SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm

Miles Ohlrich, Carl Ebeling, Eka Ginting†, and Lisa Sather

Computer Science & Engineering Department
University of Washington
Seattle, WA 98145

Abstract – The problem of finding subcircuits in a larger circuit arises in many contexts in computer-aided design. This is a problem currently solved using ad hoc techniques that rely on the circuit technology and implementation details. For example, channel graphs and signal flow are often used to extract simple gates from a transistor layout. Such techniques, however, do not generalize to different subcircuit structures and do not transfer to other technologies. We present a technology-independent algorithm for this problem based on a solution to the subgraph isomorphism problem. Although this problem is known to be NP-complete, our solution is very fast in practice for real circuits. We describe the algorithm, which uses an extension of the graph partitioning algorithm used by Gemini[3] for graph isomorphism, and present experimental results that show that the typical running time for large CMOS circuits is approximately linear in the total number of devices within the subcircuits being matched.

I. Introduction

The subgraph isomorphism problem appears in many contexts related to circuit design. Perhaps the most common instance is identifying a related collection of interconnected primitive devices in a circuit as a single high-level component. For example, converting a transistor netlist into a gate netlist involves finding the subcircuits representing gates and replacing them with the corresponding gates. This is a very important task and many specialized algorithms have been devised to perform it [1, 5, 7]. However, these rely on the specific characteristics of the technology or circuits being transformed and are not easily applied to different technologies or circuit types such as analog circuits. Moreover, these techniques rely on assumptions about the subcircuits being extracted and do not generalize to allow arbitrary subcircuits to be found.

We present a solution to the subgraph isomorphism problem as a general technique that is applicable across technologies and circuit domains. By avoiding use of any specific knowledge about the underlying technology or semantics of the target circuits, the algorithm is truly technology-independent — any circuit which can be described as a set of interconnected devices can be handled by our algorithm. Although subgraph isomorphism is known to be NP-complete and therefore intractable in general, circuits have sufficient structure to allow an almost linear time solution in practice. On the surface, this algorithm is similar to the graph isomorphism algorithm used by programs such as Gemini[3]. However, the labeling procedure used for subgraph isomorphism is necessarily very different from that used for graph isomorphism for if labels are not computed subject to strict constraints, they become meaningless. On the other hand, if too many constraints are placed on the labels then they contain too little information.

Many other problems can make use of a solution to the subgraph isomorphism problem. Finding circuit subgraphs plays a key role in constructing a hierarchical representation of a circuit from a flat representation[6]. Matching circuits hierarchically simplifies the problem of identifying the precise location of errors and also allows one to efficiently check incremental changes.

One may also use subgraph isomorphism to automatically review circuits for the use of questionable circuit constructs. Such rule checks currently use programs with built-in knowledge of these constructs. A general algorithm, however, allows these constructs to be described as circuits in a library which can be easily extended as necessary.

Another application arises in the area of technology mapping, which covers a circuit graph with components from a library. Current techniques rely on tree-covering algorithms, which require that both the input circuit and library components be represented as trees. A general subgraph isomorphism algorithm would allow one to find all possible coverings for general component graphs, including those with feedback and reconvergent fanout.

We have implemented our subgraph isomorphism algorithm in a program called *SubGemini*. The SubGemini algorithm works in two phases: In Phase I, SubGemini identifies all possible locations of the subcircuit in the main circuit. It does this by applying a partitioning algorithm to both circuits in order to choose a key vertex, $\mathcal{K}$, in the subcircuit and identify all *possible* vertices in the main circuit graph that might match the key vertex. This set of vertices is called the *candidate vector*, $\mathcal{CV}$. Phase I acts as a filter that tries to reduce the number of instances that need to be checked; Later, Phase II will check each instance in order to determine if it is part of a subcircuit. In the best case, the candidate vector includes only true instances of the subcircuit, but Phase I is guaranteed to find all possible candidates. Phase I is discussed in detail in section III.

In Phase II, SubGemini verifies whether there is an actual subcircuit at each location indicated by the candidate vector. It examines each vertex c in the candidate vector and attempts to find a mapping between vertices in the subcircuit graph and vertices in the main circuit graph, such that $\mathcal{K}$ matches c. This is done by initially postulating a match between $\mathcal{K}$ and c, labeling the two vertices with a unique label. Starting from this first label, the algorithm simultaneously labels both the main circuit and the subcircuit such that labels of vertices match if and only if there is a valid mapping between the two graphs. If this procedure finds matching labels in the main circuit for all the vertices in the subcircuit, then a subcircuit has been found. Otherwise, the candidate vertex was a false candidate. Phase II is discussed in detail in section IV.

We first introduce some terminology and definitions that will be used throughout the paper. Then we discuss the actual implementation of the algorithm. Then we discuss several optimizations and special cases that arise in circuits and how these may be handled. Finally, we give the results for SubGemini for a variety of CMOS circuit examples.

This research was funded in part by the Defense Advanced Research Projects Agency under Contract N00014-J-91-4041. Carl Ebeling was supported in part by NSF Presidential Young Investigator Awards, with matching funds provided by IBM Corporation and Sun Microsystems. Miles Ohlrich was supported by a National Science Foundation Graduate Fellowship.

† Now at Carnegie-Mellon University.

II. Preliminaries

SubGemini uses the same circuit graph model and similar labeling procedure as that described in [3, 4] for Gemini. We review these briefly before describing the SubGemini algorithm. SubGemini represents circuits as graphs as shown in Fig. 1 and Fig. 2. A circuit graph is an undirected bipartite graph with devices, represented by squares, forming one set of vertices, and nets (wires), represented by circles, forming another set. A connection between two devices is made by connecting each device vertex in the circuit's graph representation to a net vertex. Explicitly representing nets as vertices in the graph reduces the number of edges required to represent the full interconnection of N device terminals from $N(N-1)/2$ to N. Additionally, it exposes the structure of the circuit to the partitioning algorithm.

Devices may represent either primitive components such as transistors, capacitors, or higher-level devices such as adders or register files. Each device v has a type, $type(v)$, which distinguishes devices according to their function. A device also has a set of terminals through which it connects to nets. The terminals of a device are divided into sets of equivalence classes that represent the interchangeability of device connections. A transistor, for example, has one gate terminal and two source/drain terminals; this implies that nets connected to the source/drain terminals may be interchanged without affecting the circuit's function. Furthermore, we refer to the number of devices connected to a net n as $degree(n)$.

SubGemini, like Gemini, uses a partitioning algorithm to find an isomorphism mapping between two circuits. Partitioning is done so that equivalent vertices are contained in the same partition. Initially vertex invariants such as device type and net degree are used to partition the vertices. Because of the small number of different invariants, the initial partitioning using these invariants divides both graphs into only a few partitions. In a CMOS transistor netlist, for example, the device vertices will be divided into only two partitions: N-transistors and P-transistors. Partitions are then refined iteratively by using the information about the neighboring vertices of a vertex to classify the vertex more precisely. If partitioning proceeds in the same manner in both graphs, an isomorphism ensures that the partitions in the two graphs will match. Finding a partitioning with only singleton partitions gives a complete isomorphism mapping.

Partitioning is done implicitly via a labeling algorithm. Vertices are labeled using the vertex invariants and the labels are used to classify the vertices into partitions. Partitions are refined by relabeling each vertex by combining the previous label of the vertex with the labels of neighboring vertices. In practice, we use integers to approximate exact labels and use the labeling function shown in Fig. 3. It can be shown that this labeling gives the same result as exact labels with a very high probability. For a more detailed discussion of labeling, the reader is referred to [3, 4].

While using the partitioning algorithm is relatively straightforward when solving graph isomorphism, it does not apply directly to the subgraph isomorphism problem. The SubGemini algorithm must restrict the way partitioning is used because the graph and subgraph do not partition in the same way because they are very different graphs. The key to the SubGemini algorithm is the restricted way partitioning is done which makes it possible to solve the subgraph isomorphism problem.

We use $\mathcal{G}$ to refer to the main circuit graph and $\mathcal{S}$ to refer to the subgraph. A net in the subgraph, $\mathcal{S}$, may be one of two different kinds: *external* or *internal*. External nets connect devices in the subgraph to the surrounding graph. Internal nets connect only

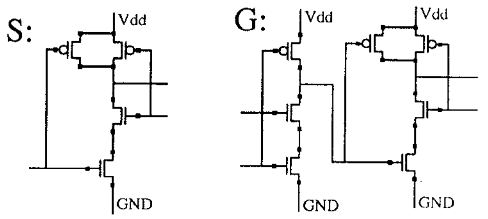

Figure 1. Transistor level representation of example circuit. There is one instance of the subgraph in the main graph at the right.

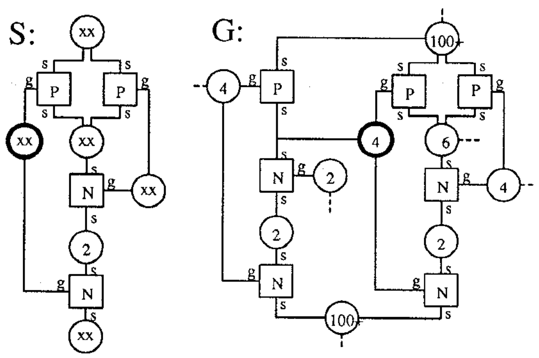

Figure 2. Graph representation of example circuit after initial labeling. Net vertices are labeled with their degree; device vertices are labeled according to transistor type. External vertices in $\mathcal{S}$ are already corrupt (xx), as they typically have a smaller degree than their images in $\mathcal{G}$. For example, see the highlighted vertices above.

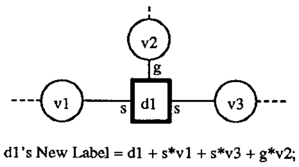

Figure 3. The general relabeling function: The new label of a vertex depends on the old label of the vertex, the labels of its neighbors, and the terminal classes through which the vertex is connected to its neighbors.

devices within the subgraph; connections between internal nets and devices outside the subgraph are forbidden. In Fig. 2, for example, all nets in $\mathcal{S}$ labeled xx are external, and the net labeled *2* is an internal net. Handling external nets correctly is important since they have differing numbers of neighbors in the subgraph and the main graph.

Finding a subgraph in the graph requires finding an isomorphism mapping between the subgraph $\mathcal{S}$ and an induced subgraph $\mathcal{G}'$ in $\mathcal{G}$. We call this subgraph, $\mathcal{G}'$, the *image* of $\mathcal{S}$ and say that a vertex $g = image(s)$ if they are isomorphic in the context of $\mathcal{G}'$ and $\mathcal{S}$. We also say that s and g *match* if $g = image(s)$.

III. Phase I - Generating the Candidate Vector

In Phase I, SubGemini locates a key vertex in the subgraph, $\mathcal{S}$, and a candidate vector of vertices in the main graph, $\mathcal{G}$, which represent possible images of the key vertex. It does so by means of an iterative partitioning algorithm that uses the structure of the subgraph to reduce the size of the candidate vector.

Initially, all vertices in both graphs are partitioned into equivalence classes according to labels based on vertex invariants: All device vertices are labeled according to their type, and all net vertices are labeled according to their degree. Vertices are then relabeled using the labels of neighboring vertices to refine the partitioning.

This leads to a problem in the context of subgraph isomorphism. The image of an external net vertex x in $\mathcal{S}$, may be connected to additional components in $\mathcal{G}$ that lie outside the image of $\mathcal{S}$. In Fig. 2, for example, the images of the external net vertices in $\mathcal{S}$ are connected to extra components in $\mathcal{G}$, as noted by the larger labels in $\mathcal{G}$ and the dotted lines to indicate additional connections. The highlighted vertices in Fig. 2 show a specific example of this.

SubGemini solves this using an additional valid bit for each vertex in $\mathcal{S}$. This valid bit is marked corrupt if the label of a vertex in $\mathcal{S}$ and its image in $\mathcal{G}$ may be different because of external connections. External net vertices are marked corrupt at the initial labeling; all other vertices in $\mathcal{S}$ are marked valid. In Fig. 2, for example, external vertices in $\mathcal{S}$ are marked xx, to show that their labels are corrupt. As vertices are relabeled, those with corrupt neighboring vertices are marked corrupt as well. This modified labeling procedure guarantees that the following label invariant holds after each relabeling:

(1) Label Invariant (Phase I): If g in $\mathcal{G}$ is the image of s in $\mathcal{S}$, then if s is marked valid, g has the same label as s.

SubGemini iteratively relabels vertices until either all net vertices or all device vertices in $\mathcal{S}$ become corrupt (during any specific iteration, only vertices of one type become corrupt since the graph is bipartite). We are left with a number of valid partitions in $\mathcal{S}$ and the corresponding partitions in $\mathcal{G}$. The smallest of these partitions is chosen as the candidate vector so that that the least amount of work has to be done in Phase II. The valid vertex from $\mathcal{S}$ with the same label as that of the candidate partition is chosen as the key vertex. If multiple valid vertices from $\mathcal{S}$ have the same label at this point, an arbitrary one can be chosen to be $\mathcal{K}$.

Label Invariant (1) guarantees that all the possible images of $\mathcal{K}$ must be in the candidate vector, although there may also be vertices in the candidate vector that are not images of $\mathcal{K}$.

Fig. 4 shows the example graph after the first relabeling during which net vertices have been relabeled. The labels A and B are computed as shown. During the next iteration, device vertices will be relabeled, and will all become corrupt because each has a corrupt net as a neighbor; at this point, the Phase I relabeling algorithm will halt. The two vertices in $\mathcal{G}$ marked A will become the candidate vector, and the vertex in $\mathcal{S}$ marked A will become the key vertex. The vertex labeled B in Fig. 4 cannot be chosen as the candidate vector, because its label does not match the label of any vertex in $\mathcal{S}$.

The Phase I relabeling algorithm can avoid unnecessary work by checking for consistency of partitions at each step. If there exists a partition P in $\mathcal{G}$, for example, which has a label that does

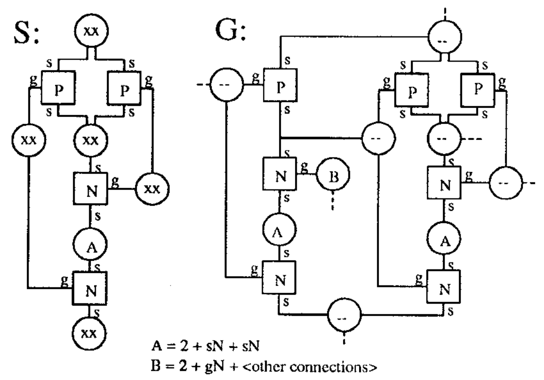

Figure 4. Graph representation of example circuit after nets are relabeled. Vertices in $\mathcal{G}$ which are marked "–" were removed from consideration as members of the candidate vector during a consistency check after the initial labeling. The label of vertex B in $\mathcal{G}$ does not match the label of any vertex in $\mathcal{S}$, so it will be removed from consideration during the consistency check after this relabeling.

not appear in any partition in $\mathcal{S}$, then those vertices in P cannot have valid images in $\mathcal{S}$, by Label Invariant (1) above, and can be safely discarded. Moreover, if after any iteration there exists a partition P_g in $\mathcal{G}$ which is smaller than the partition P_s of valid vertices in $\mathcal{S}$ with the same label, then we can deduce that there is no induced subgraph in $\mathcal{G}$ that is isomorphic to $\mathcal{S}$.

Optimized Algorithm for Phase I

Repeat:

- Relabel all valid net vertices.
- Mark invalid and remove those net vertices in $\mathcal{S}$ with invalid neighbors.
- Check for consistency.
- If all net vertices are invalidated, exit loop.
- Relabel all valid device vertices.
- Mark invalid and remove those device vertices in $\mathcal{S}$ with invalid neighbors.
- Check for consistency.
- If all device vertices are invalidated, exit loop.

Choose the smallest partition in $\mathcal{G}$ to be $\mathcal{CV}$. Choose the first element in the corresponding partition in $\mathcal{S}$ to be $\mathcal{K}$.

IV. PHASE II - EXPLORING THE CANDIDATE VECTOR

Upon completion of Phase I, SubGemini has chosen a key vertex, $\mathcal{K}$ from the subgraph, $\mathcal{S}$, and has generated a candidate vector of vertices in the main graph, $\mathcal{G}$, that might match the key vertex. During Phase II, SubGemini uses the candidate vector to know where to look in $\mathcal{G}$ for subgraphs isomorphic to $\mathcal{S}$. It examines each vertex c in the candidate vector, attempting to find an induced subgraph in $\mathcal{G}$ isomorphic to $\mathcal{S}$ where $c = image(\mathcal{K})$.

A straightforward approach is to match all the vertices of $\mathcal{S}$ to vertices located in $\mathcal{G}$ by exhaustively searching from the key vertex as in [6]. This can be very expensive, especially if done in a depth-first manner since one wrong guess early on can case much wasted time. SubGemini instead performs an implicit breadth-first search using a modified partitioning algorithm. This is more efficient in general because it maximizes the use of the information about the structure of the subcircuit during the search.

SubGemini does the following procedure for each instance c from the candidate vector in turn. It first assumes that c is the image of $\mathcal{K}$. It marks these two vertices *matched* and gives them both the same random, unique label which does not change. Starting from this one label, it then iteratively relabels the surrounding vertices in both graphs.

When relabeling vertices, SubGemini must take care that if a vertex g in $\mathcal{G}$ is the image of a vertex s in $\mathcal{S}$, then both s and g are given the same labels. Because vertices outside $\mathcal{G}'$, the assumed subgraph instance in $\mathcal{G}$, will pass unwanted information to other vertices in $\mathcal{G}$ if their labels are used during relabeling, these vertices must be excluded from the Phase II relabeling function; otherwise, the labels of s and g may differ after relabeling.

Therefore, SubGemini keeps track of whether labels are trustworthy and only uses those that are. Obviously, the label given to $\mathcal{K}$ and c is trustworthy, as these vertices are assumed to be matched. However, we still need to define when the labels of other vertices are safe. SubGemini does this by associating a safe bit (safe/suspect) for each partition in $\mathcal{S}$ and $\mathcal{G}$. A partition of vertices in $\mathcal{G}$ is said to be *safe* if it contains only images of vertices in $\mathcal{S}$. The vertices in a safe partition are also said to be safe. Additionally, the vertices in $\mathcal{S}$ whose images in $\mathcal{G}$ are safe are also safe. We assume that there exists a $\mathcal{G}'$ in $\mathcal{G}$ isomorphic to $\mathcal{S}$; therefore, when the sizes of two similarly labeled partitions in $\mathcal{G}$ and $\mathcal{S}$ are equal, we may also assume that the partition in $\mathcal{G}$ contains only those vertices that are in $\mathcal{G}'$. Consequently, we may mark all partitions in $\mathcal{G}$ that have equal-sized partitions with the same label in $\mathcal{S}$ as safe. On the other hand, a partition in $\mathcal{G}$ that is larger than its corresponding partition in $\mathcal{S}$ is necessarily suspect, because it must contain at least one vertex which is not an image of a vertex in $\mathcal{S}$, and we cannot yet determine which one that is.

The relabeling function then uses only safe labels when relabeling vertices. This guarantees that the following invariant holds after each relabeling iteration:

(2) Label Invariant (Phase II): If g in $\mathcal{G}$ is the image of s in $\mathcal{S}$, then g has the same label as s. Additionally, g and s are either both safe or both suspect.

If there exists a safe partition in both $\mathcal{G}$ and $\mathcal{S}$ with the same label, each containing only a single vertex, then these vertices are isomorphic to one another, by the definition of *safe* partitions. SubGemini marks these two vertices as *matched*, and assigns both the same random, fixed label that will be used to help further partition neighboring vertices in subsequent relabelings. SubGemini does not relabel matched vertices, but continues to relabel other vertices until all vertices in $\mathcal{S}$ are matched, or until no progress can be made.

Algorithm VerifyImage($\mathcal{K}$,$\mathcal{CV}$):

```
For each vertex $c$ in $\mathcal{CV}$ do:
    • Match $\mathcal{K}$ and $c$ together. Mark them as "safe". Assign them
      both the same unique, label.
    • While there is progress, do the following:
        – Using Phase II relabeling function, relabel neighbors of
          safe vertices in both $\mathcal{S}$ and $\mathcal{G}$.
        – Check for consistency.
        – Mark equal-sized, similarly labeled partitions in $\mathcal{S}$ and
          $\mathcal{G}$ as "safe".
        – Match singleton partitions.
        – Check for progress.
    • If all vertices have been matched, verify the isomorphism
      mapping, record the subgraph instance, and return SUCCESS.
    • If no progress has been made after some iteration, choose an
      unmatched vertex $s$ from $\mathcal{S}$ and the partition $P$ in $\mathcal{G}$ with the
      same label as $s$, and call VerifyImage($s$,$P$) recursively. If the
      recursive call succeeds, return SUCCESS. If the recursive call
      fails, go back to beginning of the For loop, to the next vertex
      $c$ in $\mathcal{CV}$.
```

An example of the Phase II relabeling algorithm is shown for the circuit in Fig. 6, the same circuit example from section III. Table 1 shows the steps SubGemini takes to find an instance of the subcircuit in the main circuit. At the end of Phase I, SubGemini chose the vertices N13 and N14 as the candidate vector, and vertex N4 as the key vertex. The columns in these tables show how the vertices are relabeled on each pass.

In this example, the key vertex and the candidate vertex are both assigned a random label, shown here as **KV**. In pass 1, only the neighbors of N4 and N14 take on new labels. In subsequent

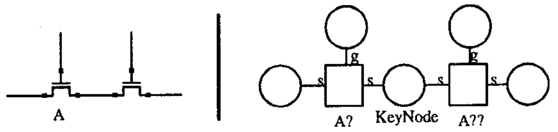

Figure 5. This is a simple case in which ambiguity will arise. Because the circuit is symmetric, the Phase II algorithm must at some point "guess" whether transistor **A** is the left or the right device vertex in the corresponding circuit graph when attempting to find a match in $\mathcal{G}$. Either choice is correct, and will lead to a match if there is one. Thus, no backtracking is required for this example.

passes, labels in effect spread out from the initially labeled vertices. Safely labeled vertices are shown in boldface and vertices are shown boxed as they become matched. By Pass 4, the partitions have been refined to the point where N1, N2 and N6 can be matched with N7, N10 and N15. Subsequent passes match more vertices and by Pass 7, all vertices in the subgraph have been matched, validating this instance of the subgraph.

Relabeling usually results in a complete matching of vertices, but ambiguity sometimes arises. When no progress is detected after relabeling and matching vertices, SubGemini chooses a partition from the subgraph and guesses a match between a vertex in this partition and a vertex from the corresponding partition in the main graph. It then continues with the relabeling procedure. If this produces a failure, then SubGemini must backtrack and try another guess (see Fig. 5). Upon completion, the Phase II relabeling algorithm either confirms the presence of a matching subgraph in $\mathcal{G}$ in which c matches $\mathcal{K}$, or shows that there really was no match to be found.

The algorithm above performs consistency checks similar to those of the Phase I algorithm. Furthermore, upon each recursive call to the algorithm, the current state, including the current labels and the list of currently matched vertices, is saved. If the call to the algorithm fails, it restores this state before returning to the caller. Additionally, "progress" is defined to occur either when at least one additional vertex is labeled and marked safe, or when two singleton partitions are matched during an iteration.

A. *Special Signals*

Usually there are special signals such as Vdd and GND which have the same meaning in the main circuit and the subcircuit. By using these special signals when specifying the subcircuit, the user may place further constraints on the subcircuit. Without this ability to treat Vdd and GND as special, SubGemini will find the CMOS inverter subcircuit shown in Fig. 7 in every NAND and NOR gate. An alternate solution to this problem is to extract gates in order using the partial order induced by the subcircuit relation on the gates. That is, one would first extract the largest gates which are not subcircuits of any other gates and then proceed to smaller and smaller gates.

Performance is also affected by the ability to refer to special signals like Vdd and GND. If these signals are treated specially, then SubGemini can avoid labeling them, a process which requires

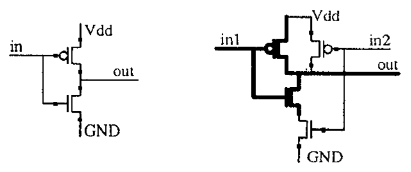

Figure 7. The inverter circuit on the left will be found as a subcircuit in the NAND circuit, as shown in bold, at the right unless Vdd and GND are recognized as special nets.

TABLE 1

Vertex	Init	Pass 1	Pass 2	Pass 3	Pass 4
Sub Graph: $\mathcal{S}$					
D1				$\mathbf{D} = p + gC + sB$	$\mathbf{H} = D + sB + gC$
D2				$\mathbf{D} = p + gC + sB$	$\mathbf{H} = D + sB + gC$
D3		$\mathbf{A} = n + sKV$	$\mathbf{A}$	$\mathbf{E} = A + sB + gC + sKV$	$\mathbf{J} = E + sKV + sB + gC$
D4		$\mathbf{A} = n + sKV$	$\mathbf{A}$	$\mathbf{E} = A + sB + gC + sKV$	$\mathbf{J} = E + sKV + sB + gC$
N1					$\boxed{\mathbf{K}} = sD + sD$
N2			$\mathbf{B} = sA$	$\mathbf{B}$	$\boxed{\mathbf{L}} = B + sD + sD + sE$
N3			$\mathbf{C} = gA$	$\mathbf{C}$	$\mathbf{M} = C + gD + gE$
N4	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$
N5			$\mathbf{C} = gA$	$\mathbf{C}$	$\mathbf{M} = C + gD + gE$
N6			$\mathbf{B} = sA$	$\mathbf{B}$	$\boxed{\mathbf{N}} = B + sE$
Main Graph: $\mathcal{G}$					
D5				$F = p + sC$	–
D6				$\mathbf{D} = p + gC + sB$	$\mathbf{H} = D + sB + gC$
D7				$\mathbf{D} = p + gC + sB$	$\mathbf{H} = D + sB + gC$
D8				$F2 = n + sC$	–
D9		$\mathbf{A} = n + sKV$	$\mathbf{A}$	$\mathbf{E} = A + sB + gC + sKV$	$\mathbf{J} = E + sKV + sB + gC$
D10				$G = n + sB$	–
D11		$\mathbf{A} = n + sKV$	$\mathbf{A}$	$\mathbf{E} = A + sB + gC + sKV$	$\mathbf{J} = E + sKV + sB + gC$
N7					$\boxed{\mathbf{K}} = sD + sD$
N8			$\mathbf{C} = gA$	$\mathbf{C}$	$\mathbf{M} = C + gD + gE$
N9			$\mathbf{C} = gA$	$\mathbf{C}$	$\mathbf{M} = C + gD + gE$
N10			$\mathbf{B} = sA$	$\mathbf{B}$	$\boxed{\mathbf{L}} = B + sD + sD + sE$
N14	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$
N15			$\mathbf{B} = sA$	$\mathbf{B}$	$\boxed{\mathbf{N}} = B + sE$

Vertex	Pass 5	Pass 6	Pass 7
Sub Graph: $\mathcal{S}$			
D1	$\mathbf{P} = H + sK + sL + gM$	$\mathbf{W} = P + sK + sL + gS$	$\boxed{\mathbf{CC}} = W + sK + sL + gY$
D2	$\mathbf{P} = H + sK + sL + gM$	$\mathbf{W} = P + sK + sL + gS$	$\boxed{\mathbf{DD}} = W + sK + sL + gX$
D3	$\boxed{\mathbf{Q}} = J + sKV + sL + gM$	$\boxed{\mathbf{Q}}$	$\boxed{\mathbf{Q}}$
D4	$\boxed{\mathbf{R}} = J + sKV + sN + gJ$	$\boxed{\mathbf{R}}$	$\boxed{\mathbf{R}}$
N1	$\boxed{\mathbf{K}}$	$\boxed{\mathbf{K}}$	$\boxed{\mathbf{K}}$
N2	$\boxed{\mathbf{L}}$	$\boxed{\mathbf{L}}$	$\boxed{\mathbf{L}}$
N3	$\mathbf{S} = M + gH + gJ$	$\boxed{\mathbf{X}} = S + gP + gQ$	$\boxed{\mathbf{X}}$
N4	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$
N5	$\mathbf{S} = M + gH + gJ$	$\boxed{\mathbf{Y}} = S + gP + gR$	$\boxed{\mathbf{Y}}$
N6	$\boxed{\mathbf{N}}$	$\boxed{\mathbf{N}}$	$\boxed{\mathbf{N}}$
Main Graph: $\mathcal{G}$			
D5	–	–	–
D6	$\mathbf{P} = H + sk + sL + gM$	$\mathbf{W} = P + sK + sL + gS$	$\boxed{\mathbf{CC}} = W + sK + sL + gY$
D7	$\mathbf{P} = H + sk + sL + gM$	$\mathbf{W} = P + sK + sL + gS$	$\boxed{\mathbf{DD}} = W + sK + sL + gX$
D8	–	–	–
D9	$\boxed{\mathbf{Q}} = J + sKV + sL + gM$	$\boxed{\mathbf{Q}}$	$\boxed{\mathbf{Q}}$
D10	–	–	–
D11	$\boxed{\mathbf{R}} = J + sKV + sN + gJ$	$\boxed{\mathbf{R}}$	$\boxed{\mathbf{R}}$
N7	$\boxed{\mathbf{K}}$	$\boxed{\mathbf{K}}$	$\boxed{\mathbf{K}}$
N8	$\mathbf{S} = M + gH + gJ$	$\boxed{\mathbf{X}} = S + gP + gQ$	$\boxed{\mathbf{X}}$
N9	$\mathbf{S} = M + gH + gJ$	$\boxed{\mathbf{Y}} = S + gP + gR$	$\boxed{\mathbf{Y}}$
N10	$\boxed{\mathbf{L}}$	$\boxed{\mathbf{L}}$	$\boxed{\mathbf{L}}$
N14	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$	$\boxed{\mathbf{KV}}$
N15	$\boxed{\mathbf{N}}$	$\boxed{\mathbf{N}}$	$\boxed{\mathbf{N}}$

Example Phase II relabeling algorithm: n and p represent device types, g and s represent terminal types, boldfaced letters represent safe vertices and boxed letters indicate vertices that have been matched.
N11, N12 and N13 are never labeled in $\mathcal{G}$, and are therefore not represented in the above table.

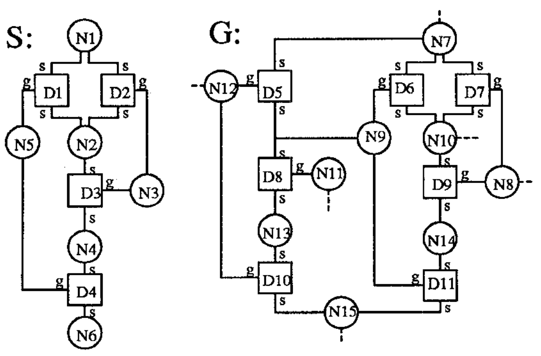

Figure 6. Circuit to be labeled by Phase II algorithm.

TABLE 2
EXPERIMENTAL RESULTS

Name $\mathcal{G}$	Name $\mathcal{S}$	Size $\mathcal{G}$	Size $\mathcal{S}$	Size $\mathcal{CV}$	Num Found	Recurs. Calls	Setup Time	Phase I Time	Phase II Time	Total Time
256 bit	2 Input NAND	8672	4	1016	1016	0	3.2	1.0	5.8	10.0
Adder	2 Inut NOR		4	768	0	0	3.1	0.9	3.3	7.3
	OAI gate		6	764	764	0	3.3	1.1	6.0	10.4
Barrel	4:1 Mux	16128	12	2688	1344	0	10.5	2.6	46.0	59.2
Shifter	4:1 Mux		8	5376	1344	2679	10.7	3.4	66.6	80.7
(64 64:1	(no inverts)									
Muxes)	16:1 Mux		60	1280	256	0	10.0	2.1	47.3	59.4
	64:1 Mux		252	256	64	0	11.0	2.3	35.9	49.2
	E-Chain		2	5376	5376	10752	9.7	3.4	29.4	42.5
RAM Array	RAM Cell	6000	6	2000	1000	0	1.9	1.1	5.3	8.3
		24000	6	8000	4000	0	9.5	5.1	29.1	43.7
		96000	6	32000	16000	0	36.4	20.4	176.5	233.3
RAM Array	Deviant RAM Cell	6000	6	1	1	0	2.2	0.5	0.0	2.7
		24000	6	1	1	0	8.9	1.9	0.0	10.8
		96000	6	1	1	0	37.7	9.5	0.0	47.2
RAM Array	Inverter	6000	2	2000	2000	0	2.1	1.0	4.2	7.3
		24000	2	8000	8000	0	8.6	4.6	14.6	28.8
		96000	2	32000	32000	0	37.0	16.6	62.4	116.0
CrossBar	BitCell	12183	8	420	420	0	8.7	1.2	1.8	11.7
	FIFO Buffer		3089	2	2	18	11.1	1.7	85.0	97.8
OutFrame	BitCell	19976	8	1050	1050	0	18.6	2.4	5.2	26.2
	FIFO Buffer		3089	5	5	45	19.3	2.6	256.4	278.3
Router	BitCell	55089	8	2100	2100	0	56.6	7.2	9.7	73.5
	FIFO Buffer		3089	10	10	90	56.5	7.9	820.0	884.4

examining a very large number of unrelated devices in the entire circuit. For example, when relabeling Vdd in Phase II of one of the RAM experiments, SubGemini needs to check all 48,000 neighbors of Vdd to see if any are marked *safe*. The algorithm must perform this check twice for each of the 16,000 RAM cells in the array, thus making 768 million checks overall.

Therefore, SubGemini allows the user to indicate that signals such as Vdd and GND are to be treated as special. The relabeling algorithm then assigns Vdd and GND unique labels that are not changed by the Phase I and Phase II relabeling algorithms. These special labels allow SubGemini to quickly identify Vdd and GND; moreover, by marking Vdd and GND as special, SubGemini avoids extensive rechecking of their neighbors.

V. Experiments

In this section we present a set of experiments that shows the operation and performance of SubGemini over a variety of examples. In all these examples both the circuit and the subcircuit are transistor netlists in SIM format. The circuits listed in Table 2 were chosen to illustrate several aspects of the SubGemini algorithm. In the first example, SubGemini is used to find several different gates in a large standard cell implementation. The second example shows how SubGemini performs on a large, symmetric circuit where it is very difficult to distinguish the boundaries of the individual subcircuits. The third example features a large RAM array which contains many copies of the same subcircuit. In the final example, SubGemini is used to find both a small circuit and a very large subcircuit in an actual chip design.

Table 2 gives the sizes of the circuits and subcircuits in terms of the number of transistors. Also given are the size of the candidate vector, the number of instances found, and the number of recursive calls to *VerifyImage*. Except where indicated, all experiments were done with Vdd and GND treated as special signals. The performance shown is the CPU time on a Sun 4/490 with 32 Mbytes of memory.

In the first experiment, SubGemini finds three different gates in a standard cell implementation of a 256-bit ripple carry adder generated using MIS, a multi-level logic optimization tool. This represents the common task of gate extraction for a medium sized circuit and the results show that the algorithm works very well for this case.

The second experiment uses a 64-bit barrel shifter composed of 64 64:1 multiplexors, each composed of a tree of the 4:1 multiplexors implemented using pass-gate logic. Note that because of the pass gate logic, standard techniques that use signal direction or channel graphs do not work. For the experiment in the second line of this example, we remove the inverters from the 4:1 multiplexor subcircuit. The remaining structure is now symmetric, which is reflected by the increased size of the candidate vector and the number of recursive calls to *VerifyImage* that are required. Even so, the running time does not increase substantially. In the next two experiments, SubGemini looks for a 16:1 multiplexor made from five 4:1 multiplexors and for a complete 64:1 multiplexor. Finally, in the E-chain experiment, SubGemini finds all isolated chains of two n-type transistors.

The next set of three experiments use a static RAM array made from the 6 transistor RAM cell in Fig. 8. In all three experiments the size of the array ranges in size from 1K to 16K cells. In the second experiment, the type of one transistor in one RAM cell of the array is changed, and SubGemini is used to find that one "deviant" cell. This experiment shows the power of the Phase I algorithm to filter out subcircuits that do not match. In the third experiment, we find all the inverters in the array.

The last experiment features the implementation of the Chaos router chip currently being designed at the University of Washington[2] and SubGemini is used to find all instances of a small and a large subcircuit. The FIFO subcircuit is a fifo comprising 21 10-bit entries implemented as a circular buffer with head and tail pointers. The BitCell subcircuit is a single memory cell used

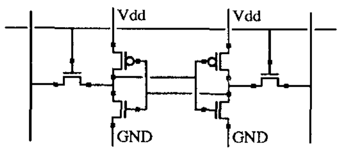

Figure 8. The 6-transistor static RAM circuit.

in the FIFO. The long time taken to find the FIFO subcircuit is caused by the large amount of symmetry in the memory array in the FIFO which slows down the labelling algorithm. Each instance of the FIFO causes 9 recursive calls on *VerifyImage*, one for each column in the memory array that must be disambiguated. However, no backtracking is required.

The running time of the algorithm depends largely on the running time of Phase II. The setup time measures how long it takes to read the input file and create the internal data structures and is linear in the size of the input files. Phase I time is actually less than the setup time and thus is not a factor. Phase II time depends on the structure of the graph and the subcircuits. SubGemini can find subcircuits like standard cell gates and RAM cells that have sufficient structure to allow the labelling algorithm to work efficiently. In these cases, the running time of the algorithm is approximately linear in the size of the candidate vector times the size of the subcircuit being sought, which is bounded by the size of the graph. For subcircuits with substantial internal symmetry or which have poorly defined boundaries, vertices must be relabeled many times and SubGemini is somewhat slower, as shown in the multiplexor and FIFO examples. In spite of this, the running time remains reasonable, largely because Phase I is usually able to find the right circuits for the candidate vector.

VI. Discussion and Future Work

Formulating the problem of finding subcircuits in a graph as an abstract graph isomorphism problem allows a general, technology and domain independent solution. Using a pure subgraph isomorphism algorithm does have some side effects however. In particular, the Phase II relabeling algorithm relies on the assumption that external nets are not shorted to other external nets of the same subgraph within the larger circuit. An example of a shorted circuit is shown in Fig. 9. While shorting inputs in this fashion does not change the subcircuit instance in the view of the user, in our representation of the circuit it does create a subgraph which is not isomorphic to the subgraph representing the original subcircuit, and thus our subgraph isomorphism algorithm does not find it.

This restriction, however, may be relaxed to apply only to external nets connected to more than one device in $\mathcal{S}$. Purely external nets (external nets connected to only one device in $\mathcal{S}$) can be recognized and handled separately. Because each purely external net of $\mathcal{S}$ is connected to only one device in $\mathcal{S}$, it can always be matched as long as the device to which it is connected is matched to a corresponding device in $\mathcal{G}$. Therefore, SubGemini optionally pre-matches all purely external nets which are not marked special in $\mathcal{S}$.

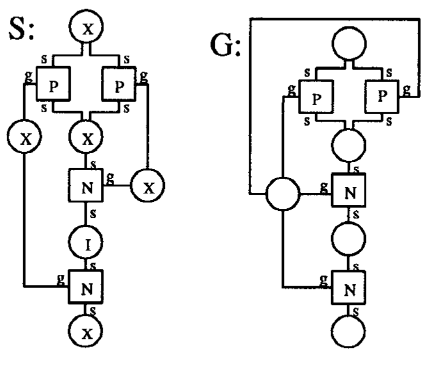

Figure 9. The instance of the NAND gate on the right has its two inputs shorted within $\mathcal{G}$. Thus it no longer matches the NAND subcircuit, $\mathcal{S}$, at the left since it has one input vertex while $\mathcal{S}$ has two.

It is possible to solve the more general shorted inputs problem using a brute-force method that looks for all possible instances of legally shorted inputs, but this increases the running time by a factor approximately equal to the number of such instances. A more clever algorithm should be possible that uses the properties of the labels in Phase II, and this is a topic of ongoing research. Additionally, the task that Phase II performs is made up of many identical, independent tasks, one for each vertex in the candidate vector. We believe that there is potential for nearly linear speedup in Phase II using a simple parallel version of our relabeling algorithm.

We have presented a general algorithm for recognizing subcircuits in a large circuit graph using a fast subgraph isomorphism algorithm. This algorithm has no knowledge about the underlying circuit except for its structure, and thus it can be used for any circuit technology. This algorithm is fast enough to be applicable to the many CAD problems where a solution to the subgraph isomorphism problem is needed.

References

[1] Michael Boehner. LOGEX - An Automatic Logic Extractor from Transistor to Gate Level for CMOS Technology. In *Proceedings of the 25th Design Automation Conference*, pages 517–522, June 1988.

[2] Kevin Bolding and Lawrence Snyder. Mesh and Torus Chaotic Routing. In *Proceedings of the Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, March 1992.

[3] C. Ebeling and O. Zajicek. Validating VLSI Circuit Layout by Wirelist Comparison. In *Proceedings of the Conference on Computer Aided Design (ICCAD)*, pages 172–173, 1983.

[4] Carl Ebeling. GeminiII: A Second Generation Layout Validation Tool. In *Proceedings of the Conference on Computer Aided Design (ICCAD)*, pages 322–325, November 1988.

[5] F. Luellau, T. Hoepken, and E. Barke. A Technology Independent Block Extraction Algorithm. In *Proceedings of the 21st Design Automation Conference*, pages 610–615, 1984.

[6] Georg Pelz and Uli Roettcher. Circuit Comparison by Hierarchical Pattern Matching. In *Proceedings of the Conference on Computer Aided Design (ICCAD)*, pages 290–293, 1991.

[7] T. Watanabe, M. Endo, and N. Miyahara. A New Automatic Logic Interconnection Verification System for VLSI Design. *IEEE Transactions on Computer Aided Design*, CAD-2:70–82, April 1983.