Matching and Covering with Multiple-Output Patterns

Marnix Arnold
Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Technique
marnix@cardit.et.tudelft.nl

January 14, 1999

Technical Report 1-68340-44(1999)-01
DRAFT VERSION

1 Introduction

The algorithms and data structures described in this report are aimed at the automatic detection of patterns of instructions in a (dynamic) execution trace of a program. Two separate phases can be distinguished: a pattern library construction phase, and a matching and covering phase.

During the library construction phase, all patterns up to a certain size are found and their occurrence frequencies are noted. The first part of the matching and covering phase consists of a matching sub-phase, in which all occurrences in the execution trace of all patterns from a given pattern library are found. This library can be user-specified, but the patterns found during the pattern library construction phase can also be used. In the second part, the covering sub-phase, a selection is made from the matched patterns in such a way that the data-ready time of the slowest trace output is minimized when the selection (cover) is implemented.

This report describes the algorithms used in the matching and covering phase. The pattern library construction phase uses a modified version of the matching algorithm and will be described last.

In the next section, an overview of related work is given. The strategy for library construction, matching and covering using partial matches is explained by means of an example in section 3. In section 4 a description of the data structures used in the implementation of partial matching and covering can be found. Sections 5 and 6 contain in-depth descriptions of the matching and covering algorithms, respectively. The library construction algorithm is discussed as part of the matching algorithm in section 5.5. Conclusions are stated in section 7.

2 Related Work

Matching and covering algorithms are well-known in the fields of code generation and logic synthesis. Keutzer [1] was the first to recognize the similarity between the software compiler's task of generating code and the technology mapping problem in automated VLSI design. Both problems can be handled with a matching algorithm, to find all possible instantiations of patterns (instructions or standard cells), followed by a covering algorithm to make a selection of matches that optimizes some criterion (code size, VLSI area or latency, etc.).

As described in [3], there are two main approaches to handling the matching problem when performing technology mapping: the Boolean and the structural approach.

The Boolean approach can only be applied to networks of Boolean functions, since it operates by checking the equivalence of functional representations of the patterns and functions representing portions of the network. This equivalence is detected using (Ordered) Binary Decision Diagrams, which makes this technique unsuitable for networks of non-Boolean functions.

Structural matching will work on networks containing nodes of any type of function. Since this approach focuses on the identification of common patterns, the subject and pattern graphs have to be written in terms of the same types of function nodes. Most approaches ([1], [4]) require all graphs to be trees (single-output, acyclic, non-reconvergent graphs). It is therefore necessary to decompose the subject graph into a set of disjoint trees. For solving this decomposition problem only heuristical methods exist.

In a more recent work by Kukimoto et al. [2], a structural matching method was introduced that can handle DAG-shaped subject graphs, allowing reconvergence within the graph. However, patterns are still restricted to single output, tree-shaped graphs. When this method is used for technology mapping, rather than tree-mapping, faster solutions are found (up to 67% faster), at the cost of a significant increase of area (up to 126% larger). This area increase is due to the fact that the DAG mapping approach freely duplicates subject nodes whenever a pattern match covers a multiple-fanout point. This may not be a problem in VLSI synthesis, where extra cells are cheap in terms of chip area (although it may be desirable to have some control over the node duplication aggressiveness). In code generation, however, node duplication leads to extra instructions. On a machine with limited execution resources this may cause the schedule length to increase and hurt the execution speed of the code rather than helping it.

None of the previously described matching algorithms allow pattern graphs to have more than one output. Our proposed matching algorithm does not have this restriction, making it possible to exploit a larger familly of pattern graphs.

3 Introducing the matching and covering strategy

Given a subject graph G_{sub}, consisting of the operations and operands in an execution trace, we must try to find all matches between pattern graphs G_{pat} (from a pattern library $PatLib$) and subgraphs of G_{sub}. The strategy for this is described informally in section 3.1.

The search for new patterns is remarkably similar to the detection of matches with existing patterns. An extended version of the pattern detection strategy can be used, as described in section 3.2.

After all matches are found, we try to find the best *cover*, or selection of matches, that is, the set of matches that, when implemented, minimizes the data-ready time of the longest path through the subject graph. The covering approach is described informally in section 3.3.

3.1 Matching

In figure 1, the subject graph and some pattern graphs are shown. Note that it is generally necessary to have patterns for all individual operations, to ensure that there is always at least one match to fall back on for each subject graph operation node, during covering.

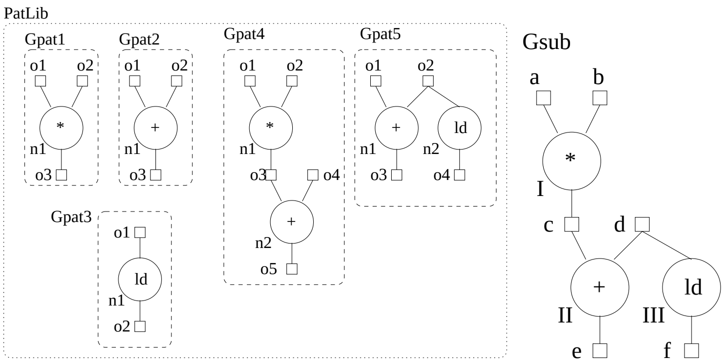


Figure 1: Example of pattern and subject graphs.

From the example it can be seen that several nodes from pattern graphs of *PatLib* correspond to nodes in G_{sub}. These relationships are called *partial matches*. A match always pertains to a single pattern graph, and it contains a vector of references to subject nodes. The position of a reference within the vector indicates which pattern node it corresponds to. Figure 2 illustrates partial matches and a match on a pattern. Since operation node I of G_{sub} matches operation node n_1 of G_{pat1}, a partial match m_1 can be created for node I and its operands. This match consists of a vector of references to subject graph nodes, corresponding to pattern graph nodes that are being matched. In the figure, m_1 is a partial match for G_{pat1} with references to subject operation node I (at the position corresponding to n_1) and subject operand nodes a (for o_1), b (for o_2) and c (for o_3). The rest of the entries in the match vector remain empty. In a similar fashion, partial match m_2 can be constructed for subject operation node II and its operands.

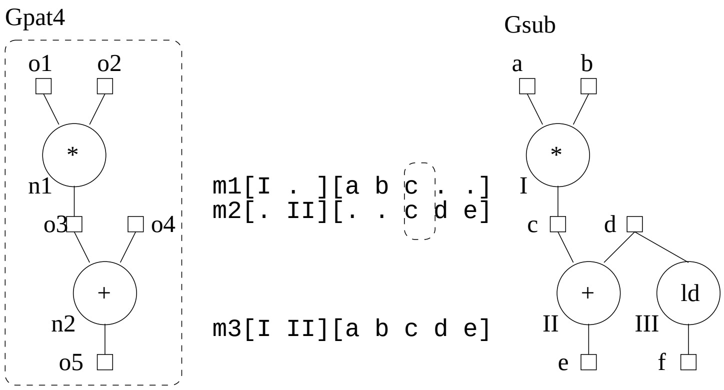


Figure 2: Partial matches merged into a complete match.

In order to get *full matches* from the partial matches that can thus be found, it is necessary to combine partial matches where possible. In our example, it can be seen that combining partial matches m_1 and m_2 yields a full match for G_{pat1}. It is obviously not correct to combine just any two matches from anywhere in the subject graph. There are several conditions under which it is allowed, however.

These will be discussed in detail in section 5. For the moment we will say that the two matches in the example may be combined because there is one vector entry the same for both matches (the third entry of both m_1 and m_2 contains a reference to operand node c). The resulting match m_3 is a full match for G_{pat1}.

After all valid combinations of partial matches have been made, all matches that are not full matches (i.e. matches with blank entries in their reference vector) can be deleted. The resulting set of matches is used to find a cover for the subject graph.

3.2 Automatic pattern library construction

In figure 1, the pattern library was introduced. It contains a number of pattern graphs, for which the matching algorithm must find matches with the subject graph. All patterns in the library were put there beforehand, which implies some advance knowledge of which patterns are expected to occur in the trace. If no such knowledge is available, however, it is useful to have a mechanism that automatically discovers which patterns occur most frequently. It is with this is mind that the pattern library construction phase is introduced.

Starting with a library that only contains basic operation patterns (patterns G_{pat1}, G_{pat2} and G_{pat3} in figure 1), we construct matches for the first subject operation node (node I) in the same way as described in section 3.1. This yields a match m_1 for G_{pat1}, as shown in figure 3. The same is done for node II, yielding match m_2. There are now two matches on operand node c and we will attempt to merge them into a new pattern. This is done by taking the union of the reference vectors of m_1 and m_2, resulting in the new match m_3.

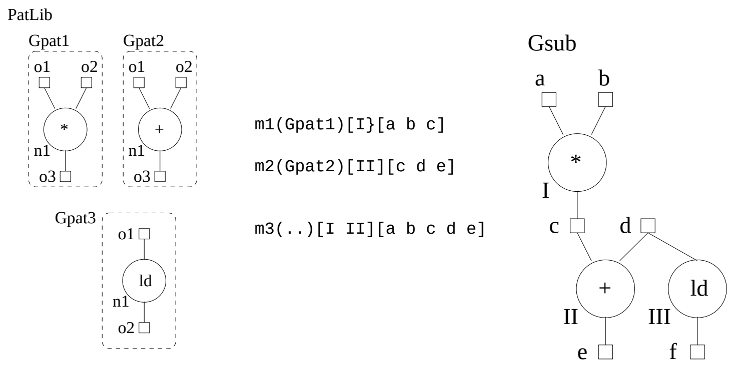


Figure 3: Pattern construction by taking the union of matches.

We can now construct a new pattern graph by copying the subject operations and operands referenced in m_3. This results in a graph G_{pat4} which is equivalent to the one also called G_{pat4} in figure 1. Before we do this, however, we must ensure that the pattern we are about to construct is not already contained in the pattern library. Rather than checking all patterns in the library against the one to be constructed (a lot of work if the library contains many patterns), we check the matches on the current operand node (d) against the newly formed m_3. If the library contains any pattern which is equivalent to the new one, then that pattern will have been matched with the exact same subject nodes, i.e. a match will have been formed on d that references all the same nodes as m_3. It is therefore sufficient to check for the existence of a complete match on d that has the same reference sets as m_3. If there is none, we can add the new pattern graph to the library.

Now that a new pattern has been added to the library, we will detect all subsequent matches for that pattern automatically. In this way, all possible combinations of operation and operand nodes can be found and kept track of as they occur in the subject graph. To prevent the number of patterns from growing unacceptably large, some limit will have to be imposed on both the number and size of the pattern graph, but theoretically any number of patterns of any size can be found.

3.3 Covering

Given a subject graph with full matches found for all nodes, we now try to find the subset of the full matches that, when implemented, minimizes the data-ready time of the slowest subject graph output. This process is called *covering*.

Until now, we have only looked at patterns as a single graph of operand and operation nodes, with the operation set of the latter being the same as that of the subject graph. However, each of these patterns, or *source graphs*, corresponds to a faster, cheaper or more efficient implementation of the same functionality. This corresponding implementation is also represented in a graph, the so-called *destination graph*. It contains the operand and operation nodes that have to be substituted into the subject graph for the nodes matching the pattern. Source and destination graphs are described in more detail in section 4.1.

In real life, a multiply operation usually takes two cycles and an addition one. The multiply-add operation however (corresponding functionally to pattern G_{pat4}) usually only takes two cycles instead of three. Figure 4 shows the pattern library of the example again, with a latency matrix for the destination graphs. The single-node patterns are their own destination graphs, they are only in the library so that there is always a pattern to fall back on if no complex patterns matched for a certain node.

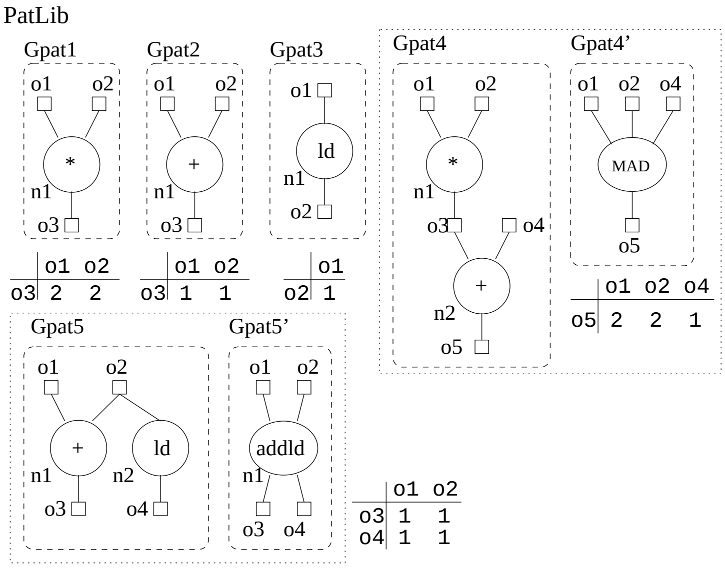


Figure 4: Pattern library with latency information.

Now that we have the latencies for the patterns, we can calculate the data-ready times that implementing the various matches would yield. Figure 5 shows the subject graph again, with all full matches that have been found in the matching phase.

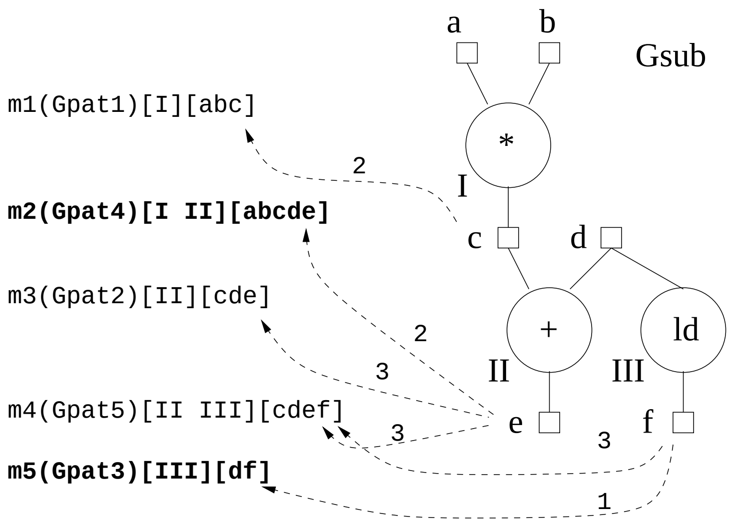


Figure 5: Subject graph with matches.

The first thing we need to do is determine which matches are so-called *output matches* for which operand nodes. A match is an output match for an operand node if that operand corresponds to an output of the pattern that is matched. In our example, m_4 is an output match of nodes e and f, but not of c and d. In figure 5, pointers are shown from each operand node to all its output matches.

The next step is to determine the ready time of each operand for each of its output matches For now, we assume a ready time of 0 for each of the subject graph inputs a, b and d. The ready time of the other operand nodes is determined by the latency through the match under consideration and the best (lowest) possible ready time of its inputs, which is in turn determined by the best applicable match there. In our example, the ready time on node e is 2 for m_2 (the maximum of the latency from any input of pattern G_{pat4} to output o_4, added to the ready time of that input, in this case 0). The ready time for m_3 is the latency through pattern G_{pat2} (from o_1 to o_3) added to the ready time of its input o_1 (corresponding to node c). The data-ready time of c in turn is found by taking the lowest possible value for all the matches found there (in this case there is only one match, m_1) which yields a ready time of 2. This results in a ready time of 3 on e for m_3. The algorithm to determine all ready times for all matches on all operands is discussed in-depth in section 6. In figure 5, the latencies for all output matches on all operand nodes are annotated on the dashed edges.

Now that we have a measure of quality for all matches on each operand node(the latency that they would attain), we can sort them by order of preference. For example, node e would have match m_2 as favourite, with m_3 and m_4 in second and third place. We can also say which subject graph output we want to give precedence when we start covering, by looking at its best possible ready time. In our example, e would have to be covered first (best latency is 2), followed by f (best latency 1)[1].

With the matches on the subject graph outputs sorted, we can now begin the actual covering. Starting at e, we choose the match that causes the lowest latency,

[1]Note that this is a heuristic 'best guess' and in no way a guarantee for an optimal solution, since it is very well possible that implementing a match on one path invalidates the favourite match on another path. For example, suppose m_4 would have been the match of choice for node III, but we would implement m_2 first, then m_4 would no longer be valid unless we resort to node duplication, which introduces a whole set of problems of its own that we will not discuss here.

in this case m_2 (latency 2). All matches that overlap m_2 have to be invalidated (m_1, m_3, m_4) since they can no longer be implemented (not that we would want to in this case). Now we proceed to cover all operands that are inputs to match m_2, but these are subject graph inputs a, b and d so we have finished covering along this path. Now we start covering on the last subject graph output: f. The best available match is m_5, which is still available, so we can implement it. We have now finished constructing a cover for the subject graph, consisting of matches m_2 and m_5.

4 Data Structures

After the general description of the workings of the matching and covering algorithms in the previous section, we can now determine what the data structures representing the various data entities should look like. They fall into three main categories: the pattern library, the subject graph and the matches. Each of these is described in the following subsections.

4.1 The pattern library

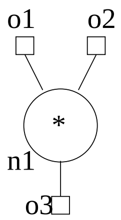
For the description of the pattern library data structures, we shall begin by describing the most basic elements first, then construct more complex elements out of these. There are several levels of hierarchy in the pattern library structure, each will be described in the following paragraphs.

Pattern graph nodes At the most basic level, there are data structures to describe pattern operation and operand nodes. These nodes are interconnected and reference each other: operation nodes refer to operand nodes as operands and results, and operand nodes refer to operation nodes as definers and users. Figure 6 shows the relation between the data structures graphically. Not shown is the type of every operand: integer, floating-point or boolean variable, or immediate, in which case its value is also stored. Also not shown are references to the parent graph of every operation node.

```
OperNode o1:
  defs: []
  uses: [n1]

OperNode o2:
  defs: []
  uses: [n1]

OperNode o3:
  defs: [n1]
  uses: []
```




```
OpNode n1:
  operands: [o1, o2]
  results:  [o3]
```

Figure 6: Pattern graph node data structures.

Pattern graphs Pattern graphs are built up out of pattern operation and operand nodes, which were described in the previous paragraph. Two types of pattern graphs can be distinguished: pattern source graphs and pattern destination graphs. Source graphs are used to match the subject graph against, they therefore only contain operations from the same operation set as the subject graph. When a match is implemented, the destination graph corresponding to the source graph is substituted into the subject graph for the matching subject nodes. Destination graphs contain operations from the extended operation set (including special functionality).

Figure 7 shows the relation between pattern source and destination graphs and their components. Not shown are the latency figures from each input to each output; these can be calculated from the latencies of the individual operations in the graphs. For efficiency reasons it makes sense to calculate all latencies ones and store them in the pattern graph data structure.

Note that the names of input and output operands for corresponding source and destination graphs must be the same, otherwise it is not clear how the substitution should be performed. Also note that it is possible for several source graphs to correspond to the same destination graph, so that commutativity can be taken into account. In figure 7, another valid source pattern would be the one where operation n_2 would have o_4 as its first operand and o_3 as its second.

```
PatSrcGraph Gpat4_src
  ops [n1, n2]
  opers [o1, o2, o3, o4, o5]
  inputs [o1, o2, o3]
  outputs [o5]
  impl Gpat4_dst

PatDstGraph Gpat4_dst
  ops [n1]
  opers [o1, o2, o4, o5]
  inputs [o1, o2, o4]
  outputs [o5]
```

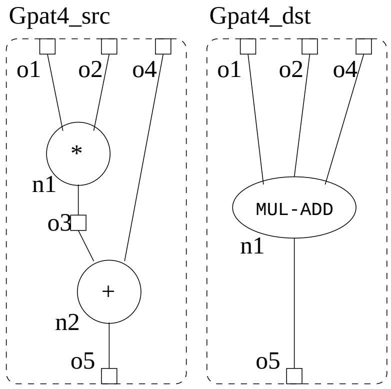


Figure 7: Pattern graph data structures.

Pattern library The pattern library is a set of pattern source graphs, for use in matching, and a set of pattern destination graphs, that are substituted during covering for the subject nodes that match the corresponding pattern source graph.

4.2 The subject graph

As was the case for the pattern graph data structures, the subject graph data structures will be described from the bottom up. Each level of hierarchy will be described in the following paragraphs.

Subject graph nodes At the lowest level of the hierarchy are the subject graph operation and operand nodes. They are roughly the same as the pattern graph operation and operand nodes, so there is no need for a detailed description here. The main difference is that the subject graph operand nodes have a list of references to matches (see section 4.3).

Subject graph The subject graph is a network of the subject graph nodes described in the previous paragraph. It represents a window in a dynamic execution trace of a program.

Register definition table

Reg	Current def
r1	a
r2	~~c~~ e
r3	d

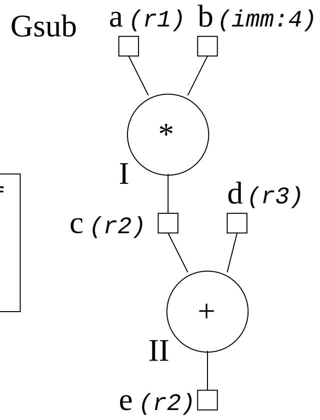


Figure 8: The register definition table.

Register definition table A special data structure has to be introduced to cope with the fact that we are dealing with a dynamic execution trace: the register definition table. During the trace simulation, each time a register is written, the corresponding register definition table entry keeps track of which operand node definition that write corresponds to. The previous definition of the register (operand node) can now no longer get additional uses. This information is necessary in order to determine partial matches, as will be described in section 5.1.

An example of the workings of the register definition table is shown in figure 8. After operation II has been executed, the current definition of register r_2 is operand node e. The previous definition, c, can no longer be used by subsequent operations in the trace simulation since its value has been overwritten.

4.3 The match

The match is the central data structure for the matching and covering algorithms. It contains a reference to the pattern source graph that it is a match for. A vector of references to subject operand nodes indicate which subject nodes match which pattern node (from the aforementioned pattern source graph). The reference vector is ordered according to the ordering of the nodes in the pattern graph, so there is a direct correspondence between pattern nodes and subject graph nodes. If all positions in the reference vector are filled, the match is said to be complete.

We define the *degree* of a match as the number of operation nodes that are encapsulated by it. For example, if all input and output operands of two operation nodes are referenced in the match, the match is said to be of degree 2. If the operations themselves would be referenced in the match, we could simply count them and arrive at the same degree. If a match is complete, its degree is the same as the number of operation nodes in the corresponding pattern graph.

The subject graph operand nodes that are referenced by a match also have a reciprocal reference, paired with an index indicating the position the operand node occupies in the match's reference vector. Figure 9 shows an example of the relations between pattern and subject nodes and the match. On subject operand node a, there is a reference to match m_1, index 1. Match m_1 references pattern $Gpat4$, so a corresponds to the pattern operand node at position 1 of $Gpat4$, which is o_1.

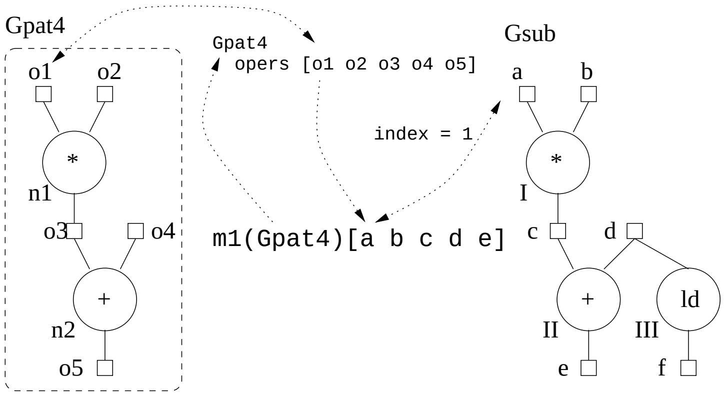


Figure 9: Relation between pattern and subject nodes and a match.

5 Matching

This section contains an in-depth description of the matching algorithm that was introduced in section 3. The algorithm makes uses of the data structures outlined in section 4. First an enumeration of the basic assumptions will be given.

Given

- A subject graph G_{sub} made up of subject operation nodes N_{sub} and subject operand nodes O_{sub}, interconnected by def- and use-edges.
- A library $PatLib$ of pattern graphs G_{pat}, consisting of pattern operation nodes N_{pat} and pattern operand nodes O_{pat}, interconnected by def- and use-edges.

The matching algorithm consists of three phases:

1. **Partial match finding:** a first degree (i.e. matching only one operation of the pattern graph) partial match construction phase.
2. **Merging matches:** the partial matches found in the previous phase are merged into higher degree partial matches and complete matches.
3. **Cleanup:** all non-complete matches are removed.

```
Matching
{
  FindPartialMatches();
  MergeMatches();
  CleanupMatches();
}
```

The three phases are described in the following subsections. A variation, integrating the first two phases into one, is described in subsection 5.4. Section 5.5 contains the description of an extension of the matching algorithm that detects patterns and stores them in the pattern library.

5.1 Partial match finding

In the partial match finding phase, we determine for each subject graph operation node which pattern graph operation nodes it corresponds to. We only look at the nodes themselves (and their operands), not their preceding and succeeding operation node context. This potentially leads to a lot of partial matches that can never become full matches, but it reduces the computational complexity of the algorithm. A tradeoff between computational complexity and number of matches created can be made by adding criteria to the *Nodematch* function, as explained below.

FindPartialMatches At the top level of the partial match finding algorithm, there are three nested loops. The outer loop iterates over all the subject operation nodes N_{sub} of the subject graph G_{sub}, the loop nested within iterates over all the pattern graphs G_{pat} in the pattern library *Patlib*, and the inner loop iterates over all pattern operation nodes N_{pat} of G_{pat}. Note that any of the loops may iterate over their domain in any particular order; one of the properties of the partial match finding algorithm is that topological order is irrelevant. Each pair of nodes (N_{sub}, N_{pat}) is compared, and if they match, a new *Match* is created for the pair and their respective operand nodes.

A pseudo-code implementation of *FindPartialMatches* is given below.

```
FindPartialMatches()
{
  foreach subject operation node Nsub of Gsub
    foreach pattern Gpat in pattern library Patlib
      foreach pattern operation node Npat of Gpat
        if Nodematch(Nsub, Npat)
          // create a new match
          new Match(Nsub, Npat);
}
```

Nodematch The function *Nodematch* compares a subject operation node N_{sub} and a pattern operation node N_{pat} and returns whether *all* of the following criteria have been met:

- The opcodes of the operations are the same.
- If any input pattern operand O_{pat} is an immediate, then the corresponding subject operand O_{sub} should be an immediate with the same value.
- If any input pattern operand O_{pat} is connected to more than one input of N_{pat}, then the corresponding subject operand O_{sub} must be connected to the same set of inputs on N_{sub}.
- For each pair of corresponding results O_{sub} and O_{pat}:
  - The number of uses of O_{sub} is at least as high as the number of uses of O_{pat}.
  - If the number of uses of O_{sub} is greater than the number of uses of O_{pat}, then O_pat should be a pattern output, so that after substitution of the pattern into the trace (during covering), the value of O_{sub} is still available to the uses not covered by the pattern.
  - If O_{sub} is the current (valid) definition of its register (section 4.2), meaning that new uses could be added later on during the trace simulation, then O_{pat} must be an output of the pattern, so that after substituting the pattern into the trace, the value of O_{sub} is still available.

```
Nodematch(subject operation node Nsub,
          pattern operation node Npat)
{
  // check opcodes
  if(Nsub.opcode != Npat.opcode)
    return FALSE;

  // check input operand nodes
  foreach input operand Osub of Nsub/Opat of Npat
   {
    if(Opat.type == IMMEDIATE)
     {
      if(Osub.type != IMMEDIATE)
        return FALSE;
      if(Osub.imm_val != Opat.imm_val)
        return FALSE;
     }

    // check if shared pattern operands correspond
    // to shared subject operands
    if(Opat_checklist.Contains(Opat))
     {
      idx = Opat_checklist.GetIndex(Opat);
      if(Osub_checklist[idx] != Osub)
        return FALSE;
     }
    else
     {
      Opat_checklist.Append(Opat);
      Osub_checklist.Append(Osub);
     }
   }

  // check result operand nodes
  foreach result operand Osub of Nsub/Opat of Npat
   {
    if(Osub.num_uses < Opat.num_uses)
      return FALSE;
    if((Osub.num_uses > Opat.num_uses || Osub.is_current_def)
       && !Opat.IsPatternOutput())
      return FALSE;
   }

  return TRUE;
}
```

The abovementioned criteria are necessary for correctness. At some computational expense, additional criteria may be specified to reduce the number of partial matches that will be created because *Nodematch* is true, but which can never be merged into full matches. For example:

- If any of the inputs O_{pat} of N_{pat} has its defining operation node N_{pat_pre} within the pattern graph, then:
  - The corresponding O_{sub} should have a definer N_{sub_pre}.

– $Nodematch(N_{sub\_pre}, N_{pat\_pre})$ is true.

Match constructor Whenever a match between a subject operation node N_{sub} and a pattern operation node N_{pat} has been found, a new match must be constructed. This is done by looking up the reference vector indices for all operand nodes O_{pat} connected to N_{pat}, then adding references to the corresponding subject operand nodes O_{sub} at the appropriate index. The reciprocal reference is set on O_{sub} by adding the ($Match$, index) pair to its list of match references. A reference to the parent graph G_{pat} of N_{pat} is then added to the $Match$.

```
Match constructor(subject operation node Nsub,
                  pattern operation node Npat)
{
  // add input and output operand references
  foreach operand node Opat of Npat/Osub of Nsub
   {
    index = Opat.index;
    Match.ref[index] = Osub;
    Osub.AddMatchRef(Match, index);
   }

  Match.patterngraph = Npat.parentgraph;
}
```

5.2 Merging partial matches

After the partial match finding phase, we must now try to construct complete matches out of the partial ones that have just been generated. Looking at all the operand nodes O_{sub} of the subject graph G_{sub} (in no particular order), we look at all the matches found on O_{sub} and see which of these matches can be merged into new, higher degree partial or even complete matches.

MergeMatches The function $MergeMatches$ looks at the matches found on each subject operand node O_{sub} and attempts to merge them with each other into new matches by calling the function $Merge$ on them.

Note that the outer match loop (foreach M_1) only considers the matches already on O_{sub} before any new, higher degree matches were added. The inner loop (foreach M_2), however, considers all matches, including new ones, from M_1 onwards. This is done for efficiency reasons; there are several ways to arrive at each higher degree match, but only one is needed.

```
MergeMatches()
{
 foreach subject operand node Osub of Gsub
  {
   // iterate only over matches already present
   foreach match M1 on Osub
    {
     // iterate including new matches from previous iterations
     foreach match M2 on Osub from M1 onwards
       Merge(Osub, M1, M2);
    }
  }
}
```

Merge Two matches M_1 and M_2 may be merged on operand node O_{sub} if they meet the following criteria:

- They refer to the same pattern graph.
- They both have the reference to O_{sub} in the same position of their reference vector.
- For each position of their reference vectors, if both M_1 and M_2 have a reference to a subject operand node in that position, those must refer to the same operand node. Otherwise the matches are said to *clash* at that position.

While the abovementioned criteria must be met for correctness, the following ensure that no redundant matches (matches equal to one already on O_{sub}) are created:

- M_1 is not a subset of M_2, that is, merging M_1 and M_2 would not yield match equivalent to M_2.
- M_2 is not a subset of M_1.
- There is no match already on O_{sub} that is equivalent to the merger of M_1 and M_2[2].

If all criteria are met, a new match $M3$ can be created and a reference is appended to the match list on O_{sub}. Note that implementation shown below is slightly different.

The function *Merge* combines the merging of two matches M_1 and M_2 with checking for subset and clash conditions, for efficiency reasons: we now have to iterate along the reference vectors of the matches only once instead of four times (if we were to check each condition separately).

The function iterates over the reference vectors of M_1 and M_2 and finds which subject operand nodes O_{sub1} and O_{sub2} are referenced. For every pair of operand references, one of four situations may arise:

- M_1 references an operand node but M_2 references nothing. We now know that M_1 cannot be a subset of M_2.
- M_1 and M_2 both reference an operand node. If those are different operand nodes, there is a clash: the merge is illegal.
- M_1 references nothing but M_2 references an operand node. We now know that M_2 cannot be a subset of M_1.
- M_1 and M_2 both reference nothing.

If there is no clash, we put the reference in the current position of the reference vector of the new match $M3$. After all references of M_1 and M_2 have been processed this way, we can check for the subset conditions. If either is still true, we reject the new match.

A redundancy check with all matches already present on O_{sub} is performed afterwards by calling the function *Redundant* with the newly formed match. The *Commit* function appends references to the new match to the appropriate subject operand nodes.

[2]This criterion encapsulates the previous two but requires more computation time to evaluate

```
// merge the two matches, if possible. Return FAILED otherwise
match Merge(operand Osub, match M1, match M2)
{
  // Check if pattern graphs are the same
  if(M1.pattern != M2.pattern)
    return FAILED;

  // Check if position in the matches is the same
  index = position of Osub in M1;
  if(M2[index] != Osub)
    return FAILED;

  // initialization
  M1_is_subset_of_M2 = TRUE;
  M2_is_subset_of_M1 = TRUE;

  // iterate along operand reference vectors
  foreach index in M1 // index for vector
   {
    Osub1 = M1[index];
    Osub2 = M2[index];
    if(Osub1 != NULL)
     {
      if(Osub2 == NULL)
        M1_is_subset_of_M2 = FALSE;
      else
        if(Osub2 != Osub1)  // clashing operands
          return FAILED;
      M3[index] = Osub1;
     }
    else // Osub1 == NULL
     {
      if(Osub2 != NULL)
        M2_is_subset_of_M1 = FALSE;
      M3[index] = Osub2;
     }
   }

  if(M1_is_subset_of_M2 || M2_is_subset_of_M1)
    return FAILED; // subset, new match would be redundant

  // Check for redundancy with all other matches on Osub
  if( Redundant(Osub, M3) )
   return FAILED;

  // Set references from operands to the new match
  Commit(M3);
  return SUCCEEDED;
}
```

Redundant checks all higher degree matches M_2 on O_{sub} and looks for one equivalent to M_1. There are three conditions that have to be met for M_1 to be equivalent to M_2:

- The patterns referenced by M_1 and M_2 are the same.
- The degree of M_1 must be equal to the degree of M_2.
- $(M_1[index] == M_2[index])$ must hold for all values of *index*.

As soon as redundancy is detected, *Redundant* returns TRUE, since there is no need to look at further matches.

```
// Return TRUE if the operand node already
// contains a match just like this.
Redundant(Operand Osub, Match M1)
{
  foreach match M2 on Osub except M1
   {
    // must reference the same pattern
    if(M1.pattern != M2.pattern)
      continue;

    // degree must be the same
    if(M1.degree != M2.degree)
      continue;

    is_redundant = TRUE;
    foreach index in M1 // index for operand reference vector
     {
      Osub1 = M1[index];
      Osub2 = M2[index];
      if(Osub1 != Osub2)
       {
        is_redundant = FALSE;
        break;
       }
     }
    if(is_redundant)
      return TRUE;
   }
  return FALSE;
}
```

Commit The match M is not redundant with any of the matches already on O_{sub}, so we must add reciprocal references on the subject operand nodes it references.

```
// Add references to each of the operands
// the match reference vector contains.
Commit(Match M)
{
  foreach index in M // index for reference vector
   {
    Osub = M[index];
    if(Osub == NULL)
      continue;
    Osub->AddMatchRef(M, index);
   }
}
```

5.3 Cleaning up the partial matches

The final phase of the matching algorithm eliminates all incomplete matches, so that only full matches remain on all the subject graph nodes. This is done by comparing the degree of all matches to the number of operation nodes in the corresponding pattern graphs. If those are not equal, then the match is incomplete and must be deleted.

Another condition under which a match is deleted is that it would introduce a dependency cycle into the subject graph. This is the case if any of the matched pattern's outputs lead to any of its inputs, through some path in the subject graph.

```
// Remove all incomplete matches
CleanupMatches()
{
  foreach subject operand node Osub of subject graph Gsub
    foreach match M1 on Osub
      if(M1.degree != M1.patterngraph.num_operations
         || M1.CausesCycle())
        delete M1;
}
```

5.4 Integrated match creation and merging

In the previous subsections, the first and second phases of the matching algortihm, first degree match construction and match merging, have been described separately. However, it is possible to perform both tasks in a single iteration over the subject graph's operation nodes. This subsection describes the function $FindPartialAndMerge$, which is an implementation of integrated partial match finding and merging. In the top-level function $Matching$, it replaces $FindPartialMatches$ and $MergeMatches$. Many of the function called in $FindPartialAndMerge$ are described in the previous subsections.

FindPartialAndMerge

```
FindPartialAndMerge()
{
  foreach subject operation node Nsub
    foreach pattern graph Gpat
      foreach pattern operation node Npat
        FindPartialAndMerge(Nsub, Npat);
}

FindPartialAndMerge(subject operation node Nsub,
                    pattern operation node Npat)
{
  if(Nodematch(Nsub, Npat))
   {
    // first degree match construction
    M = new Match(Nsub, Npat);

    // match merging
    foreach input and output operand Osub of Nsub
     {
      // matches previously on this operand
      foreach match M1 on Osub up to M
```

```
        {
         // including newly added matches from
         // previous iterations of both M2 and M1
         foreach match M2 on Osub from M onwards
           Merge(Osub, M1, M2);
        }
      }
    }
}
```

5.5 Automatic library construction

The matching and covering algorithms described in the previous section operate using a predefined pattern library. In this section, we will look at an algorithm for the automatic detection and construction of patterns. Hooking into the matching algorithm's function $FindPartialAndMerge$, this algorithm operates by constructing new patterns out of previously detected (and matched) ones. In the following paragraphs, the function hierarchy of the automatic library construction algorithm will be described, starting with a modified version of $FindPartialAndMerge$.

FindPartialAndMerge This function is essentially the same as the one described in section 5.4, with one additional function call. Each time the partial matches on a subject operation node N_{sub} have been merged, the function $ConstructNewPatterns$ is called, as shown below.

```
FindPartialAndMerge()
{
  foreach subject operation node Nsub
    foreach pattern graph Gpat
      foreach pattern operation node Npat
        FindPartialAndMerge(Nsub, Npat);
}

FindPartialAndMerge(subject operation node Nsub,
                    pattern operation node Npat)
{
  if(Nodematch(Nsub, Npat))
   {
    // first degree match construction
    M = new Match(Nsub, Npat);

    // match merging
    foreach input and output operand Osub of Nsub
     {
      // matches previously on this operand
      foreach match M1 on Osub up to M
       {
        // including newly added matches from
        // previous iterations of both M2 and M1
        foreach match M2 on Osub from M onwards
          Merge(Osub, M1, M2);
       }
     }
```

```
    // pattern construction
    ConstructNewPatterns(Nsub);
   }
}
```

ConstructNewPatterns This function considers all complete matches on the current subject operand O_{sub} and attempts to construct new matches by taking the pairwise union of those matches ($MatchUnion$). If this fails, we continue with the next pair of matches. If it succeeds, then the result of the union corresponds to a match with the new pattern we are trying to construct (an example of this is given in section 3.2).

We must now check if there already is a complete match with exactly the same operation and operand references ($RedundantMatch$). If this is the case, we know that the pattern corresponding to that match is equivalent to the one we are trying to construct. The new pattern would therefore be redundant, which means that we want to forget about constructing it and delete the match M_3.

If the aforementioned checks hold, then the new match M_3 represents a new, unique pattern. We can now construct the corresponding pattern graph ($ConstructPattern$) and construct partial matches for M_3, for use in later merging operations ($ConstructPartialMatches$).

```
ConstructNewPatterns(subject operation node Nsub)
{
  foreach input Osub of Nsub
    foreach complete match M1 on Osub
     {
      foreach complete match M2 on Osub from M1 onwards
       {  // excluding not matches added during this iteration of M1

        // try to take union
        M3 = MatchUnion(M1, M2);

        // check if MatchUnion succeeded
        if(M3 == NULL)
          continue;

        // check if resulting pattern already exists
        if(!RedundantMatch(Osub, M3))
         {
          // Build the new pattern and add to library
          ConstructPattern(M3);

          // Make partial matches from match M3
          ConstructPartialMatches(Nsub, M3.patterngraph);
         }
       }
     }
}
```

MatchUnion This function attempts to take the union of two matches M_1 and M_2 by taking the unions of their subject operation and operand reference vectors, and assigning them to the new match M_3.

Taking the union fails if M_1 is a subset of M_2 (or vice versa), in which case the resulting match would be redundant with M_1 (or M_2, respectively).

The exception to this rule is when we try to take the union of a match with itself. This only happens if the match was referenced on the same operand node more than once, which in turn only happens if that operand node was referenced more than once by the match (i.e. used on more than one input of the match). This implies that a new pattern is possible, in which the multiplicated operand is replaced by a single operand. This special case is handled separately, as can be seen in the pseudo-code implementation of $MatchUnion$, below.

For efficiency reasons, taking the union of the operation reference vectors of M_1 and M_2 and checking the subset conditions are integrated. First, all subject operation references of M_1 are put into the subject operation reference vector of the new match M_3. If all the operation references that were added from M_1 were also contained in M_2, then M_1 is a subset of M_2 and $MatchUnion$ fails. After this, the subject operation references of M_2 are added to M_3, if they weren't already contained in M_3. If no new references were added, then M_2 is a subset of M_1 and $MatchUnion$ also fails.

If taking the union of the operation reference vectors succeeds, then the size constraints must be checked. If the number of operation references in the new match is greater than a predefined maximum number MAX_OPS_IN_PATTERN, then the new pattern would become unacceptably large: $MatchUnion$ fails.

If the size constraints are met, the union of the subject operand reference vectors of M_1 and M_2 can be computed. It is no longer necessary to check for subset conditions, since those would already have been detected when the operation reference vector union was computed. All references are 'uniquified': any operand node is referenced only once, even though it may have appeared more than once in the reference vectors of M_1 or M_2. Note that, if desired, a size constraint on the number of operand references could be added after this.

```
match MatchUnion(match M1, match M2)
{
  // make empty match
  M3 = new match();

  if( M1 == M2 )  // merging with self?
   {
    M3.operationrefs = M1.operationsrefs;
   }
  else
   {
    M1_subset_M2 = TRUE;
    M2_subset_M1 = TRUE;

    // add operation references from M1
    foreach Nsub in M1.operationrefs
     {
      M3.operationrefs.add(Nsub);
      if(!M2.operationrefs.contains(Nsub))
        M1_subset_M2 = FALSE;
     }

    if(M1_subset_M2 == TRUE)
      goto fail:

    // add operation references from M2
    foreach Nsub in M2.operationrefs
```

```
      {
       if(!M3.operationrefs.contains(Nsub))
        {
         M3.operationrefs.add(Nsub);
         M2_subset_M1 = FALSE;
        }
      }

    if(M2_subset_M1 == TRUE)
      goto fail:

    // check size constraints
    if(M3.operationrefs.count() > MAX_OPS_IN_PATTERN)
      goto fail:

    // add operand references from M1
    foreach Osub in M1.operandrefs
      if(!M3.operandrefs.contains(Osub))
        M3.operandrefs.add(Osub);
  }

  // add operand references from M2
  foreach Osub in M2.operandrefs
    if(!M3.operandrefs.contains(Osub))
      M3.operandrefs.add(Osub);

  return M3;

fail:
  delete M3;
  return NULL;
}
```

RedundantMatch The function *RedundantMatch* differs from the function *Redundant* in that the latter compares matches that are known to refer to the same pattern graph. *RedundantMatch* compares a match M for which there is no pattern graph yet with all complete matches on O_{sub} in order to determine whether an equivalent pattern graph already exists. If there is a match M_1 on O_{sub} such that all of the following criteria are met, then the pattern graph corresponding to M is equivalent to the pattern graph corresponding to M_1:

1. The subject operation reference vectors of M and M_1 are of equal length. This corresponds with the pattern graphs having an equal number of operations.
2. The subject operand reference vectors of M and M_1 are of equal length. This corresponds with the pattern graphs having an equal number of operands.
3. All operations referenced in M are also referenced in M_1. This corresponds to the pattern graphs containing the same operations.
4. All operands referenced in M are also referenced in M_1. This corresponds to the pattern graphs containing the same operands.
5. All operands mapping to a pattern output in the pattern graph corresponding to M_1 would also map to an output of the pattern graph corresponding to M.

The criteria are listed by order of the computational effort it requires to evaluate them. The further down the list, the more stringent the criteria become. Pattern graphs failing only the final criterion are isomorphic, except that they have different outputs: one or more values are used only internally in one of the patterns, while being made available externally (as outputs) in the other.

```
RedundantMatch(subject operand Osub, match M)
{
  // iterate over all complete matches
  foreach complete match M1 on Osub
   {
    // 1. Compare operation reference vector lengths
    if(M1.operationrefs.count() != M.operationrefs.count())
      goto next:

    // 2. Compare operand reference vector lengths
    if(M1.operandrefs.count() != M.operandrefs.count())
      goto next:

    // 3. Compare operation reference sets
    foreach Nsub in M1.operationrefs
      if(!M.operationrefs.contains(Nsub))
        goto next:

    // 4. Compare operand reference sets
    foreach Osub in M1.operandrefs
      if(!M.operandrefs.contains(Osub))
        goto next:

    // 5. Check pattern outputs
    foreach Osub in M.operandrefs
     {
      // determine if this would be a pattern output
      is_output = FALSE;
      foreach Nsub in Osub.uses
       if(!M.operationrefs.contains(Nsub))
        {
         is_output = TRUE;
         break;
        }

      // find corresponding pattern operand via M1
      Opat = M1.patterngraph.operands[Osub.matchref(M1).index];
      // if Osub is a pattern output in M, but not in M1
      // or vice versa
      if(Opat.IsOutput() != is_output)
        goto next:
     }

   // M1 and M are equivalent
   return TRUE;

next:
   }
```

```
  // all matches checked, no equivalent found
  return FALSE;
}
```

ConstructPattern Constructing the pattern graph corresponding to the match M is fairly straightforward. First, pattern operand nodes are created for each referenced operand node, and added to the pattern graph. If the subject operation defining the current subject operand is not referenced in M (i.e. lies outside the pattern), then the operand maps to a pattern input, which is marked as such. If there is any subject operation node that uses the operand but is not referenced in M, then the operand maps to a pattern output.

For each subject operation node referenced in the match, a pattern operation node is created and added to the graph. The operation's inputs and outputs are hooked up to it by adding def- and use-edges to the operands. The new graph is then added to the pattern library *PatLib*.

```
ConstructPattern(match M)
{
  Gpat = new pattern graph;

  // add pattern operand nodes
  for i = 1 to M.operandrefs.count()
   {
    Osub = M.operandrefs[i];
    Opat = new pattern operand("O{i}");
    Gpat.AddOperand(Opat);

    // determine if Opat is a pattern input
    Nsub = Osub.Def();
    if(Nsub != NULL)
      if(!M.operationrefs.contains(Nsub))
        Gpat.AddInput(Opat);

    // determine if Opat is a pattern output
    is_output = FALSE;
    foreach Nsub in Osub.uses
     if(!M.operationrefs.contains(Nsub))
      {
       is_output = TRUE;
       break;
      }
    if(is_output == TRUE)
      Gpat.AddOutput(Opat);
   }

  // add operation nodes
  foreach Nsub in M.operationrefs
   {
    Npat = new pattern operation(Nsub.opcode);
    Gpat.AddOperation(Npat);

    // hook up inputs...
    for i = 1 to #input operands of Nsub
     {
```

```
      Osub = Nsub.inputs[i];
      Opat = Gpat.operands[Osub.matchref(M).index];
      Npat.operand[i] = Opat;
      Opat.AddUse(Npat);
     }

    // ...and output
    Osub = Nsub.output;
    Opat =  Gpat.operands[Osub.matchref(M).index];
    Opat.AddDef(Npat);
   }


  // add to library
  Patlib.AddPattern(Gpat);
}
```

ConstructPartialMatches This function constructs partial matches for the newly created pattern graph on all nodes of the subject graph that have already been processed (i.e. preceding N_{sub_end}). This is necessary to keep the partial matches consistent throughout the graph. The partial matches thus created may be merged into complete matches later (when processing nodes succeeding N_{sub_end}). If we neglected to create the partial matches, then instead of detecting a match with pattern G_{pat}, we would find a new pattern (equivalent to G_{pat} of course, but that fact would remain undetected).

```
ConstructPartialMatches(subject operation Nsub_end,
                        pattern Gpat)
{
  foreach operation Nsub in Gsub up to Nsub_end
    foreach operation Npat of Gpat
     {
      FindPartialAndMerge(Nsub, Npat);
     }
}
```

IsCompletable If the defining operation for a subject operand node is not present in the partial match, then that match can never become a complete match anymore. The operand node can never get a new definer in the subject graph, hence no new subject operation reference can be put into the partial match.

The function iterates over all pattern operation nodes of G_{pat}, the pattern graph corresponding to M. If there is a corresponding subject operation referenced in M, we want to check its input operands. If those correspond to pattern inputs, we cannot check anything. If they should have a defining operation referenced in M, but there is none, then M can never be completed and *IsCompletable* returns FALSE.

```
IsCompletable(match M)
{
  Gpat = M.patterngraph;
  foreach Npat of Gpat
   {
    // skip empty reference entries
    if(M.operationrefs[Npat.index] == NULL)
```

```
      continue;

    // check inputs for definers
    foreach input Opat of Npat
     {
      Osub = M.operandrefs[Opat.index];
      assert(Osub != NULL);  // sanity check

      // no definer in graph? (== graph input)
      if(Gpat.inputs.contains(Opat))
        continue;

      // check if Osub's definer is referenced in M
      Npat_pre = Opat.Def();
      Nsub_pre = M.operationrefs[Npat_pre.index];
      if(Nsub_pre == NULL)
        return FALSE;
     }
   }
  return TRUE;
}
```

6 Covering

After all possible matches between pattern graphs and the subject graph have been found by the matching algorithm (as described in section 5), we can now try to find a selection of matches to implement the subject graph with. This selection is called a *cover* of the subject graph, it must be chosen so that each node of the subject graph is covered by exactly one match. There are several possible covers, we are interested in the ones that minimize the latency of the longest path in subject graph.

The algorithm that is used for covering (finding a cover) is based on dynamic programming. It consists of the following phases:

1. **Evaluating the matches:** for each operand node in the subject graph, determine the data-ready time for all output matches that apply to it. Sort those matches by data-ready time, ascending.

2. **Sorting the graph outputs:** sort all subject graph output operands (operands that have no uses in the subject graph) by order of data-ready time of their best match, descending.

3. **Choosing a cover:** Starting at the subject graph output operand with the highest data-ready time for its best match, implement the match with the lowest possible data-ready time, recursively (depth-first). Mark the obtained data-ready times on all operand nodes. Terminate at subject graph inputs or already covered operand nodes.

```
Cover()
{
  EvaluateMatches();
  SortGraphOutputs();
  CoverOutputs();
}
```

The first phase, evaluating the matches, is described in section 6.1. The second phase is just a sorting operation and will not be described in detail. The third phase, choosing a cover from the available matches, is described in section 6.2.

6.1 Evaluating the matches

The matches on each subject graph operand have to be evaluated in terms of the data-ready time they would give the operand, when implemented. This is done by a hierarchy of functions, of which $EvaluateMatches$ is the top one.

EvaluateMatches This function iterates over all subject graph operands, in no particular order, and evaluates the matches on each subject operand node. The reason that no particular order is required is that the recursive function $EvaluateMatches$ stores its results on the subject operand node $Osub$. Therefore, if during any iteration the same node is visited more than once, the function simply returns the results that were calculated on the first visit. This eliminates the need for a topological sort.

```
EvaluateMatches()
{
  foreach subject operand node Osub
    TimeOutputMatches(Osub);
}
```

TimeOutputMatches The purpose of this function is to determine the data-ready time for operand O_{sub} that each of its matches would give it, when implemented. The matches are sorted by this quantity.

First we check if the subject graph operand node O_{sub} is an input of the subject graph, that is, if it has no defining operations in the graph. If this is the case, we return the node's data-ready time, as calculated during a previous covering iteration (or zero if the current covering iteration is the first).

Then we check if this node O_{sub} has been visited in a previous iteration. This is done by checking for entries in the output match reference list. If there are none, we obviously haven't called $FindOutputMatches$ yet, so we must execute the rest of the function. If there are, we simply return the previously calculated best data-ready time for output matches on O_{sub}.

We then determine the data-ready time for each of the output matches. This is done by taking the maximum of the latency of the pattern from any of the pattern inputs to the current operand, added to the data-ready time of the input. After all output matches have been timed, they are sorted according to data-ready time, ascending.

```
int TimeOutputMatches(subject operand node Osub)
{
  // Check if Osub is a subject graph input
  if(Osub.IsSubjectGraphInput())
    return Osub.data_ready_time;

  // Check if we have already visited this Osub.
  Mref = first output match reference of Osub;
  if(Mref != NULL)
    return Mref.data_ready_time;

  FindOutputMatches(Osub);
```

```
  max_data_ready = 0;

  // find data-ready time for all output matches
  foreach output match reference Mref on Osub
   {
    match M = Mref.match;
    index = Mref.index;
    Gpat = M.patterngraph;
    Opat_out = Gpat.operands[index];

    // time for each match input
    foreach pattern input operand Opat_in of M
     {
      pat_latency = Gpat.latency(Opat_in, Opat_out);
      input_index = Opat_in.index;
      Osub_in = M[input_index];
      data_ready = TimeOutputMatches(Osub_in) + pat_latency;
      max_data_ready = Max(max_data_ready, data_ready);
     }

    Mref.data_ready_time = max_data_ready;
   }

  // sort output match references according to data-ready time
  foreach output match reference Mref1 on Osub
    foreach output match reference Mref2
                  on Osub from Mref1 onwards
      if(Mref1.data_ready_time > Mref2.data_ready_time)
        Swap(Mref1, Mref2);
}
```

FindOutputMatches determines which matches on the operand are so-called output matches (see also section 3.3). This is fairly straightforward: the reference to the match can be used to find the pattern operand node that the current subject operand node corresponds to. If this pattern operand is a pattern output, then the match is an output match to the current subject operand.

```
FindOutputMatches(subject operand node Osub)
{
  foreach match reference Mref on Osub
   {
    match M = Mref.match;
    index = Mref.index;
    Gpat = M.patterngraph;
    Opat = Gpat.operands[index];
    if(Opat.IsOutputNode)
      Osub.AddOutputMatchRef(Mref)
   }
}
```

6.2 Choosing a cover

After all output matches on all subject graph operand nodes have been evaluated and sorted in terms of best attainable data-ready time, the time has now come to make a selection from those output matches: the cover. This is done by calling the top-level function *CoverOutputs*.

CoverOutputs This function iterates over the subject graph output operands, which have been sorted by projected data-ready time, descending, in the previous phase. It calls the recursive function *ImplementBestMatch* for each of them.

```
CoverOutputs()
{
  foreach subject graph output operand Osub (sorted)
   ImplementBestMatch(Osub);
}
```

ImplementBestMatch First we check if the current node has already been covered as part of a earlier covering iteration. If this is the case, we return the attained data-ready time from that cover.

The best available matches on the subject sub-graph rooted in node O_{sub} are implemented, recursively. The best available match is the first one on O_{sub} that would not introduce any dependency cycles into the subject graph.

To guarantee that the critical path to the current subject operand node is covered first, we have to determine the precedence with which the subject operand nodes corresponding to the match inputs are covered. This is done by calculating the *slack* for each one of these nodes. The slack is defined as the difference between the time a match input should be ready (the current match output's data-ready time minus the pattern latency) and the data-ready time of the input's best match. The match inputs are sorted by slack, ascending, before covering them in that order.

The attained data-ready time is calculated and marked on the subject operand nodes. All nodes covered by the selected matches are marked as having been covered by the function *SelectMatch*.

```
int ImplementBestMatch(subject operand node Osub)
{
  index = Mref.index;
  Gpat = M.patterngraph;
  Opat_out = Gpat.operands[index];

  // check if we've already covered this node
  if(Osub.has_been_covered == TRUE)
    return Osub.data_ready_time;

  // find the best match that can be implemented
  foreach output match reference Mref of Osub
    if(Mref.match.IsImplementable())
      break;
  M = Mref.match;

  SelectMatch(M);

  // determine precedence
  output_time = TimeOutputMatches(Osub);
```

```
  foreach pattern input operand Opat_in of M
   {
    Osub_in = subject node corresponding to Opat_in;
    pat_latency = Gpat.latency(Opat_in, Opat_out);
    input_time = TimeOutputMatches(Osub_in);
    Osub_in.slack = output_time - pat_latency - input_time;
   }

  sort pattern input operands Osub_in by slack, ascending;

  // calculate ready-time
  foreach pattern input operand Osub_in, sorted
   {
    Opat_in = pattern node corresponding to Osub_in;
    pat_latency = Gpat.latency(Opat_in, Opat_out);
    data_ready = ImplementBestMatch(Osub_in) + pat_latency;
    max_data_ready = Max(max_data_ready, data_ready);
   }

  Osub.data_ready_time = max_data_ready;
  return max_data_ready;
}
```

SelectMatch The match selected for implementation probably overlaps several other matches. These can never be implemented anymore, so they must be deleted. This function removes all other matches that are referenced on the operand nodes in the selected match.

Note that any of the removed matches may have been the best match for another operand node. This means that the best implementation, the one that match evaluations for other operands are based on, is no longer available there. The impact of removing any match may resonate throughout the subject graph. Keeping track of these effects is not trivial and would require a lot of extra computational effort (each time a match is implemented, we have to re-evaluate all matches on all nodes). We therefore settle for staying with our old match evaluation results and implement the next best available match, as determined previously, everywhere.

```
SelectMatch(match M)
{
  foreach index in M // index for operand reference vector
   {
    Osub = M[index];
    Osub.has_been_covered = TRUE;

    foreach match M_not_taken on Osub
     {
      // delete matches that overlap with M
      if(M_not_taken != M)
        delete M;
     }
   }
}
```

7 Conclusions

In this report, new matching and covering strategies were presented. The matching algorithm can handle pattern graphs of arbitrary topology, as long as each pattern's operation and operand nodes are all connected in some way (i.e. SIMD-style patterns are not detected).

The covering algorithm is based on dynamic programming. When there are only tree-shaped patterns in the pattern library and the subject graph is a tree as well, it behaves exactly like the dynamic programming algorithm and therefore produces optimal results. If more complex patterns are used, the covering algorithm is heuristic and optimality cannot be guaranteed.

References

[1] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *DAC, Proceedings of the Design Automation Conference*, pages 617–623, May 1987.

[2] Yuji Kukimoto, Robert K. Brayton, and Prashant Sawkar. Delay-optimal technology mapping by dag covering. In *Proceedings of the Design Automation Conference*, 1998.

[3] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGRAW-HILL, 1994.

[4] R. Rudell. Logic synthesis for vlsi design. Technical Report UCB/ERL M89/49, University of California at Berkeley, April 1989.