now becomes \( G_n + 3n \log_2 n + M_n + s + 5n \) for the data driven scheme and \( G_n + 3n \log_2 n + M_n + s + 2.5n \) for the demand driven one. Of the 16 possible input states, 10 require total recalculation. One has a cost of \( 2n \log_2 n + M_n + 5n \) data driven and \( 2n \log_2 n + M_n + 2.5n \) demand driven, two have a cost of \( 2n \log_2 n + M_n + s + 5n \) data driven and \( 2n \log_2 n + M_n + s + 2.5n \) demand driven, and two a cost of \( G_n + 3n \log_2 n + M_n + 5n \) data driven and \( G_n + 3n \log_2 n + M_n + 2.5n \) demand driven.

<table>
<thead>
<tr>
<th>input states (total 16)</th>
<th>data driven</th>
<th>demand driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>( G_n + 3n \log_2 n + M_n + s + 5n )</td>
<td>( G_n + 3n \log_2 n + M_n + s + 2.5n )</td>
</tr>
</tbody>
</table>
Table 6 Cost summary

The average required calculation cost is:

\[ 0.75Gn+2.625n\log_2 n+0.937Mn+0.75s+4.687n \text{ data driven and} \]

\[ 0.75Gn+2.625n\log_2 n+0.937Mn+0.75s+2.344n \text{ demand driven.} \]

The total data size is 5+11n, and the minimum required cache size (as used by the MinCache and MedCache scheme) is 5+3n, or 27% of the total data size for large values of n (again smaller than predicted by the simulations).

### 6.2.2 Numerical example

Assuming as before \( n = 10^5 \), \( M = 10 \), \( G = 0.5 \) and \( s = 10^4 \), The following costs are obtained:

<table>
<thead>
<tr>
<th></th>
<th>demand driven</th>
<th>data driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoCache cost</td>
<td>6.292.892</td>
<td>6.542.892</td>
</tr>
<tr>
<td>Cached cost</td>
<td>5.576.430</td>
<td>5.810.730</td>
</tr>
</tbody>
</table>

Table 7 Numerical example

The algorithm is even more measurement-centred, with almost no settings affecting the data processing or representation. As a result, a change to even one of the settings will always lead to the execution of the larger part of the graph. Again, 2/3 of the possible input states result in complete recalculation. Yet, due to the more even distribution of the calculation costs, the performance difference between the various schemes is slightly more pronounced (see Figure 134).

![Figure 134 Average costs for the other analyser](image-url)
6.2.3 Introducing the ChanceFactor

Again, the calculation cost can be determined as a function of the ChanceFactor.

The probability that node \( n \) produces the same value for changed input arguments is called \( r_n \). The probability \( P_n \) that the node \( n \) will be executed in the MaxCache scheme (when it would be executed in the MinCache scheme) is

\[
P_n = 1 - \prod_{\text{arguments}(n)} (P_{\text{arg \, unused}} \cdot r_{\text{arg \, unused}} + (1 - P_{\text{arg \, unused}})).
\]

Using the numbering scheme from Figure 135, the following probabilities are obtained (this calculation was omitted for the previous example graph due to its greater length, but it is completely analogous):

\[
\begin{align*}
P_a &= 1 - (1 - \frac{1}{2})^2 \cdot (1 - \frac{1}{2}) = \frac{3}{4}; \quad (P_{\text{input}} = \frac{1}{2}, r_{\text{input}} = 0) \\
P_b &= \frac{3}{4}; \\
P_c &= 1 - r_a; \\
P_d &= 1 - ((1 - P_b + r_b \cdot P_b) \cdot (1 - P_c + r_c \cdot P_c))^2; \\
P_e &= 1 - (1 - P_d + r_d \cdot P_d); \\
P_f &= P_e; \\
P_g &= 1 - (1 - P_c + r_c \cdot P_c); \\
P_h &= 1 - ((1 - P_e + r_e \cdot P_e) \cdot (1 - P_f + r_f \cdot P_f)); \\
P_i &= 1 - ((1 - P_g + r_g \cdot P_g) \cdot (1 - P_h + r_h \cdot P_h)); \\
P_j &= 1 - ((1 - P_f + r_f \cdot P_f) \cdot (1 - P_g + r_g \cdot P_g)); \\
P_k &= 1 - ((1 - P_i + r_i \cdot P_i) \cdot (1 - P_j + r_j \cdot P_j));
\end{align*}
\]

If \( C_n \) is the calculation cost of node \( n \), and \( C_{\text{input}} \) is zero, then the total cost can be obtained as:

\[
C_{\text{total}} = \sum_n P_n \cdot C_n
\]

The resulting calculation cost in the data driven MaxCache scheme, relative to the NoCache cost, is plotted as a function of the ChanceFactor \( (n=10^4, s=10^4, G=2, M=25) \) in Figure 136.
The MaxCache cost is lower, and the overhead (Figure 137) is slightly larger for low ChanceFactor values, but lower for high values (compared to Figure 124).

The total cost (calculation cost + caching overhead) as plotted on Figure 138, is lower than for the previous example graph. The difference is more pronounced for high ChanceFactor values. For an overhead importance of 1%, the MaxCache total cost is always below the NoCache and MinCache cost, but for an overhead importance of 50%, the MinCache cost is lower for ChanceFactor values under 0.35. This transition point seems to correspond slightly better to the areas of method effectiveness (InputFraction 27%, ThreadFraction 47%).
The MedCache cost behaves not at all like the MaxCache cost (Figure 139).

For the MedCache scheme the only nodes at the end of threads are B and C. Since these nodes are never skipped, the overhead is not influenced by the ChanceFactor (see Figure 140).
Note that even when overhead importance is increased up to 50% of the calculation cost, the total MedCache cost (calculation cost+overhead) is always lower than the NoCache Cost, and lower than the MinCache cost for ChanceFactor values over 0.25 (Figure 141).

![Figure 141 MedCache cost+overhead](image)

Comparing all schemes for this problem (Figure 142), the MaxCache scheme is the most effective for all ChanceFactor values, when the overhead is small (1%) compared to the calculation cost. For a more important overhead (50%), the MinCache scheme is most effective for a ChanceFactor below 0.25, and the MedCache scheme is the most effective one between 0.25 and 0.45, and for a ChanceFactor over 0.97. The MaxCache scheme is most effective for a ChanceFactor between 0.45 and 0.97.

![Figure 142 Comparison of caching schemes](image)

The MaxCache cost relative to the ChangedFraction is displayed in Figure 143. There are only 4 inputs, so the number of possible ChangedFraction values is limited to 5. As in the first algorithm, the average cost is higher than the simulated cost in graphs with similar parameters (but it is a small graph).
Note that while the average cost increases, the calculation cost for individual input states is not particularly affected by the ChangedFraction, almost all ChangedFraction values can result in complete recalculation. This may be due to the relatively small number of input terminals in these examples and their asymmetric distribution over the graph.

Figure 144 shows the MaxCache cost relative to the ChangedFraction for a ChanceFactor of 0.5. The maximum cost (for ChangedFraction 100%) is below 70%, which fits the simulations better. For the first example, a ChanceFactor of 0.5 still resulted in a maximum cost of 100%.

6.2.4 Conclusion

This graph is even smaller and less symmetric than the previous example, but the cost is more evenly distributed. As a result, the caching schemes perform better. A comparison of the caching performance to the simulated values shows that the most effective scheme as a function of the ChanceFactor corresponds surprisingly well to the areas of effectiveness as determined by the simulations.
Figure 145 Most effective schemes, plotted on the InputFraction-ChanceFactor figures

Figure 145 shows the most effective schemes for this algorithm, as a function of the ChanceFactor, for low (left) and high (right) overhead importance. The calculated performance for this algorithm matches the simulated areas of effectiveness surprisingly well.

Two small, nearly identical examples such as these are hardly sufficient to justify general conclusions, yet it seems that while an uneven distribution of input terminals will reduce caching performance, the simulation results concerning the best caching schemes remain valid. On the other hand, a very uneven distribution of calculation cost causes a more serious deviation from the predicted performance.
6.3 Importance of caching overhead relative to calculation cost

In the previous figures, the overhead was expressed as a fraction of the total data size of the graph, and added to the calculation cost after multiplying it with a more or less arbitrary weight factor. Since this was how the overhead was handled in the simulations, the results obtained here could be compared to the simulation results. However, in this case the relation between the number of samples n and the calculation cost is known. The importance of the overhead (following the simulator’s assumption that it is proportional to the data size) can be determined by comparing the total data size with the total calculation cost (Figure 146).

Figure 146 Overhead importance, relative to calculation cost

Figure 146 is a plot of the total data size, which is 6+13n for graph 1 and 5+11n for graph 2, relative to the total calculation cost which is 2n^2+Gn+3nlog_2n+Mn+s+5n for graph 1, and Gn+3nlog_2n+Mn+s+5n for graph 2. The figure shows how the relative importance of the overhead depends on the data size n (=the number of samples), and also on other program parameters such as M, G and s (these are given arbitrary values in Figure 146).

Depending on the values of M, G and s, there is a substantial difference in overhead importance between the two algorithms. The figure suggests that the comparison overhead caused by caching is negligible for small data sizes (which could be expected), but also (for algorithms with complex calculations at least) for very large data sizes.

Figure 146 also shows that the (somewhat arbitrary chosen) values of 1% and 50% that were used to plot the areas of method effectiveness are not implausible.
As Figure 147 demonstrates, the caching overhead quickly becomes insignificant next 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. It should be kept in mind, however, that computer speeds increase faster than memory speeds, and that in some cases, overhead could include retrieving a cached value from secondary storage.

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 not necessary.
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 148 (note that the scale and the relative proportions of the areas are drawn arbitrarily).
Figure 148 Areas in data size - complexity plane
6.4 Summary

The calculation cost and the caching overhead can be calculated by relatively simple formulae as a function of the ChanceFactor for a given program graph. The most effective caching methods for these two examples are not notably different from the ones suggested by the simulations, as represented by the areas of method effectiveness, though the results differ from the generalised performance the more unequally distributed the calculation cost and data size are.

The simulation results suggested a more or less linear influence of the ChanceFactor on the cost and overhead, but Figure 129 and Figure 142 indicate that the behaviour may vary significantly from one graph to the next. The MedCache cost, for example, behaves very differently in Figure 129 and Figure 142. The overhead importance also differs considerably (Figure 146), even though the graphs are really very similar in their graph properties (though not in the average complexity of their calculations).

For an actual algorithm, the cost as a function of the ChangedFraction is far from resembling the smooth curve of the averaged simulations, as the cost will increment in discrete steps.

The MinCache and MedCache schemes offer a reduction in required cache space and overhead greater than predicted by the simulator, due to the number of scalar results to be cached. As noted before, the simulator did not take into account the data size distribution, such as that input terminals are more likely to be scalar than other nodes.

The calculation cost reduction is lower than predicted by the simulations in both these examples. The reason for this is that a majority (2/3) of the input states results in complete recalculation of the graph. This might be caused by the asymmetrical distribution of input terminals in both algorithms. Also, in the first algorithm, the most calculation-intensive nodes are always executed, and the nodes that can be skipped due to caching do not make much difference to the overall calculation cost in comparison.

These examples demonstrate that even extremely simple algorithms such as these require some effort to analyse manually, which underscores the usefulness of simulations to derive generalised conclusions. The performance evaluation also requires assumptions about the probability of all the possible input conditions, and advance knowledge of the hardware on which the algorithm will be run. If some operations may return the same results for a changed set of arguments, the probability for these occurrences must be estimated as well.

A method designed to quickly determine the MinCache calculation cost was presented, which offers an estimation whether caching is likely to produce significant savings for a given graph. This method consists of summing the costs of all nodes in the graph, with each node’s cost multiplied by 1-2^p, where p is the number of input terminals on which the node depends. This can be quickly determined for each node. The calculation costs for other caching methods will always be equal to or lower than the MinCache cost.

The presented calculation method can be adapted to dataflow with extensions. For an iteration subgraph, the subgraph’s input terminal probabilities should be adjusted reflecting their probability of change. The result for the subgraph can be multiplied by the expected average number of iterations. For conditional graph parts, the conditional nodes’
costs are multiplied by the probability that the graph part executes. These values may be
difficult to determine, but this is not a disadvantage exclusive to the calculation method, as
it is equally true for simulations, for example. The simulator could be adapted to work on a
specific algorithm graph including dataflow extensions, but such a simulation would also
require the range in which the number of iterations is expected to fall, the probability of
each outcome of a condition, etc. For a given algorithm with a known graph, it is instead
preferable to calculate the performance using the methods presented in this chapter.
7. Implementation assessment

This chapter presents an overview of a prototype implementation, and discusses some of the practical difficulties encountered when implementing caching in an environment with dataflow extensions, or which arise when converting a text program to a dataflow graph.

The prototype shows that a more complex execution scheme can cause unexpected behaviour and results in complications for the programmer. This can cancel the possible benefits gained by the added complexity.
7.1 Role of the prototype

A prototype was developed to study the details of cache implementation in a dataflow environment. The experience with the prototype eventually led to some of the conclusions in the previous sections. Some of the findings were also incorporated in the simulator.

The prototype was originally conceived as a demand driven dataflow environment using the MinCache scheme. This was later expanded to include the MaxCache scheme. Other features were added as the corresponding concepts were developed.

The demand driven scheme is more complex than the data driven execution scheme. Whether a node’s arguments changed or not is insufficient to determine the validity of the node in the demand driven scheme, because it is possible that the node has not yet used the previous argument values. Caching in this scheme requires time tags on the data to determine whether the result of a node is consistent with the current set of input terminals. It was chosen specifically to study the implementation details of these aspects.

The prototype contains a lexical analyser and parser that accepts a simplified MatLab-like syntax. It takes a text program description as input and converts it into an executable dataflow graph. The program description can be mixed with interaction statements, such as assigning values to input terminals and requesting the values of output results. In addition to assigning values to input terminals and requesting the values of output results, the user can save the workspace, print the graph status and view a log of all executed operations.

The prototype communicates with MatLab, which acts as a server to perform calculations, store and display data and interact with the VXI rack that was used to perform actual measurements.

![Diagram of the prototype schematic](image)

*Figure 149 prototype schematic*

The prototype suffered from several poor design choices. Its practical usefulness was limited by the slowness of the communication with MatLab (which was based on Apple
events). The C code generated by the lexical analyser and parser available at the time did not integrate easily into the C++ code used for the rest of the program.

A more important limitation of the prototype is that even a complete redesign would still not make it suitable to evaluate the performance of the implemented caching schemes, due to the lack of example algorithms.

Because of the several problems and the limited analysis possible using the prototype, it was not developed any further. Instead the prototype was replaced by the simulator, drawing on the lessons learned.
7.2 Language features

The prototype’s language supports expressions built from standard arithmetic and relational operators, as well as a number of array operations and MatLab built-in functions. User-defined m-files can be added to the library of recognised functions. Functions can have multiple return values (nodes can have more than one output).

7.2.1 Program Control

Supported control structures include virtual data dependency, conditional assignment and iteration by while loops. Loops can be nested. A simple scope manager keeps track of variable names across loop boundaries.

7.2.1.1 Virtual dependency

The prototype supports virtual dependency, by allowing additional arguments to be added to a function. These arguments must be valid before the function will execute, even though their value is not used in calculating the function.

To force a graph to execute in this scheme (such as when a change occurred to something not represented in the graph, for example a change in the device under test), it is necessary to introduce virtual input terminals. Reassigning the same value to an ordinary input can not be used to force a recalculation, as the caching scheme will recognise that the value did not really change.

Virtual input terminals are connected as arguments to those nodes whose execution may have to be forced. As the value of a virtual input is not actually used, a different value can be assigned each time a recalculation is necessary.

![Figure 150 virtual data dependency](image)

An action like SaveToFile can be operated by a virtual output terminal. The user can then request the action in the same way he asks for a calculation result. The action is only executed if some of its input data has changed.

142
A possible problem arises here. Suppose some action ("print a hardcopy" for example) must be performed each time it is requested by the user. The user would then have to change a virtual input terminal before requesting the action through the virtual output. Alternatively a dummy input could be introduced, that always reports a changed value. Neither of these solutions appear to promote clear and understandable programming.

Another question is whether a virtual node argument (whether changed or not) should or should not trigger recalculation of the node if all the real arguments are unchanged. Such questions can only be answered on a case by case basis. In the prototype, a changed virtual argument is treated like any other argument. This might lead to some redundant operations, but this is preferable to undue skimming over operations.

### 7.2.1.2 Conditional execution

Conditional execution is provided by a merge node (a conditional assignment in the program text). Since the execution is demand driven, a switch node is not needed: selecting a branch can be handled by selecting the appropriate output terminal.

The execution scheme evaluates the condition first, to determine which of the two other arguments will be needed. As a result, the conditional node requires special treatment by the execution scheme.

### 7.2.1.3 Iteration

A while structure is implemented in the prototype to study iteration. The while structure consists of a subgraph, which is managed by a controlling 'while'-node. All values used in the subgraph pass through the while-node. It behaves just like any other

---

143
node on the outside; unlike the conditional node, a while-node does not require special treatment by the execution scheme.

The iteration uses loop variables, which are allocated transparently. Each loop variable has a corresponding input and output terminal in the subgraph. The value assigned to the input terminal to start the iteration is called the pre-loop value. This is usually the value of a node not in the subgraph. The value assigned to the variable after each iteration step is the subgraph output terminal associated with the loop variable, and is called the post-loop value. The implementation details of the iteration structure are offered in section 7.3.5.

7.2.1.4 Procedural abstraction

Because the prototype uses static dataflow execution, procedural abstraction on the graph level was not implemented completely, as re-entrant procedures are not supported by static dataflow. For a subgraph to be a real procedure, it must be possible to invoke the subgraph on different locations. This can be achieved in various ways. The subgraph can be copied on these locations, or the subgraph can generate separate data spaces for each of its instances. In the latter case, the execution scheme must be able to distinguish and handle multiple data tokens from the different instances on the arcs of the subgraph. If this happens dynamically at runtime, the procedure is re-entrant. This was not implemented in the prototype. As a result, recursion is not possible in the prototype. The scope manager could be used to implement “inline” procedures, where part of the algorithm could be isolated into a subgraph, but the usefulness of inline procedures is rather limited. Changing the scope is only really used for processing iterations.

The node containing the subgraph will likely have multiple output values. In the demand driven execution scheme, it is possible that not all output values are requested. The enclosing node could in turn only demand the requested values from the subgraph. This requires that each output value of a node has its own time tags to determine its validity, rather than node tags that determine the validity of the node. This was omitted in the prototype, so a demand of one output value results in complete recalculation of the node.

7.2.2 Changing array elements

An assignment to an array element raises some questions. Since a node is supposed to produce the results of its operation as its output, the output of the assignment node should be the changed array. But caching a complete copy of the array in the cache of every node that changes a single array element seems wasteful. Unfortunately though, unless the element to be changed is fixed and known in advance, it is practically impossible to track dependencies for individual array elements.
Figure 153 array assignment

The prototype supports array indexing (left and right hand side). The entire array is cached for every node that changes an array element.
7.3 Implementation of the prototype

7.3.1 Data structure

The graph structure is administered by a number of tables. The NodeTable maintains the nodes and their data, the Scope Manager maintains a table of the identifiers in the different scopes, and other tables contain the list of update methods of the nodes, a list of constants and other information. These tables constitute the foundation of the prototype. The most important data structures are the Scope Manager and the NodeTable.

The Scope Manager contains the symbol tables, constant tables and methods tables for the various scopes, which are needed in the graph construction process. Scope changes are invoked to handle iteration loop bodies. The Scope manager collects the lists of subgraph input terminals and output nodes.

The NodeTable contains all the nodes that constitute the graph. Each node maintains a table of argument indices, a table of output data (the cache), an update method, a list of the input terminals the node depends on, a time tag, a time of last change tag and a constant flag. A while node additionally contains a table of input terminals for the subgraph with indices to their preloop and postloop nodes, a table of indices to the output nodes of the subgraph, and a reference to the loop termination condition node. This multidimensional list structure was ill-suited to extension beyond the original concepts, and the simulator is based instead on a structure modelled on a relational database, which allows more flexibility in the access to graph information.

7.3.2 Executing the dataflow graph

When an output value is requested by the user, the output passes the request on to its arguments. Each argument in turn passes on the demand, which propagates through the graph until it reaches only input terminals and constants. If these have a value available, it will be passed downstream to the requesting nodes. Otherwise the user is prompted to provide an input value.

In the prototype implementation, all nodes are provided with a cache that stores the last value of the node. Because a node must be able to determine whether its cached value is still valid, that is, whether it is consistent with the current values on the input terminals, a clock is incremented each time an input terminal is changed. Every node or terminal contains the time value of its last change. A node has a valid result in its cache if its arguments are valid and none are newer than the cached value.

The following pseudocode describes the execution scheme. Handling of constants and input terminals is omitted for brevity.

```plaintext
checkDataConsistency
for (each Parent in Node.ParentList)
    Parent.checkDataConsistency;
if (Node.Time < Parent.Time)
    Node.Time = Parent.Time;
```
Node.Valid = false;
if (Node.Valid)
    no recalculation needed;
else
    Node.Recalculate;
Node.Valid = true;

7.3.3 Input Terminal sets

Every time an output is requested, a consistency check is propagated through the graph. This can be avoided since the consistency of a node is ultimately determined by a (small) set of input terminals. If each node is provided with a list of the input terminals it depends on, checking the consistency is reduced to comparing the time tag of the node with the tags of the terminals in the list. The check will then propagate only to nodes that are no longer valid. If the graph is very large, or connection time is important (like in a distributed system) this could result in important savings (propagation overhead was not considered by the simulator, but the reduction in propagation overhead would be roughly proportional to the reduction in calculation cost achieved by the MinCache scheme).

If the graph contains a conditional node (a merge node), part of the calculation becomes conditional. Hence, the dependencies of a conditional node (and all other nodes depending on this one) become dependent on the runtime value of the condition. It takes only a small additional effort to recompile the input terminal sets of the nodes during the recalculation phase. Each node takes as its input terminal set the union of the input terminal sets of its parents. An input terminal places only itself in its input terminal set. A conditional node includes parents depending on the state of its condition (the condition is always a parent).

The recalculation scheme then becomes:

checkDataConsistency
if (Node.Time < Node.InputTerminalSet.Time) //Node is not up to date
    if(Node is Conditional)
        Condition.checkDataConsistency;
        if(Condition is true)
            TrueParent.checkDataConsistency;
        else
            FalseParent.checkDataConsistency;
    else
        for (each Parent in Node.ParentList)
            Parent.checkDataConsistency;
        Node.Recalculate;
        Node.RecompileInputTerminalSet;
        Node.Time = Node.InputTerminalSet. Time;
else
    no recalculation needed;

The set of input terminals is compiled thus:

RecompileInputTerminalSet
if(Node is Input Terminal)
    InputTerminalSet.add(Node);
else
    if(Node is Conditional and Condition.Time >0)
if(Condition is true)
    InputTerminalSet = union(Condition.InputTerminalSet,
    True.InputTerminalSet);
else
    InputTerminalSet = union(Condition.InputTerminalSet,
    False.InputTerminalSet);
else
    for(each Parent in Node.ParentList)
        InputTerminalSet = union(InputTerminalSet,
        Parent.InputTerminalSet);

The set of a conditional node remains valid as long as the condition does not change. When the condition does change, the node has to be recalculated, and the input terminal set is updated at the same time.

In the case of virtual dependency links, whether the input terminal set of a virtual argument should be included in the input terminal set of a node must be determined case by case. The safest solution is to include the virtual dependencies by default, and this is the case implemented in the prototype.

### 7.3.4 Introducing the MaxCache scheme

One point that became obvious during the prototype operation is that ‘new’ values may still lead to redundant operations. A relational operator (for instance as condition input for a conditional node) has only two possible output values for any argument value, and can easily lead to a large part of the graph recalculating exactly the same values as before. The prototype was adapted so that a node compares a newly calculated value with the one previously held in its cache. In addition to the time tag, a node features a “time since last change” tag. When a node is no longer up to date, it will check whether its newer arguments have changed value since its last update. If not, it will only update its time tag, but will not execute its recalculation method. If the arguments changed, it will recalculate and compare the new result with the value in its cache. When the recalculated result is really a new value, the node will update its “time since last change” tag. This way downstream nodes will no longer be executed when a node produces the same value for new argument values.

The following pseudocode describes the execution scheme. Note that the time tag is determined in the same way as before.

```plaintext
checkDataConsistency()
    update time of input terminal set for this node;
    if (Node.Time < Node.InputTerminalSet.Time)
        if (Node’s operation is Conditional)
            Condition->checkDataConsistency();
            if (Node.Time < Condition.TimeOfLastChange)
                changed = true;
            if (Condition is true)
                True->checkDataConsistency();
                if (Node.Time < True.TimeOfLastChange)
                    changed = true;
            else
                False->checkConsistency();
        else
            ... (remaining pseudocode)
```
if (Node.Time < False.TimeOfLastChange)
    changed = true;
else
    for (all arguments of Node)
        argument->checkDataConsistency();
        if (Node.Time < argument.TimeOfLastChange)
            changed = true;

if (changed)
    recalculate();
    if (new value == cached value)
        changed=false;
    recompileInputTerminalSet();
else
    no recalculation needed;

Node.Time = Node.InputTerminalSet.Time;
if(changed)
    Node.TimeOfLastChange=Node.Time;

7.3.5 Implementing iteration

The while structure consists of a subgraph, which is manipulated by the controlling ‘while’-node. The while-node controls the flow of all data from and to the subgraph. The control node passes the values of its parents to the input terminals of the subgraph and requests a calculation result from some specified outputs in the subgraph. The control node also manages the caching of all iteration results. Externally, it behaves just like any ordinary node.

The while-node will initialise the input terminals of the subgraph with the values of its parents and then evaluate some condition-node. Until the condition becomes false, the node will repeatedly:

1. request calculation results from certain nodes
2. assign the obtained results to the corresponding input terminals

When the iteration terminates, the while-node requests all output values from the subgraph. Those which were not yet used in the iteration are now calculated, while the other output values are already available.

The while-node usually has multiple outputs, some of which might not be requested in the demand driven execution. Due to the nature of the iteration mechanism, the executed part of the subgraph is determined by which values are needed for the next iteration step, and is independent of the number of requested output values.

The “while” structure is constructed as follows: when the while statement is encountered in the program text, a new scope is declared. The statements are then processed in the usual fashion (as described in section 7.4).

An input terminal is created for each variable used in the loop. Variables that get something assigned to them in the loop body (left hand side variables) must be initialised
before the iteration, in case the number of iterations is zero. This initial value is called the pre-loop value of the input terminal. Before the first iteration step, the pre-loop value is assigned to the input terminal. The post-loop value is the value assigned to the input terminal for subsequent iteration steps, and is usually the output of a node in the subgraph.

A variable that is not assigned a value in the loop body (only occurs in the right hand side of assignment statements) is still an argument to the while node, and requires an input terminal to the subgraph, but the value of this input terminal remains constant during subsequent iteration steps. For such an input terminal, the pre-loop and post-loop values are identical.

When the end of the loop body is encountered in the program text, the control node is created to manage the iteration. The left-hand-side variables in the scope become the output values of the control node. The loop scope is terminated, and in the enclosing scope the identifiers connected to left-hand-side variables in the loop are associated with the loop node’s outputs.

The loop termination condition and its input terminals are kept separated from the other nodes. This is because the condition must be evaluated before the postloop values are assigned to the loop body’s input terminals. Otherwise the iteration would perform a single step too much.

Example iteration:

```java
m=0
while(m<10)
    m=m+1
end
```

An input terminal is created for the m used in the condition. The constant 10 and the comparison node are generated, and the condition is incorporated into the newly created while-node. A new input terminal is created for the loop variable m used in the body of the loop. The addition node becomes the postloop value for m. It also becomes an output of the while-node. Upon reaching the end statement, the while node is finalised. Besides its arguments (the pre-loop value for m), this node knows the input terminals of the subgraph, their preloop and postloop nodes, and the loop termination condition. The loop body scope is then terminated, and the enclosing scope is adjusted so the identifier m now refers to the output of the loop node.
When the while node must recalculate, it will assign the preloop value for m to the condition input and evaluate the condition. An assignment to an input terminal increments the input’s time tag (otherwise nothing would happen after the first iteration step). If the condition is true, the preloop value is used as output m. If not, the preloop value is assigned to the loop body input. The while node then requests the postloop value of m, which causes the loop body to execute. The postloop value is then assigned to the condition input, and the condition is evaluated. If the condition is true, the while node requests all outputs from the subgraph. In this case the output is consistent with the subgraphs inputs, and no more calculations are needed. The while node then finishes, passing the new value for m to its downstream nodes.

Pseudocode implementation for while-loop:

```plaintext
for (each Element in InTermList)
    // InTermList is the list of Input terminals of the subgraph
    Element.Data = Element.PreLoop.Data;
    // PreLoop refers to the node providing the initial value of Element
    Element.Time = Element.PreLoop.Time;
for (each Element in ConditionList)
    // ConditionList is the Input terminal list of Condition
    Element.Data = Element.PreLoop.Data;
    Element.Time = GetTime; // Adjust the time tag to force evaluation
    do
        for (each Element in InTermList)
            Element.PostLoop.checkDataConsistency;
        // PostLoop is the node providing Element’s value for the next iteration
        for (each Element in ConditionList)
            if (Element.PostLoop!=Element) // variable not right hand side only
                Element.Data = Element.PostLoop.Data;
                Element.Time = GetTime;
        LoopEndCondition.checkDataConsistency;
        if (LoopEndCondition is true)
            continue;
        else
            for (each Element in InTermList)
                if (Element.PostLoop!=Element)
                    Element.Data = Element.PostLoop.Data;
                    Element.Time = GetTime;
```

Figure 154 iteration structure
while(LoopEndCondition)
for(each Element in ResultList) // loop-invariant results
   Element.checkDataConsistency;

In this implementation, loop-invariant computations are evaluated only once. This happens without specific effort to detect or isolate them, because only the data that is needed to assign to the subgraph’s input terminals is calculated in each iteration step. The other results are only requested after the iteration terminates.

When the while-node assigns values to the input terminals of the subgraph, the timer must be incremented, and these terminals must get an updated time tag. If those values retained their post-loop time tags, the execution scheme would not notice the update, and the subgraph will never execute. The same is true for the arguments to the loop termination condition. The arguments to the condition do not necessarily change each time the loop must be executed, a change to any of the other arguments requires execution of the iteration too. In order to force the condition to evaluate, the time tags on the terminals in the condition’s input terminal set must be incremented before an execution of the loop.
7.4 Converting a text program into a dataflow graph

The text description of the algorithm is converted into an executable dataflow graph. Statements in the program text are converted one by one. The conversion from text to dataflow graph will be illustrated by a few examples. Suppose the tables containing the dataflow graph are as represented in Figure 155 (hypothetical situation).

For a statement like A=ln(B)+C, first the expression ln(B)+C is processed as follows. The identifiers B and C are looked up in the symbol table (table of identifiers and pointers to the nodes the identifiers refer to) of the current scope (Figure 156). This produces the nodes n2 and n5 which represent the current values of B and C. These could be input terminals, or they could be interior nodes of the graph if B and/or C had been the left hand argument of previous statements. If an identifier is not found in the symbol table, a new input terminal is created for it.

Then the method table is checked for the existence of a node with ln(n2) as update method. The method of a node consists of the operation name, and the indices of the argument nodes. In this case such a node exists (n4), otherwise a new node would be

Figure 155 Graph tables

Figure 156 Find identifiers B and C
created with \( \ln(n_2) \) as update method. This automatically avoids duplication of common sub-expressions (Figure 157).

![Figure 157 Finding a common sub-expression](image)

The method table is now checked for the existence of a node with \( n_4 + n_5 \) as update method. In this case, a new node, \( n_6 \) is created with \( n_4 + n_5 \) as update method. Then the pair \( \langle A, n_6 \rangle \) is added to the symbol table (Figure 158).

![Figure 158 Adding a new node](image)

If other associations for identifier A existed previously, they are removed from the symbol table (Figure 159). Each node keeps a count for the number of references (direct or indirect) by which it can be reached. Nodes whose count reaches zero become ‘unreachable’ by the program and can be removed.
For an assignment such as \( E=A \), the current node corresponding to the symbol \( A \) is retrieved from the symbol table, and an entry is added for this node with the symbol \( E \) (Figure 160). Multiple identifiers may refer to the same node, but the same identifier can not refer to multiple nodes in one scope.

By building the graph this way, repeated expressions are not needlessly duplicated in the graph structure. It can be demonstrated [21] that a smaller representation is not possible without changing the algorithm.

A list of constants is also maintained to prevent the duplication of constant values. Nodes with only constant arguments are constants too, and will be calculated only once.

### 7.4.1 Example of a conversion

In the prototype a text program is read and the corresponding dependency structure built. The user can interactively add commands to the program, change the value of the input terminals or request the value of an output terminal. In the latter case, the returned value will always conform to the latest value of the input terminals. Calculations will only be executed when necessary. If a new instruction is added, the graph is automatically checked for the existence of this instruction, thus avoiding common sub-expressions.
The following network analyser program uses a while loop for averaging. The sample program features a control input terminal, \texttt{dut}, by which the user must inform the system when the Device Under Test is changed or replaced by a new one.

\begin{verbatim}
fClk = 1e7
divider := 7
m = 2^divider
fsAcq = fClk/m
dut := 1
n = 1024
signal = rand(n,1) .* 2 .-1
averages :=5
m=1
%initialisation of loop variables omitted for brevity
while(m<=averages)
    [X,Y]=measure(n;generate(signal,fsACQ;m);setup(fsACQ,n);dut)
    X=fft(X)
    Y=fft(Y)
    Gxy= ((m-1)*Gxy+(conj(X).*Y))/m
    Gxx= ((m-1)*Gxx+(conj(X).*X))/m
    Gyy= ((m-1)*Gyy+(conj(Y).*Y))/m
    m=m+1
end

tff = Gxy./Gxx
coh=(conj(Gxy).*Gxy)./(Gxx.*Gyy)
impuls=ifft(tff)
corr1=ifft(Gxx)
corr2=ifft(Gyy)
cross=ifft(Gxy)
\end{verbatim}

The program consists of the main graph, a loop body subgraph and a condition subgraph. Note that the while node has an output for each variable occurring on the left hand side in the loop body.
Figure 161 The main graph with WHILE node
Figure 162 Loop body subgraph

Figure 163 The condition subgraph
7.5 Summary of prototype observations

7.5.1 Demand driven execution

At first sight, the MinCache scheme using input terminal sets seemed ideal for demand driven execution. However, it was soon observed that the MinCache scheme is not sufficient to eliminate all redundant operations. Additionally, demand driven execution is more complex than data driven execution, and in combination with caching it poses some questions which are hard to answer, especially concerning virtual dependency and dummy input terminals. Augmenting demand driven execution with input terminal sets would reduce propagation overhead of a graph by an amount proportional to the simulated calculation cost reduction in the demand driven MinCache scheme. However, propagation overhead was never an issue in the prototype. Input terminal sets make handling nodes with side-effects more difficult, and the demand propagation overhead is unlikely to be sufficiently important in normal (non-distributed) applications to justify the increase in complexity.

7.5.2 Caching coupled with dataflow extensions

7.5.2.1 Iteration

In the prototype, the results of an iteration are cached by the iteration control node, so if none of the iteration’s arguments (values used in the iteration) changed, the results can be returned without referring to the subgraph.

If the nodes in the loop body subgraph cache their last value only as in the prototype, the caching scheme avoids repeating operations with identical arguments between subsequent iteration steps. Caching between non-consecutive iteration steps would require each node in the loop body to cache the results for all iteration steps.

In dynamic dataflow execution, caching would not benefit much from the dynamic part of the dataflow information. If a part of the dataflow graph is created and destroyed dynamically during execution, the data dependency information about the cached values becomes unavailable for subsequent executions. Even if it remains available, the information may well be useless, as the dataflow graph will have changed. The prototype was limited to static dataflow, where the dataflow graph is fixed and this problem does not occur.

7.5.2.2 Virtual dependencies

Virtual dependency relations create extra arguments to a node. The value of these arguments is irrelevant, since their presence only serves to delay the execution of the node until the virtual argument nodes have completed their operations. When the environment uses caching, it is somewhat unclear whether a virtual argument should result in the
execution of the node, when the “real” arguments are unchanged. Executing the operation may be safe but redundant, while failure to execute might lead to program inconsistency.

7.5.3 Caching overhead and performance

The program performing the calculations and comparisons (MatLab) performs element-by-element comparison to ascertain the equality of two arrays, so for the prototype the comparison overhead is proportional to the data size. One could imagine more efficient ways to compare large arrays (for example using a checksum method, or at least decide at the first non-identical element encountered). A few benchmark tests showed that the CPU time for an array comparison is equal to other O(n) array operations in MatLab.

Real-time performance measurements on the prototype were meaningless, due to the highly fluctuating nature of the communication overhead for the data exchanges between the prototype application and MatLab. Also, no conclusions were possible without a representative set of algorithms on which performance could be measured. Such a representative set of algorithms was not available for implementation in the prototype’s language.

7.5.4 Programming interface

The prototype executes nodes during the construction of the graph, as soon as necessary data becomes available. That way the developer gets immediate feedback on his code. The implemented execution scheme can handle changes to the graph structure and compute the new result without redundant operations. However, the text interface only allows the addition of new nodes. The reason for this limitation is that the prototype contains no method to directly edit the graph structure through the text interface. The prototype does, however, support the automatic removal of graph parts that become unreachable (by assigning their identifiers to other nodes).

The design of the programming interface is an important consideration in the development of a programming environment. The examples of the previous section illustrate that a text description of an algorithm can be converted fairly easy to a dataflow graph as long as all operations are side-effect free. However, it is obviously awkward to specify virtual dependency links through text in a clear and elegant way. An environment where dependencies are explicitly entered, like a graphical programming environment or a spreadsheet-based programming interface, would make it much easier to specify control links, but lacks the ability to integrate existing programs. Also, complex networks tend to become obscure and difficult to edit in a graphical representation, and graphs with many control links are far less comprehensive than ordinary sequential programs with proper text layout. On the plus side, the proposed method has little trouble eliminating common sub-expressions while constructing the dataflow graph.
7.6 Detailed example of prototype operation

This section presents the prototype output generated for the example of section 7.4.1. The number of iterations is limited to one for brevity.

7.6.1 Input file

fCLK=1e7
divider:=7
m=2^divider
fsACQ=fCLK/m
fsGEN=fsACQ:
dut:=1

n=1024
signaal=rand(n,1).*2.-1

averages:=1
m=1
x=0
y=x
X=x
Y=x
Gxy=x
Gxx=x
Gyy=x

while(m<=averages)
m=m+1
[x,y]=measure(n,generate(signaal,fsGEN,m),setup(fsACQ,n,m),dut)
X=fft(x)
Y=fft(y)
Gxy=Gxy + (conj(X).*Y)
Gxx=Gxx + (conj(X).*X)
Gyy=Gyy + (conj(Y).*Y)
end

Gxy=Gxy/averages
Gxx=Gxx/averages
Gyy=Gyy/averages

tff=Gxy./Gxx
coh=(conj(Gxy).*Gxy)./(Gxx.*Gyy)
impuls=ifft(tff)
corr1=ifft(Gxx)
corr2=ifft(Gyy)
cross=ifft(Gxy)

Ax=[0:1:length(x)-1]*fsACQ/length(x)

plot(Ax,(20*log10(abs(tff))))
pause(0.1)
7.6.2 Prototype output

The output documents how the input file is converted into a dataflow program. While the environment is demand driven, during the graph construction process, new nodes are executed immediately if their arguments are already available. Afterwards, nodes are only executed on demand.

assign 1e+07 to node (0) at time 1
new constant 1e+07 is node (0)
nodetime of node (0) is 1
node (0) is constant.

new identifier divider is new node (1)
assign 7 to node (1) at time 2
assign 2 to node (2) at time 3
new constant 2 is node (2)
found that identifier divider is node (1)
new operation 1 / 1 2, (0), (3). new node (4)
new identifier m is node (3)
nodetime of node (3) is 0
node (3) is not up to date: forward..
nodetime of node (2) is 3
node (2) is constant.

nodetime of node (1) is 2
node (1) is input terminal: back.

recalculating (3).
execute : n2d0^n1d0;

found that identifier fCLK is node (0)
found that identifier m is node (3)
new operation 1 / 1 2, (0), (3). new node (4)
new identifier fsACQ is node (4)
nodetime of node (4) is 0
node (4) is not up to date: forward..
nodetime of node (0) is 1
node (0) is constant.

nodetime of node (3) is 2
node (3) : no recalculation needed
recalculating (4).
execute : n0d0/n3d0;

found that identifier fsACQ is node (4)
new identifier fsGEN is node (4)
nodetime of node (4) is 2
node (4) : no recalculation needed

new identifier dut is new node (5)
assign 1 to node (5) at time 4
assign 1024 to node (6) at time 5
new constant 1024 is node (6)
new identifier n is node (6)
nodetime of node (6) is 5
node (6) is constant.

found that identifier n is node (6)
assign 1 to node (7) at time 6
new constant 1 is node (7)
new operation 4 rand 1 2, (6), (7). new node (8)
found that constant 2 is node (2)
new operation 1 / 1 2, (8), (2). new node (9)
found that constant 1 is node (7)
new operation 1 - 1 2, (9), (7). new node (10)
new identifier signaal is node (10)
nodetime of node (10) is 0
nodetime of node (9) is 0
nodetime of node (8) is 0
nodetime of node (6) is 5
node (6) is constant.

nodetime of node (7) is 6
node (7) is constant.

execute : tmp0=rand(n6d0,n7d0);
recalculating (8).

nodetime of node (2) is 3
node (2) is constant.

execute : n8d0 .^n2d0;
recalculating (9).

nodetime of node (7) is 6
node (7) is constant.

execute : n9d0-n7d0;
recalculating (10).

new identifier averages is new node (11)
assign 1 to node (11) at time 7
found that constant 1 is node (7)
identifier m becomes node (7)
nodetime of node (7) is 6
node (7) is constant.

assign 0 to node (12) at time 8
new constant 0 is node (12)
new identifier x is node (12)
nodetime of node (12) is 8
node (12) is constant.

found that identifier x is node (12)
new identifier y is node (12)
nodetime of node (12) is 8
node (12) is constant.

found that identifier x is node (12)
assign n19d0 to (19)
assign n20d0 to (20)
assign n21d0 to (21)
assign n23d0 to (23)
assign n25d0 to (25)
assign n27d0 to (27)
assign n28d0 to (28)
assign n30d0 to (30)
assign n32d0 to (32)
assign n33d0 to (33)
assign n37d0 to (37)
assign n40d0 to (40)
node (15) : calculating upward
  execute : n13d0<=n14d0;
assign n7d0 (1) to node (13) at time 10
node (15) : calculating upward
  execute : n13d0<=n14d0;
assign n11d0 (1) to node (14) at time 11
  nodetime of node (15) is 11
  node (15) : no recalculation needed
  nodetime of node (18) is 0
  node (18) : no recalculation needed
  node (18) is not up to date: forward..
  nodetime of node (16) is 6
  node (16) is input terminal: back.
  nodetime of node (17) is 9
  node (17) is constant.
  recalculating (18).
  execute : n16d0+n17d0;
  nodetime of node (19) is 5
  node (19) is input terminal: back.
  nodetime of node (20) is 1
  node (20) is input terminal: back.
  nodetime of node (21) is 2
  node (21) is input terminal: back.
  nodetime of node (23) is 2
  node (23) is input terminal: back.
  nodetime of node (25) is 4
  node (25) is input terminal: back.
  nodetime of node (26) is 0
  node (26) is not up to date: forward..
  nodetime of node (19) is 5
  node (19) is input terminal: back.
  nodetime of node (22) is 0
  node (22) is not up to date: forward..
  nodetime of node (20) is 1
  node (20) is input terminal: back.
  nodetime of node (21) is 2
  node (21) is input terminal: back.
  nodetime of node (18) is 6
  node (18) : no recalculation needed
  recalculate (22).
  execute : generate(n20d0,n21d0);
  nodetime of node (24) is 0
  node (24) is not up to date: forward..
  nodetime of node (23) is 2
  node (23) is input terminal: back.
  nodetime of node (19) is 5
  node (19) is input terminal: back.
  nodetime of node (18) is 6
  node (18) : no recalculation needed
  recalculating (24).
  execute : setup(n23d0,n19d0);
  nodetime of node (25) is 4
  node (25) is input terminal: back.
  nodetime of node (26) is 0
  node (26) is not up to date: forward..
  nodetime of node (26) is 6
  node (26) : no recalculation needed
  recalculating (26).
  execute : [tmp0,tmp1]=measure(n19d0);
  nodetime of node (29) is 0
  node (29) is not up to date: forward..
  nodetime of node (26) is 6
  node (26) : no recalculation needed
  recalculating (29).
  execute : tmp0=fft(n26d0);
  nodetime of node (31) is 0
  node (31) is not up to date: forward..
  nodetime of node (26) is 6
  node (26) : no recalculation needed
  recalculating (31).
  execute : tmp0=fft(n26d1);
  nodetime of node (36) is 0
  node (36) is not up to date: forward..
  nodetime of node (33) is 8
  node (33) is input terminal: back.
  nodetime of node (35) is 0
  node (35) is not up to date: forward..
  nodetime of node (34) is 0
  node (34) is not up to date: forward..
  nodetime of node (29) is 6
  node (29) : no recalculation needed
  recalculating (34).
  execute : tmp0=conj(n29d0);
nodetime of node (31) is 6
node (31) : no recalculation needed

recalculating (35).
execute : n34d0 * n31d0;

recalculating (36).
execute : n33d0 + n35d0;

nodetime of node (39) is 0
node (39) is not up to date: forward..
nodetime of node (37) is 8
node (37) is input terminal: back.

nodetime of node (38) is 0
node (38) is not up to date: forward..
nodetime of node (34) is 6
node (34) : no recalculation needed

nodetime of node (29) is 6
node (29) : no recalculation needed

recalculating (38).
execute : n34d0 * n29d0;

recalculating (39).
execute : n37d0 + n38d0;

nodetime of node (43) is 0
node (43) is not up to date: forward..
nodetime of node (40) is 8
node (40) is input terminal: back.

nodetime of node (42) is 0
node (42) is not up to date: forward..
nodetime of node (41) is 0
node (41) is not up to date: forward..
nodetime of node (31) is 6
node (31) : no recalculation needed

recalculating (41).
execute : tmp0 = conj(n31d0);

nodetime of node (31) is 6
node (31) : no recalculation needed

recalculating (42).
execute : n41d0 * n31d0;

recalculating (43).
execute : n40d0 + n42d0;

node (15) : calculating upward
execute : n13d0 <= n14d0;
assign n18d0 (2) to node (13) at time 12

nodetime of node (15) is 12
node (15) : no recalculation needed

nodetime of node (18) is 6
node (18) : no recalculation needed

nodetime of node (26) is 6
node (26) : no recalculation needed

nodetime of node (26) is 6
node (26) : no recalculation needed

nodetime of node (29) is 6
node (29) : no recalculation needed

nodetime of node (31) is 6
node (31) : no recalculation needed

nodetime of node (36) is 8
node (36) : no recalculation needed

nodetime of node (39) is 8
node (39) : no recalculation needed

nodetime of node (43) is 8
node (43) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (44) is 4
node (44) : no recalculation needed

found that identifier Gxy is node (44, 5)
found that identifier averages is node (11)
new operation 1 / 1 2, (44, 5), (11). new node (45)
identifier Gxy becomes node (45)
nodetime of node (45) is 0
node (45) is not up to date: forward..
nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (11) is 7
node (11) is input terminal: back.

recalculating (45).
execute : n44d5/n11d0;

found that identifier Gxx is node (44, 6)
found that identifier averages is node (11)
new operation 1 / 1 2, (44, 6), (11). new node (46)
identifier Gxx becomes node (46)
nodetime of node (46) is 0
node (46) is not up to date: forward..
nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (11) is 7
node (11) is input terminal: back.
recalculating (46).
execute : n44d6/n11d0;

found that identifier Gyy is node (44, 7)
found that identifier averages is node (11)
new operation 1 / 1 2, (44, 7), (11), new node (47)
identifier Gyy becomes node (47)
nodetime of node (47) is 0
node (47) is not up to date: forward..
nodetime of node (44) is 4
node (44) : no recalculation needed

nodetime of node (11) is 7
node (11) is input terminal: back.
recalculating (47).
execute : n44d7/n11d0;

found that identifier Gxy is node (45)
found that identifier Gxx is node (46)
new operation 1 / 1 2, (45), (46), new node (48)
new identifier tff is node (48)
nodetime of node (48) is 0
node (48) is not up to date: forward..
nodetime of node (45) is 7
node (45) : no recalculation needed

nodetime of node (46) is 7
node (46) : no recalculation needed
recalculating (48).
execute : n45d0 ./n46d0;

found that identifier Gxy is node (45)
new operation 4 conj 1 1, (45), new node (49)
found that identifier Gxx is node (45)
new operation 1 . * 1 2, (49), (45), new node (50)
found that identifier Gxx is node (46)
new operation 1 . * 1 2, (49), (46), new node (51)
new identifier tff is node (46)
nodetime of node (46) is 0
node (46) is not up to date: forward..
nodetime of node (49) is 0
node (49) is not up to date: forward..
nodetime of node (49) is 7
node (49) : no recalculation needed
recalculating (49).
execute : tmp0=conj(n45d0);

nodetime of node (45) is 7
node (45) : no recalculation needed
recalculating (50).
execute : n49d0 .* n45d0;

nodetime of node (51) is 0
node (51) is not up to date: forward..
nodetime of node (46) is 7
node (46) : no recalculation needed

nodetime of node (47) is 7
node (47) : no recalculation needed
recalculating (51).
execute : n46d0 .* n47d0;

recalculating (52).
execute : n50d0 ./ n51d0;

found that identifier tff is node (48)
new operation 4 ifft 1 2, (48), new node (53)
new identifier impuls is node (53)
nodetime of node (53) is 0
node (53) is not up to date: forward..
nodetime of node (48) is 7
node (48) : no recalculation needed
recalculating (53).
execute : tmp0=ifft(n48d0);

found that identifier Gxy is node (46)
new operation 4 ifft 1 2, (46), new node (54)
new identifier corr1 is node (54)
nodetime of node (54) is 0
node (54) is not up to date: forward..
nodetime of node (46) is 7
node (46) : no recalculation needed
recalculating (54).
execute : tmp0=ifft(n46d0);

found that identifier Gyy is node (47)
new operation 4 ifft 1 2, (47), new node (55)
new identifier corr2 is node (55)
nodetime of node (55) is 0
node (55) is not up to date: forward..
nodetime of node (47) is 7
node (47) : no recalculation needed
recalculating (55).
execute : tmp0=ifft(n47d0);

found that identifier Gxy is node (45)
new operation 4 ifft 1 2, (45), new node (56)
new identifier cross is node (56)
nodetime of node (56) is 0
node (56) is not up to date: forward..
nodetime of node (45) is 7
node (45) : no recalculation needed

recalculating (56).
execute : tmp0=ifft(n45d0);

found that constant 0 is node (12)
found that constant 1 is node (7)
found that identifier x is node (44, 1)
new operation 4 length 1 1, (44, 1). new node (57)
found that constant 1 is node (7)
new operation 1 - 1 2, (57), (7). new node (58)
new operation 8 : 1 3, (12), (7), (58). new node (59)
found that identifier fsACQ is node (4)
new operation 1 * 1 2, (59), (4). new node (60)
found that identifier x is node (44, 1)
found that operation 4 length 1 1, (44, 1). is node (57)
new operation 1 / 1 2, (60), (57). new node (61)
new identifier Ax is node (61)

nodetime of node (61) is 0
node (61) is not up to date: forward..
nodetime of node (60) is 0
node (60) is not up to date: forward..
nodetime of node (59) is 0
node (59) is not up to date: forward..
nodetime of node (12) is 8
node (12) is constant.

nodetime of node (7) is 6
node (7) is constant.

nodetime of node (58) is 0
node (58) is not up to date: forward..
nodetime of node (57) is 0
node (57) is not up to date: forward..
nodetime of node (44) is 4
node (44) : no recalculation needed

recalculating (57).
execute : tmp0=length(n44d1);

nodetime of node (7) is 6
node (7) is constant.

recalculating (58).
execute : n57d0-n7d0;

recalculating (59).
execute : n12d0:n7d0:n58d0;

nodetime of node (4) is 2
node (4) : no recalculation needed

recalculating (60).
execute : n59d0*n4d0;

nodetime of node (57) is 4
node (57) : no recalculation needed

recalculating (61).
execute : n60d0:n57d0;

found that identifier Ax is node (61)
assign 20 to node (62) at time 13
new constant 20 is node (62)
found that identifier tff is node (48)
new operation 4 abs 1 1, (48). new node (63)
new operation 4 log10 1 1, (63). new node (64)
new operation 1 * 1 2, (62), (64). new node (65)
new operation 0 plot 0 3, (61), (65). new node (66)
alstone (66).
nodetime of node (66) is 0
node (66) is not up to date: forward..
nodetime of node (61) is 4
node (61) : no recalculation needed

nodetime of node (65) is 0
node (65) is not up to date: forward..
nodetime of node (62) is 13
node (62) is constant.

nodetime of node (64) is 0
node (64) is not up to date: forward..
nodetime of node (63) is 0
node (63) is not up to date: forward..
nodetime of node (48) is 7
node (48) : no recalculation needed

recalculating (63).
execute : tmp0=abs(n48d0);

recalculating (64).
execute : tmp0=log10(n63d0);

recalculating (65).
execute : n62d0*n64d0;

recalculating (66).
execute : plot(n61d0,n65d0);

assign 0.1 to node (67) at time 14
new constant 0.1 is node (67)
new operation 0 pause 0 1, (67). new node (68)
alstone (68).
nodetime of node (68) is 0
nodetime of node (67) is 14
node (67) is constant.

execute : pause(n67d0);
recalculating (68).
7.6.3 Graph structure

The statistics on the generated nodes include the node’s result (for array values only the first element’s value is reported), children, number of identifiers pointing to the node, number of identifiers through which the node is reachable (“users”), and whether it is a thread end node, for each of the node’s output values (a node can have several output values).

Operations are identified by type, operation string, number of results and number of arguments. There are 21 operation types, from unary operates to matrix assignment using subrange. The operation code is followed by the list of arguments.

The set of input terminals, when applicable, and the time of the most recent input on which the node depends, are followed by the node time.

7.6.3.1 List of nodes

< (0) > ((1e+07) childrenList= : (4). childrenCount 1 idCount 1 users 46 threadEnd) time 1, constant.

< (1) > ((7) childrenList= : (3). childrenCount 1 idCount 1 users 46) inputset= : (1). inputtime 2 time 2.

< (2) > ((2) childrenList= : (3). childrenCount 1 idCount 0 users 67 threadEnd) time 3, constant.

< (3) > ((128) childrenList= : (4). childrenCount 1 idCount 0 users 45) 1 ^ 1 2 arguments= (2), (1). inputset= : (1). inputtime 2 time 2.

< (4) > ((78125) childrenList= : (44), (60). childrenCount 3 idCount 2 users 45 threadEnd) 1 / 1 2 arguments= (0), (3). inputset= : (1). inputtime 2 time 2.

< (5) > ((1) childrenList= : (44). childrenCount 1 idCount 1 users 22 threadEnd) inputset= : (5). inputtime 4 time 4.

< (6) > ((1024) childrenList= : (44). childrenCount 1 idCount 1 users 44 threadEnd) time 5, constant.

< (7) > ((1) childrenList= : (44), (58), (59). childrenCount 3 idCount 0 users 67 threadEnd) time 6, constant.

< (8) > ((1). childrenCount 0 idCount 0 users 22) 4 rand 1 2 arguments= (6), (7). time 1, constant.

< (9) > ((1). childrenCount 0 idCount 0 users 22) 1 .* 1 2 arguments= (8), (2). time 1, constant.

< (10) > ((1) childrenList= : (44). childrenCount 1 idCount 1 users 22 threadEnd) 1 - 1 2 arguments= (9), (7).
time 1, constant.

< (11) > ((1) childrenList= : (45), (46), (47). childrenCount 3 idCount 1 users 15 threadEnd)
inputset= : (11). inputtime 7
time 7.

< (12) > ((0) childrenList= : (44), (59). childrenCount 8 idCount 0 users 148 threadEnd)
time 8, constant.

< (13) > ((2) childrenList= : (15). childrenCount 1 idCount 2 users 3 invalidChildren threadEnd)
inputset= : (13). inputtime 12
time 12.

< (14) > ((1) childrenList= : (15). childrenCount 2 idCount 3 invalidChildren threadEnd)
inputset= : (14). inputtime 11
time 11.

< (15) > ((0). childrenCount 0 idCount 0 users 1)
1 <= 1 2 arguments= (13), (14).
inputset= : (13), (14). inputtime 12
time 12.

< (16) > ((1) childrenList= : (18). childrenCount 1 idCount 1 users 22)
ininputset= : (16). inputtime 0
time 6.

< (17) > ((1) childrenList= : (18). childrenCount 1 idCount 0 users 21 threadEnd)
time 9, constant.

< (18) > ((2) childrenList= : (22), (24). childrenCount 2 idCount 1 users 21 threadEnd)
1 + 1 2 arguments= (16), (17).
ininputset= : (16). inputtime 6
time 6.

< (19) > ((1024) childrenList= : (24), (26). childrenCount 2 idCount 2 users 22 threadEnd)
ininputset= : (19). inputtime 0
time 5.

< (20) > ((1) childrenList= : (22). childrenCount 1 idCount 2 users 12 threadEnd)
ininputset= : (20). inputtime 0
time 1.

< (21) > ((78125) childrenList= : (22). childrenCount 1 idCount 2 users 12 threadEnd)
ininputset= : (21). inputtime 0
time 2.

< (22) > ((0) childrenList= : (26). childrenCount 1 idCount 0 users 10 threadEnd)
0 generate 0 2 arguments= (20), (21), (18).
ininputset= : (20), (21), (16). inputtime 6
time 6.

< (23) > ((78125) childrenList= : (24). childrenCount 1 idCount 2 users 12 threadEnd)
ininputset= : (23). inputtime 0
time 2.

< (24) > ((0) childrenList= : (26). childrenCount 1 idCount 0 users 10 threadEnd)
0 setup 0 2 arguments= (23), (19), (18).
ininputset= : (23), (19), (16). inputtime 6
time 6.

< (25) > ((1) childrenList= : (26). childrenCount 1 idCount 2 users 12 threadEnd)
inputset= : (25). inputtime 0

time 4.

< (26) > ((1) childrenList= : (29). childrenCount 1 idCount 1 users 5, (1) childrenList= : (31). childrenCount 1 idCount 1 users 5)
4 measure 2 1 arguments= (19), (22), (24), (25).
inputset= : (19), (20), (21), (16), (23), (25). inputtime 6
time 6.

< (27) > ((0). childrenCount 0 idCount 1 users 1)
inputset= : (27). inputtime 0
time 8.

< (28) > ((0). childrenCount 0 idCount 1 users 1)
inputset= : (28). inputtime 0
time 8.

< (29) > ((1) childrenList= : (34), (38). childrenCount 2 idCount 1 users 4)
4 fft 1 2 arguments= (26).
inputset= : (19), (20), (21), (16), (23), (25). inputtime 6
time 6.

< (30) > ((0). childrenCount 0 idCount 1 users 1)
inputset= : (30). inputtime 0
time 8.

< (31) > ((1) childrenList= : (35), (41), (42). childrenCount 3 idCount 1 users 4)
4 fft 1 2 arguments= (26, 1).
inputset= : (19), (20), (21), (16), (23), (25). inputtime 6
time 6.

< (32) > ((0). childrenCount 0 idCount 1 users 1)
inputset= : (32). inputtime 0
time 8.

< (33) > ((0) childrenList= : (36). childrenCount 1 idCount 1 users 2 threadEnd)
inputset= : (33). inputtime 0
time 8.

< (34) > ((1) childrenList= : (35), (38). childrenCount 2 idCount 0 users 2)
4 conj 1 1 arguments= (29).
inputset= : (19), (20), (21), (16), (23), (25). inputtime 6
time 6.

< (35) > ((1) childrenList= : (36). childrenCount 1 idCount 0 users 1 threadEnd)
1 . 1 2 arguments= (34), (31).
inputset= : (19), (20), (21), (16), (23), (25). inputtime 6
time 6.

< (36) > ((1). childrenCount 0 idCount 1 users 1)
1 + 1 2 arguments= (33), (35).
inputset= : (33), (19), (20), (21), (16), (23), (25). inputtime 8
time 8.

< (37) > ((0) childrenList= : (39). childrenCount 1 idCount 1 users 2 threadEnd)
inputset= : (37). inputtime 0
time 8.

< (38) > ((1) childrenList= : (39). childrenCount 1 idCount 0 users 1 threadEnd)
1 . 1 2 arguments= (34), (29).
inputset= : (19), (20), (21), (16), (23), (25). inputtime 6
< (39) > ((1). childrenCount 0 idCount 1 users 1)
1 + 1 2 arguments= (37), (38).
inputset= : (37), (19), (20), (21), (16), (23), (25). inputtime 8
time 8.

< (40) > ((0) childrenList= : (43). childrenCount 1 idCount 1 users 2 threadEnd)
inputset= : (40). inputtime 0
time 8.

< (41) > ((1) childrenList= : (42). childrenCount 1 idCount 0 users 1)
4 conj 1 1 arguments= (31).
inputset= : (19), (20), (21), (16), (23), (25). inputtime 6
time 6.

< (42) > ((1) childrenList= : (43). childrenCount 1 idCount 0 users 1 threadEnd)
1 .* 1 2 arguments= (41), (31).
inputset= : (19), (20), (21), (16), (23), (25). inputtime 6
time 6.

< (43) > ((1). childrenCount 0 idCount 1 users 1)
1 + 1 2 arguments= (40), (42).
inputset= : (40), (19), (20), (21), (16), (23), (25). inputtime 8
time 8.

< (44) > ((2). childrenCount 0 idCount 1 users 1,
1 (1) childrenList= : (57), childrenCount 1 idCount 1 users 3,
(1). childrenCount 0 idCount 1 users 1,
(1). childrenCount 0 idCount 1 users 1,
(1). childrenCount 0 idCount 1 users 1,
(1) childrenList= : (45). childrenCount 1 idCount 0 users 6 threadEnd,
(1) childrenList= : (46). childrenCount 1 idCount 0 users 5 threadEnd,
(1) childrenList= : (47). childrenCount 1 idCount 0 users 3 threadEnd)
6 8 1 arguments= (7), (6), (10), (4), (4), (5), (12), (12), (12), (12), (12), (12), (12), (12), (12).
inputset= : (1), (5). inputtime 4
(14) (11) (14). time 4 loopEnd (15).

< (45) > ((1) childrenList= : (48), (49), (50), (56), childrenCount 4 idCount 1 users 6)
1 / 1 2 arguments= (44), (45), (11).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (46) > ((1) childrenList= : (48), (51), (54). childrenCount 3 idCount 1 users 5)
1 / 1 2 arguments= (44), (6), (11).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (47) > ((1) childrenList= : (51), (55). childrenCount 2 idCount 1 users 3)
1 / 1 2 arguments= (44), (7), (11).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (48) > ((1) childrenList= : (53), (63). childrenCount 2 idCount 1 users 2)
1 / 1 2 arguments= (45), (46).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (49) > ((1) childrenList= : (50). childrenCount 1 idCount 0 users 1)
4 conj 1 1 arguments= (45).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (50) > ((1) childrenList= : (52). childrenCount 1 idCount 0 users 1)
1 * 1 2 arguments= (49), (45).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (51) > ((1) childrenList= : (52). childrenCount 1 idCount 0 users 1)
1 * 1 2 arguments= (46), (47).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (52) > ((1). childrenCount 0 idCount 1 users 1)
1 ./ 1 2 arguments= (50), (51).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (53) > ((1). childrenCount 0 idCount 1 users 1)
4 ifft 1 2 arguments= (48).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (54) > ((1). childrenCount 0 idCount 1 users 1)
4 ifft 1 2 arguments= (46).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (55) > ((1). childrenCount 0 idCount 1 users 1)
4 ifft 1 2 arguments= (47).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (56) > ((1). childrenCount 0 idCount 1 users 1)
4 ifft 1 2 arguments= (45).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (57) > ((1024) childrenList= : (58), (61). childrenCount 2 idCount 0 users 2)
4 length 1 1 arguments= (44, 1).
inputset= : (1), (5). inputtime 4
time 4.

< (58) > ((1023) childrenList= : (59). childrenCount 1 idCount 0 users 1)
1 - 1 2 arguments= (57), (7).
inputset= : (1), (5). inputtime 4
time 4.

< (59) > ((1024) childrenList= : (60). childrenCount 1 idCount 0 users 1)
8 : 1 3 arguments= (12), (7), (58).
inputset= : (1), (5). inputtime 4
time 4.

< (60) > ((1024) childrenList= : (61). childrenCount 1 idCount 0 users 1)
1 * 1 2 arguments= (59), (4).
inputset= : (1), (5). inputtime 4
time 4.

< (61) > ((1024) childrenList= : (66). childrenCount 1 idCount 1 users 1 threadEnd)
1 / 1 2 arguments= (60), (57).
inputset= : (1), (5). inputtime 4
time 4.

< (62) > ((20) childrenList= : (65). childrenCount 1 idCount 0 users 0 threadEnd)
time 13, constant.

< (63) > ((1) childrenList= : (64). childrenCount 1 idCount 0 users 0)
4 abs 1 1 arguments= (48).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (64) > ((1) childrenList= : (65). childrenCount 1 idCount 0 users 0)
4 log10 1 1 arguments= (63).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (65) > ((1) childrenList= : (66). childrenCount 1 idCount 0 users 0)
1 * 1 2 arguments= (62), (64).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (66) > ((0). childrenCount 0 idCount 0 users 0)
0 plot 0 3 arguments= (61), (65).
inputset= : (1), (5), (11). inputtime 7
time 7.

< (67) > ((0.1). childrenCount 0 idCount 0 users 0)
time 14, constant.

< (68) > ((0). childrenCount 0 idCount 0 users 0)
0 pause 0 1 arguments= (67).
time 1, constant.

7.6.3.2 List of identifiers
fCLK : (0),
divider : (1),
m : (44),
fsACQ : (4),
fsGEN : (4),
dut : (5),
n : (6),
signaal : (10),
averages : (11),
x : (44, 1),
y : (44, 2),
X : (44, 3),
Y : (44, 4),
Gxy : (45),
Gxx : (46),
Gyy : (47),
tff : (48),
coh : (52),
impuls : (53),
corr1 : (54),
corr2 : (55),
cross : (56),
Ax : (61)
7.6.3.3 List of constants
(0), (2), (6), (7), (12), (62), (67).

7.6.3.4 List of operations
1 ^ 1 2, (2), (1). : (3),
1 / 1 2, (0), (3). : (4),
4 rand 1 2, (6), (7). : (8),
1 .* 1 2, (8), (2). : (9),
1 - 1 2, (9), (7). : (10),
1 / 1 2, (44, 5), (11). : (45),
1 / 1 2, (44, 6), (11). : (46),
1 / 1 2, (44, 7), (11). : (47),
1 ./ 1 2, (45), (46). : (48),
4 conj 1 1, (45). : (49),
1 .* 1 2, (49), (45). : (50),
1 .* 1 2, (46), (47). : (51),
1 / 1 2, (50), (51). : (52),
4 ifft 1 2, (48). : (53),
4 ifft 1 2, (46). : (54),
4 ifft 1 2, (47). : (55),
4 length 1 1, (44, 1). : (57),
1 - 1 2, (55), (7). : (58),
8 : 1 3, (12), (7). : (59),
1 * 1 2, (59), (4). : (60),
1 / 1 2, (60), (57). : (61),
4 abs 1 1, (48). : (63),
4 log10 1 1, (63). : (64),
1 * 1 2, (62), (64). : (65),
0 plot 0 3, (61), (65). : (66),
0 pause 0 1, (67). : (68)
8. Conclusion and guidelines

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 will be tempted to recalculate the results, thus defeating the purpose of avoiding redundant operations.

To prevent the redundant execution of an operation automatically, 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.

Caching of intermediate results will be useful:

1. In situations where a program is repeatedly executed, either with partially changed input values, or with small modifications to the algorithm. In these situations, parts of the program will be repeatedly executed without any change to the argument values or to the operations.

2. When operations frequently result in unchanged values even though their arguments changed. This can also lead to the execution of programs parts with the same values as during a previous execution.

Caching can avoid the redundant operations that occur in those situations, and should be considered whenever the resources consumed by the redundant operations outweigh the increase in complexity due to caching. To determine the savings possible through caching for a specific program, an analysis of the data dependencies in the program is required. Caching efficiency is influenced by the structure of the dataflow graph of the
algorithm. The graph structure will determine the number of nodes that are affected by a change and require recalculation.

There are several advantages to integrating caching in the execution scheme of a dataflow-based environment. In virtual instrumentation environments based on dataflow diagrams, the available data dependency information can be used to:

1. Reduce the cache overhead. 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. Reducing the caching overhead is useful when this overhead is substantial relative to the calculation cost.

2. Reduce the amount of intermediate results stored. 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 on which it depends. The reduction of intermediate results to store will be most important when the data used in the program is of considerable size or the program must run on a small system.

Different schemes to integrate caching in a dataflow environment were presented. The 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.

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 called 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 only overhead in the scheme consists of comparing new input settings with previous values.

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 do not have to execute and can return their cached result, also marked as “unchanged”.

177
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 called 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.

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.

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 easily determined quantitative properties, representative of each graph, were derived. The properties that affect performance can be divided into node properties, structural properties, interaction properties, and environment 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 was 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 in the graph. 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. In general, caching is more effective for graphs consisting of few, large threads that 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 was 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 was found that the dataflow execution method (data driven or demand driven) does not significantly influence the relative performance of the caching schemes (it does influence overall performance).
The trade-off between calculation cost and comparison overhead will be determined by how these factors compare for a specific system. This overhead importance 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.

The performance of the caching schemes depends on numerous factors. In our simulations on random generated graphs, none of the schemes is consistently superior. Whether caching is appropriate or whether using the data dependency information will be beneficial for a given problem must be determined case by case.

Regions of scheme effectiveness can be defined in the multi-dimensional parameter space. For several sample algorithms, the performance was calculated mathematically, and the exact results for these algorithms conform reasonably well to the simulations. The regions of scheme effectiveness allow one to estimate the caching scheme most likely to be effective for a given algorithm, from general properties of the algorithm’s dataflow diagram, its expected ChangedFraction, and the estimated overhead importance. For simple algorithms, mathematical methods can also be used to determine the amount of performance benefits possible.

Considering the research presented in the previous chapters, a number of guidelines can be established for the designer of software development environments on the one hand and for the software developer on the other. All the same, a course of action can only be determined by allowing for the particular circumstances of each case.

8.1 Guidelines for the environment designer

The environment designer will have to consider a number of factors to base his design choices on, such as:

- How many times does the available memory size on the intended platform exceed the total data size of all intermediate results? This will determine whether caching is possible, and how many previous results can be stored for each operation. If the memory available for caching barely exceeds the total data size, the use of dataflow information to reduce the number of previous results to be stored is highly recommended. If the available memory is insufficient to store all intermediate results, the MedCache or MinCache scheme can be used.

- The allowed complexity of the environment might preclude the implementation of the demand driven execution scheme, and of the more complicated approaches to limit redundant propagation.

- Is the dataflow propagation overhead considerable on the intended platform? If the overhead caused by redundant propagation (through graph sections that are already up-to-date) is considerable, the environment user should be encouraged to use subgraph encapsulation. Solutions such as input terminal sets in the demand driven execution scheme should preferably be avoided.
• The performance of cache retrieval that can be achieved in the environment will partly determine the importance of the caching overhead. To determine whether the cached result of, for example, an operation on a matrix is still valid, the current argument matrix must be compared with the previous one. If the two matrices differ completely, a full element by element comparison should not be needed to establish this. The average performance will be difficult to determine if it depends on the cache hit rate. If this overhead can be made negligible, the choice of the caching scheme will only depend on the memory available for caching.

• The nature of the programs that will be developed in the environment may hint at the predominant range of program graph parameters that will occur. Together with the factors already mentioned, this may determine which caching schemes, if any, are most appropriate to implement in the environment.

• The expected programming expertise of the environment user will determine the extent to which caching can be allowed to impact the use of the environment. In a dataflow-based environment, caching will demand very little additional user effort, especially when measures against redundant propagation are not required.

Depending on the circumstance, some or more of these factors may not be known in advance.

For a measurement system development environment, caching of intermediate results should certainly be considered as a viable feature, as it can significantly reduce the redundant operations that typically occur during repeated executions of a measurement program with slight variations.

However, possibly memory restrictions and caching overhead can make caching unattractive, especially when intermediate results are large. In a dataflow-based environment, the use of available dataflow information should be considered as a method to reduce caching overhead and memory requirements.

Depending on the nature of the dataflow extensions in the environment, using dataflow information may not be possible or may impose restrictions on caching. In an environment using dynamic dataflow execution, for example, the dataflow graph changes dynamically between executions, and data dependencies between cached values during one execution will have no meaning during the next execution.

If the environment designer wants to include caching, but must restrict overhead or memory requirements, the integration in the environment of dataflow extensions that obscure the data dependency relations should be avoided. Given the incredible variety of possible dataflow extensions, it is difficult to offer specific guidelines on the implementation of extensions.

Regarding the execution scheme, data driven and demand driven execution really do not differ very much once dataflow extensions are introduced.

In pure dataflow graphs, the performance of the demand driven scheme, compared to the data driven scheme, depends mostly on the number of output nodes in the graph. Demand driven execution is more effective for graphs with multiple output terminals,
because the multiple output terminals offer a limited means to conditional execution not available in data driven execution (something similar could probably be implemented in pure data driven dataflow using extra dummy input terminals, but it would be a very artificial solution).

When the demand driven performance is normalised to the fraction of the graph needed to calculate the requested output, the effect of caching is comparable to its effect under data driven execution. The execution scheme (data driven or demand driven) does not fundamentally influence which caching scheme is most effective.

Demand driven execution can be considered more general with the data driven scheme as a special case, however, demand driven execution is more complex to implement, because there is a propagation of demands as well as of results, and with caching, the need to maintain consistency on multiple executions.

Data driven execution, when provided with extensions that allow upstream placement of conditions, offers the same level of control as demand driven execution, without the need for multiple output terminals.

Propagation overhead is the overhead created by propagating demands or activation tokens from node to node through the dataflow graph. Propagation through graph parts that do not require recalculation is redundant. Unless propagation overhead is really very important (for example on a distributed system), it will be better to leave the propagation method unchanged. Refinements like input terminal sets work great in theory but could lead to undeterministic execution when used in combination with dataflow extensions. It is preferable to have the execution scheme visit all nodes related to the requested result.

The propagation overhead will be most annoying for a finely grained dataflow graph, consisting of many simple nodes. The environment should provide granularity options to reduce propagation overhead, like encapsulation of a subgraph into an enclosing node abstraction. Only when the enclosing node must be executed, will the subgraph be evaluated. This works in data driven dataflow as well as in demand driven dataflow, and when used intelligently, can prevent redundant propagation almost as well as input terminal sets.

Since no caching scheme is consistently superior, the environment designer should offer several schemes as an option for the user, unless some deciding factors are already known during the design process. Assuming caching is possible:

- If memory size will be severely restricted, the MaxCache scheme is a poor candidate.
- If memory is not a problem, the usefulness of the MedCache scheme is very limited.
- If overhead is negligible but memory is restricted, the MedCache scheme is likely preferable.
- If memory is unlimited and overhead negligible, there is no point in using dataflow information at all.
8.2 Guidelines for the environment user

The user of the development environment can estimate the most effective caching scheme for his program based on the general properties of the program graph, or on a mathematical performance calculation for the specific problem if the program graph allows (assuming full availability of all options in suitable powerful implementations).

Besides the information derived from the dataflow graph, the user will have to determine several other parameters:

- The expected pattern of user interactions, and whether redundant operations are likely to occur because of it. An estimation of the ChangedFraction.
- The probability that operations in the graph may produce the same results for changed argument values. An estimation of the ChanceFactor.
- Memory restrictions imposed by the platform used.
- The data size of intermediate results and the calculation complexity of the algorithm, also the relative complexity of a cache look-up in the chosen environment.

Depending on the values of these parameters, the developer can select a caching scheme using the methods discussed in the previous chapters.

Regarding the structure of the dataflow program, there is little the developer can do to make the structure more suitable to a caching scheme, since the data dependencies are determined by the algorithm. The dataflow extensions offered by the environment should be adapted to the caching schemes by the environment designer. If propagation overhead is important, related nodes (from the same thread for example) should be grouped in a subgraph to reduce the propagation overhead. If the caching scheme alters the execution method (through input terminal sets for example), the user must ensure that his program’s execution is deterministic and consistent.
ancestors of a node
• input terminals the node depends on, directly or through its parents.

arc, branch
• representation of a data dependency in a dataflow graph.
• link over which data is exchanged in a dataflow environment.

BranchFactor
• the number of output results of a node or input terminal. More specifically, the average value for the graph, which is equal to the number of arcs in the graph over the number of nodes (including input terminals).

cache
• memory reserved for caching.

caching
• keeping a value in memory after its use, under the assumption that it might be useful again.

caching overhead
• overhead caused by comparing new values to cached results.

calculation cost
• execution cost of operations.

ChanceFactor
• the probability for a node to produce the same results for changed arguments. More specifically, the average value for the graph.

ChangedFraction
• the number of input terminals changed in an interaction, relative to the number input terminals.

children of a node
• nodes that use result(s) of the node as argument(s), downstream neighbours of a node.
data driven
• dataflow execution method where a node executes as soon as all its arguments are available.

data token, activation token
• representation of data exchanged between nodes.

demand driven
• dataflow execution method where a node executes only upon demand.

downstream
• direction in which data propagates through the graph, from input terminals to output terminals.

graph, dataflow graph
• the graphical representation of data dependencies between operations in a specific algorithm, consisting of nodes, arcs, and terminals.
• the graphical program in a dataflow-based environment, including extensions not representing data dependency.

graph size
• the number of nodes in a graph, including input terminals unless otherwise stated.

input, node input, node argument
• an incoming arc to a node, and the source of this arc.

InputFraction
• the number of input terminals relative to the number of nodes (including input terminals) in the graph.

input terminal
• a “source” of data in the dataflow graph, user input.

interaction
• a number of input terminal values are changed, and the graph is executed (an output terminal is requested for demand driven execution).

MaxCache
• caching scheme where all nodes are cached, and compare new values to previous ones.

MedCache
• caching scheme where input terminals and end nodes of threads are cached, and compare new values to previous ones.

MinCache
• caching scheme where input terminals and end nodes of threads are cached, and only input terminals compare new values to previous ones.

NoCache
• dataflow execution without caching.
node
• a graph element representing a data generating or transforming operation, includes input terminals unless otherwise stated.

node output, node result
• A node can have several outputs, and an output can have several outgoing arcs connected to it.

output node
• a node that has an output terminal attached to it, a node whose result is directly available to the user.

OutputNumber
• the number of output terminals of the graph.

output terminal
• a “sink” of data in the dataflow graph, representing that the node result to which it is attached is directly available to the user. Not in itself a graph element.

overhead importance
• a factor representing the importance of caching overhead, relative to the calculation cost. It depends on the complexity of the operations and the implementation of caching.

parents of a node
• nodes whose results are used as arguments by the node, upstream neighbours of a node.

path
• an alternating sequence of nodes and arcs, with each arc being directed from the node immediately preceding to the one immediately succeeding it in the sequence.

performance
• the calculation cost when using caching, relative to the cost without caching.
• the caching overhead when using dataflow information, relative to overhead without using dataflow information (which is approximated by the total data size of the graph).

redundant operation
• an operation whose execution is redundant at the moment because it is known that this will produce the same result as its previous execution.

thread
• the largest collection of one or more connected nodes with identical ancestors. Each node or input terminal belongs to one thread.

thread end nodes, end nodes of a thread
• nodes that have children outside of the thread.
ThreadLength
• the number of nodes or input terminals in a thread. More specifically, the average value for the graph.

ThreadFraction
• the number of threads relative to the number of nodes (including input terminals) in the graph.

total cost
• sum of the calculation cost and the caching overhead (the caching overhead is multiplied by the overhead importance).

upstream
• direction from output terminals to input terminals, reverse from downstream.
References


16 T. Green, “Noddy’s guide to visual programming”, Interfaces, autumn 95, British Computer Society, human-computer interaction group.


20 M. Beck, R. Johnson, K. Pingali, “From Control Flow to Dataflow”, TR 89-1050, October 1989, Department of Computer Science, Cornell University.


23 W. Yang, S. Horwitz, T. Reps, Detecting program components with equivalent behaviors, University of Wisconsin - Madison computer sciences technical report #840, April 1989.


55 M. Kendall, Kendall’s Advanced Theory of Statistics, Vol. 1; Distribution Theory, A. Stuart, J. K Ord (editors), Charles Griffin & Company Ltd.