Copy Detection for Intellectual Property Protection of VLSI Designs*

Andrew B. Kahng, Darko Kirovski, Stefanus Mantik, Miodrag Potkonjak and Jennifer L. Wong
{abk,darko,stefanus,miodrag,wongjen}@cs.ucla.edu

UCLA Computer Science Dept., Los Angeles, CA 90095-1596

Abstract

We give the first study of copy detection techniques for VLSI CAD applications; these techniques are complementary to previous watermarking-based IP protection methods in finding and proving improper use of design IP. After reviewing related literature (notably in the text processing domain), we propose a generic methodology for copy detection based on determining basic *elements* within structural representations of solutions (IPs), calculating (context-independent) signatures for such elements, and performing fast comparisons to identify potential violators of IP rights. We give example implementations of this methodology in the domains of scheduling, graph coloring and gate-level layout; experimental results show the effectiveness of our copy detection schemes as well as the low overhead of implementation. We remark on open research areas, notably the potentially deep and complementary interaction between watermarking and copy detection.

1 Introduction

With more functionality integrated on a single chip and shorter design cycle times, design reuse methodologies have become a focal point of industrial and academic activities. To enable the reuse-based paradigm and its associated business model, intellectual property protection (IPP) is an essential prerequisite: the rights of both the IP provider (owner, creator) and the IP buyer (user, integrator) must be protected.

There are two main approaches to IPP: *prevention* of unauthorized use, and *detection* of unauthorized use. Techniques for prevention, which are analogous to "locks" on the IP, include encryption, legal infrastructure, and closed infrastructures for IP dissemination. Techniques for detection, on the other hand, are aimed at discovering illegal copies of IP *after* the "locks" have been broken.[1] In the VLSI CAD realm, the constraint-based watermarking approach to IPP has received particular attention [16]: it prevents misappropriation by indelibly embedding the owner's signature into an IP, so that if an illegal copy is found the true owner's rights can be established. However, the utility of watermarking is mostly after an illegal copy of IP has been found. The question of how to find the illegal copy in the first place – the *copy detection problem* – has not yet been addressed for VLSI CAD.

We informally define the copy detection problem as follows: *Given a library of n* registered *pieces of IP, and a new unregistered piece of IP, determine if any portion of any registered IP is present in the unregistered IP.* This definition reflects the use model for, say, a foundry which runs a copy detection program on any incoming design at the level of GDSII Stream representation. Copy detection is clearly complementary to existing watermarking-based IPP techniques; below, we will show that it can also be enhanced by watermarking techniques.

1.1 Related Work

While our work gives, to the best of our knowledge, the first effort on copy detection in CAD, there is a large body of related research in several fields.

Research on copy detection and plagiarism started in the early 1970s mainly as a technique for preventing widespread programming assignment copying [26] and to help support software reuse [19]. Over time a number of increasingly sophisticated techniques have been developed for programming assignment copy detection [12, 25, 29]. Most recently, even fractal and neural network-based techniques have been proposed for this task [23, 28].

The research closest to that presented in this paper has been conducted in the database community, notably for text copy detection; cf. techniques developed at Stanford [5, 27] and elsewhere [21]. A key approach is to find "signatures" (e.g., by hashing) of syntactically meaningful fragments (e.g., words or paragraphs), then create "term-document" or other incidence matrices that capture the presence of fragments within documents or IPs. Such incidence matrices are captured for all elements of a library of registered IPs. Then, when presented with a new IP, the copy detection system chunks the IP into fragments, and looks for matches of signatures in its library.

Copy detection for VLSI CAD has been mainly performed at the layout level, where there is a need to eliminate or reduce redundant computation during VLSI artwork analysis (design rule checking, layout-versus-schematic (LVS) and pattern-based parasitic extraction). Techniques include isometry-invariant pattern matching [6, 22] and fast subgraph isomorphism algorithms [24].[2] Somewhat related work addresses template matching at various levels of the design process, where a design is covered by smaller templates available in a given library [15, 9].

A third area of related work is in string matching, which has received a great deal of attention since the early 1970s; see [1] for an excellent review. Several exceptionally effective algorithms have been proposed for rapid string matching in text [4, 18, 17]. Awk [2] is a popular and powerful programming language that greatly facilitates development of tailored pattern scanning and processing software. Finally, a number of copy detection techniques have been developed in biotechnology [3] and image processing [11].

1.2 Contributions

Unique requirements for copy detection arise in the VLSI CAD context, including "invariance" of various design representations under hypergraph isomorphism, alternative spatial embedding, scaling and rotation, renaming, etc.[3] Hence, our work differs from

* This research was supported in part by NSF under grant CCB-9734166, by a grant from Cadence Design Systems, Inc., and by the MARCO/DARPA Gigascale Silicon Research Center.

[1] Note that in many contexts, "locks" such as proprietary reader/writer hardware, secure networks, etc. can harm the traditional business model. This increases the importance of detection in achieving IPP.

[2] LVS essentially assumes that it knows what has been copied (i.e., the schematic into the layout), and simply verifies the isomorphism. However, in our domain we do not know *a priori* what, if anything, has been copied.

[3] The VLSI CAD domain also offers an interesting scale reversal when compared to, e.g., the web-text domain. A foundry may have only thousands of registered design

those reviewed above in several key aspects, including our domain, our algorithmic and statistical techniques, and our detection goals. In Section 2, we propose a generic methodology for copy detection which efficiently identifies potential violations of IP rights. Section 3 describes example implementations of copy detection in the domains of scheduling, graph coloring and gate-level layout. Experimental results show the effectiveness of our copy detection schemes, as well as the low overhead of implementation. We conclude in Section 4 with areas for future research, including potential synergies between watermarking and copy detection.

2 A Generic Copy Detection Methodology

Our generic copy detection methodology has the following elements.

- For the given application domain, we identify a common structural representation of solutions (IPs), as well as what constitutes an "element" of the solution structure. Examples of such elements might include vertices in a netlist hypergraph, placed locations of edges in a custom layout, macros in a hierarchical GDSII Stream description of layout, steps in a schedule, and so on.
- For a given element type, we identify a means of calculating *locally context dependent* signatures for such elements, i.e., signatures that are functions of only an extremely local neighborhood of the element.
- Optionally, to speed comparison of IPs, we identify rare and/or distinguishing elements of a registered IP (cf. "iceberg queries" in [10]), and/or a hierarchy of signature types that may lead to faster filtering of negative (no match) comparisons.
- We develop fast (ideally, linear in the sizes of the IPs) comparison methods to identify suspicious unregistered IPs, e.g., by rare combinations of rare signatures.

Subsequently, more detailed examination of suspicious IPs can be performed.

3 Specific Techniques and Experiments

3.1 Example: Scheduling in High-Level Synthesis

In this subsection, we define the objectives and methods for copy detection of programs used in system-level synthesis. An IP consists of a number of high-level procedures linked in an arbitrary fashion (e.g., DCT, vector motion compensation in MPEG). We assume:

- the adversary extracts a procedure or an entire library from the IP (e.g., DCT), and embeds the extracted code into his/her design;
- the adversary relinks the extracted procedures in an arbitrary fashion but without significant modification of the actual specification within each of the procedures; and
- the adversary may inline a procedure in the newly created specification or conduct peephole (local) perturbations.

IPs, but each may contain many millions of syntactic fragments. By contrast, the World-Wide Web contains millions of IPs, but each contains at most thousands of syntactic fragments.

We adopt this set of assumptions because of common risks involved in code obfuscation [8] and requirements for hardware-software maintenance (e.g. patches, incremental synthesis).
The goal of the copy detection algorithm is to detect all procedures that have been copied from the original software. To perform this task, we have developed a copy detection mechanism operating at both the instruction selection level and the register assignment level; only the former is described here.

We state the problem of copy detection for high-level synthesis as follows. Given a set P of registered instruction sequences (procedures) of arbitrary lengths, and a *suspected* (i.e., suspicious) instruction sequence S, find the subset $P_0 \subseteq P$ consisting of all instruction sequences (procedures) $P_i \in P$ that occur in S (i.e., P_0 is a maximal subset such that $\forall P_i \in P_0$, $P_i \in S$).

```
Find probability p_i of occurrence of each symbol a_i from
  alphabet a_i ∈ A in a given set P of code sequences.
Select a subset B of symbols from alphabet B ⊆ A
  such that ∀a_i ∈ B, p_i > 0 ∨ p_i < ε.
PoolPatterns = empty.
For each P_i ∈ P
   Identify locations of all symbols from B.
   Create a pattern pat for each K-tuple of symbols from B,
     with max distance d among elements of the K-tuple (d < θ).
   Add pat(symbols, distanceMatrix) to PoolPatterns.
EndFor
Select the least frequent M patterns from PoolPatterns.
Find a subset C ⊆ B of symbols that covers all patterns
  from PoolPatterns_M and has the smallest sum of
  occurrence probabilities.
For each element c ∈ C
  If c ∈ S
    For each pattern pat ∈ PoolPatterns_M that contains c
      If corresponding symbols are at locations
        specified by c.distanceMatrix signal pat is found.
        The locations of symbols are verified in increasing
        order of the symbol's probability of occurrence.
      If pat is found
        Perform pattern matching to verify copy [17, 13].
```

Figure 1: Pseudo-code for software copy detection at the instruction selection level (pre-processing and detection).

To address this problem, we have developed an algorithm that uses probabilistic bounded search to identify copies. The algorithm is described in the pseudocode of Figure 1. We define a set of symbols A, the alphabet, which corresponds to the machine instruction set. Let p_i be the frequency of occurrence of symbol $a_i \in A$ in a given set P of code sequences. The algorithm initially determines the value of p_i for all a_i. Then, a subset $B \subseteq A$ of symbols from the alphabet is selected such that for each symbol $b_i \in B$ the probability of its occurrence is greater than zero and smaller than a predetermined constant ε ($0 < p_i < \varepsilon$), where ε is the bound for the probabilistic search.

For each procedure $P_i \in P$, the algorithm identifies the locations of all symbols from B. We consider "signatures" based on K-tuples of symbols from B. In particular, we find all K-tuples for which d, the maximum distance between any two elements of the K-tuple, is less than a prescribed value θ. The algorithm then creates a pattern *pat* for each such K-tuple. Due to the possibility of basic block reordering, the distance between two symbols is computed according to the distance in the dynamic execution. In addition, due to possible instruction reordering, symbols are not searched at exact distances, but within a neighborhood (of cardinality N) of the exact location.[4] Parameters ε, K, N, and θ are selected such that all pro-

[4]The value of parameter N determines the sensitivity of the copy detection process. Larger values enable the algorithm to handle greater perturbations by instruction reordering, but increase runtime since more patterns are generated.

cedures from P contain at least one pattern. The probability that a specific pattern appears in a code sequence is:

$$Prob\{pat \subset S\} = \sum_{i=1}^{K} \prod_{j=1}^{i} (1 - (1 - p_{c_j \in pat})^N)$$

All identified patterns are stored in a pool of patterns, (*PoolPatterns*). Each pattern is represented using its symbols and the matrix that specifies the distances between symbols. To reduce the sample of IP code selected for comparison, the algorithm selects a set of M least frequent patterns from *PoolPatterns* that cover all procedures from P; this is called the *constrained PoolPatterns* set. The algorithm also identifies a subset $C \subseteq B$ of symbols that cover all patterns from the constrained *PoolPatterns* and has the smallest sum of symbol occurrence probabilities.

Finally, the suspected sequence of instructions is sequentially parsed for symbols from C. If a symbol $c \in C$ is found, all patterns that contain c are matched using their distance matrices for occurrence of the remainder of their symbols. The remaining symbols are searched in the order of their occurrence probabilities. If a specific pattern is identified in S, the algorithm performs an exact pattern matching of all procedures that contain *pat* and S to verify the copy detection signal [17], or else performs nonexact pattern matching using the *diff* utility program [13].

Application	Suspected Code Size	IP Procedures	P_{FA}	CPU time detection
JPEG encoding	391	24	$3.3 \cdot 10^{-7}$	1.8s
JPEG decoding	379	24	$1.8 \cdot 10^{-8}$	1.2s
PGP encryption	443	36	$2.1 \cdot 10^{-9}$	3.2s
MPEG decoding	114	17	$5.2 \cdot 10^{-11}$	2.9s
G.721 encoding	26	4	$9.1 \cdot 10^{-5}$	0.0s
GSM encoding	98	8	$8.1 \cdot 10^{-7}$	0.7s

Table 1: Effectiveness of the copy detection mechanism for behavioral specifications.

Experimental Confirmation

We have performed a set of experiments to evaluate the effectiveness of the copy detection mechanism for behavioral specifications. We use the standard multimedia benchmark applications [20], Sun's UltraSparc instruction set and its instruction-set simulator SHADE [7]. In the preprocessing step, for the set of applications shown in Table 1, we identify the distribution of occurrence of instructions as well as the required distance matrices for all established patterns. Since the volume of data is large, we have stored detailed histograms at `http://www.cs.ucla.edu/~leec/mediabench/applications.html`. Because the performance of the copy detection mechanism is by and large based on the statistical analysis of the IP code, the approach performs lengthy explorations in the pre-processing step with an objective to increase the performance of the algorithm (i.e., lower $Prob\{pat \subset S\}$). While the pre-processing step took, on the average, 46 hours for a single application, the actual detection process required in all experimental cases is less than 10 seconds. Table 1 shows the obtained results for the detection process. Column 1 shows the name of the application; Column 2 shows the size of the suspected code and the number of procedures; Column 3 shows the number of "original" procedures; Column 4 shows the cumulative probability of false alarm P_{FA}; and Column 5 shows the CPU time for the detection process. For all IP procedures, the probability of detection was 100%. As presented in Table 1, the probability of a false alarm, accumulated for all considered patterns, quantifies the performance of the algorithm because it is proportional to the number of negative tests due to exact pattern matching.

3.2 Example: Graph Coloring

We have developed a similar copy detection strategy for graph coloring. The graph coloring problem has many applications in CAD and software compilation, mainly related to resource assignment problems. The copy detection algorithm for solutions to the graph coloring problem contains two subprocedures. The first subprocedure identifies the coloring pattern and the structure of a subgraph S of the interval graph $S \in G$ of the synthesized IP. Subgraph S is selected using a statistical methodology that targets inclusion of nodes with characteristics that are unique or infrequent throughout G (e.g. number of neighbors, cardinality of the color class, number of edges in all neighbors, etc.). It is important to stress that the coloring of the selected subgraph is memorized as a relative coloring (i.e., only difference in colors is stored, rather than usage of particular colors).

In the second subprocedure, the algorithm tries to determine whether the suspected design (i.e. its interval graph) D contains a colored subgraph equivalent to S. To achieve this, the detection procedure searches for a particular structure and its relative coloring using a search guided in the increasing likelihood of a particular node constraint. If the comparison procedure returns a positive comparison, then the entire original IP is matched against the suspected design in an exhaustive search. The exhaustive search is greatly facilitated by the fact that most unique parts of the colored graphs are already matched in the previous phase.

Experimental Confirmation

We have tested our copy detection approach on a set of graph coloring examples. The first step was to identify common patterns on a large set of random and compiler-produced graphs. To identify 100 patterns of length 8 to 15 on a set of 100 large graphs (the number of nodes was between 200 and 4,000), we required slightly less than 3 days of run time on a Sun Ultra-5 workstation. Note that this is one-time expense. Next, we embedded 20 of test graphs in large graphs and mixed the graphs within a set of 200 graphs. We were able to identify all 20 graphs. Although there were 41 false alarms in the first phase, all were detected within the first second of attempted detailed matching. The total run time for copy detection was 125 minutes.

3.3 Example: Gate-Level Netlist

In the automated place-and-route domain, we seek to protect a gate-level cell netlist that may contain embedded placement information. Such a design artifact typically arrives in Cadence Design Systems, Inc. LEF/DEF interchange format; we parse this to yield a netlist hypergraph with pin direction information. The fundamental test for netlist copying is *isomorphism checking*, i.e., finding subhypergraphs of one (unregistered) netlist in another (registered) netlist. Isomorphism checking is essentially near-linear time for rigid graphs, i.e., graphs without automorphisms – and this includes almost all graphs (cf., e.g., [24] in the VLSI CAD literature). Nevertheless, we must still *filter* calls to isomorphism checkers, because there are so many subhypergraphs that are potentially subject to copying. Filtering depends on (a hierarchy of) comparisons that span a continuum between "coarse" and "detailed", and is what enables practically useful methods.[5]

[5] For example, checking whether two chips' netlists have the same number of cells, same number of macro types, same sorted cell degree sequences, same number of con-

Our filtering approach is based on finding a "signature" for each individual cell (i.e., vertex in the netlist hypergraph) using a simple encoding of the cell's neighborhood. Specifically, we record for cell c_i the sequence of values:[6]

- $|N_{i,1}|$ = the cardinality of the set of distinct nets incident to c_i,
- $|C_{i,1}|$ = the cardinality of the set of distinct cells on the nets in $\{N_{i,1}\}$,
- $|N_{i,2}|$ = the cardinality of the set of distinct nets incident to the cells in $\{C_{i,1}\}$, etc.

Several practical considerations arise. (1) Because the diameter of a netlist hypergraph is not large, and because we would like such signatures to identify specific cells even in a small fragment of the original netlist, we record only the first k elements of this sequence (in our experiments below, we use $k = 6$). On the other hand, to increase the likelihood that such sequences can uniquely determine a match, we actually compute such sequences in several variants of the hypergraph, corresponding to deleting hyperedges whose degree exceeds some threshold d. (In the experiments below, we generate three sequences for each cell, corresponding to $d = 4, 7, 10$.) We also break each entry of the sequence into subentries according to pin direction (in, out, in-out). Thus, there are $6 \times 3 \times 3 = 54$ numbers in each cell's sequence. (2) Finding one match of all 54 numbers in a sequence is much rarer than, say, three different matches of 18 numbers. To capture this, we give geometrically more *credit* for a longer match, e.g., credit $= 2^{\lfloor (x-1)/9 \rfloor}$, where x is the number of positions in which two sequences' entries match. (3) Finally, because we do not wish to spend CPU time comparing all cell sequences from the unregistered IP against all cell sequences from the registered IP, we lexicographically order the entries of the 54-number sequences with all entries due to $i = 1$ before entries due to $i = 2$, etc. Furthermore, we adopt the convention that the number of positions in which two sequences match is simply given by the length of the longest common prefix of both sequences. In this way, finding the *best* matches for *all* sequences of the unregistered IP, within the list of sequences for the registered IP, is accomplished in linear time by pointer-walking in two sorted lists. Hence, we do not need to resort to use of "rare" signatures for complexity reduction.[7]

Experimental Confirmation

We have applied the above procedure to compute cell sequences for 6 industry standard-cell designs in LEF/DEF format. The number of cells in the designs (Cases A - F) are respectively 3286, 12133, 12857, 20577, 57275 and 117617. Cases E and F are from the same design team and may contain common subdesigns. Table 2 shows the total matching credits when Case i is matched into Case j, i.e., the best match for each cell in Case i is found within Case j. Table 3 shows the total matching credits when a *portion* of Case i (a connected component of 500 cells, found by breadth-first search from a randomly chosen cell) is matched into Case j. (Here, the results are averaged over three separate trials.) We express the total matching credit as a percentage of the maximum possible total credit. In our current use model, all registered IPs are checked against the unregistered IP. Hence, we are able to see which IPs have higher matching credits relative to the other IPs. Typically, matching percentages for non-copied IPs are in a fairly narrow range, while those of copied IPs are significantly higher.[8] We see that the proposed signature scheme very effectively identifies the correct potential matches.

Weighted Matching Between Full Designs						
	A	B	C	D	E	F
A	100%	7.99%	2.90%	4.19%	2.84%	2.84%
B	2.89%	100%	1.27%	2.12%	1.26%	1.26%
C	0.77%	0.76%	100%	1.20%	0.72%	0.73%
D	2.69%	2.79%	0.42%	100%	0.33%	0.33%
E	0.25%	0.36%	0.25%	0.24%	100%	30.3%
F	0.16%	0.27%	0.16%	0.15%	28.7%	100%

Table 2: Matching percentage between two full designs, based on weighted sum of credits. The matching percentage between Cases E and F may be high because of potential reused IP between these designs.

Weighted Matching of Partial vs. Full Designs						
Partial Designs	Full Designs					
	A	B	C	D	E	F
A	32.6%	8.24%	6.41%	7.21%	6.28%	6.28%
B	5.95%	14.6%	4.80%	6.46%	4.59%	4.59%
C	4.03%	4.03%	19.3%	4.40%	3.98%	3.98%
D	11.2%	12.7%	10.8%	23.6%	10.6%	10.6%
E	6.46%	6.50%	6.43%	6.41%	13.6%	10.1%
F	5.49%	5.63%	5.45%	5.44%	7.13%	15.8%

Table 3: Percentage of matching between partial design and full design with weighted sum of the credits. Each entry is an average over three experimental trials.

4 Conclusions and Ongoing Research

In conclusion, we have given the first study of copy detection techniques for VLSI CAD. Our methods complement the previous watermarking-based IP protection literature. Our generic methodology for copy detection is based on determining basic *elements* within structural representations of solutions (IPs), calculating *locally context-dependent* signatures for such elements, and performing fast comparisons to filter potential violators of IP rights. Example implementations in the scheduling, graph coloring and gate-level layout domains show the effectiveness and low implementation overhead of our methods. Ongoing work addresses improvements to scalability of our methods, as well as automatic determination of thresholds for suspicion of copying. Open areas for research

nected components, etc. are all coarse but potentially effective comparisons; checking isomorphism is a detailed comparison.

[6] Even if some sequences are the same, this does not mean that the netlists are isomorphic. However, the procedure will leave only a few candidates for stolen IP fragments, and these can be checked in essentially linear time. Vertices can also be annotated with information (logic type, hierarchy level, etc.) to induce corresponding marked degree sequences, as discussed below – again, this is to produce a staged "filtering" before applying detailed isomorphism tests.

[7] We have also considered copy detection in polygon layouts that may have been exposed to migration and compaction tools during copying. We initially filter macros by signatures according to simple attributes (number of features per layer, size, etc.). A second filter (before actual isomorphism checking) uses vertex signatures in "conflict graphs" defined over features in the layout; in a conflict graph, the number of vertices equals the number of layout features, and there is an edge between vertices if corresponding features are within distance d of each other (varying d induces a family of such graphs). When d is significantly larger than the minimum feature size/spacing, then slight changes in layout will not affect the conflict graph.

[8] Note that in Tables 2 and 3, there was a big difference between matching of Case E and Case F and matching between any other case and Case F. Larger IPs will tend to afford better distinction between copied IPs and non-copied IPs, as seen by comparing the two Tables.

include: (i) the potentially deep and complementary interaction between watermarking and copy detection, e.g., if watermarks can be introduced to facilitate copy detection; (ii) development of copy detection methods that are more immune to topology changes such as buffering, fanout clustering, complementation, etc.; and (iii) development of automated techniques to trace the "genealogy" (genes and ancestors) of given pieces of design IP, much along the lines of phylogenetic trees in bioinformatics.

References

[1] A.V. Aho, "Algorithms for Finding Patterns in Strings", *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), 1990.

[2] "The GNU awk program", Available by anonymous FTP from `prep.ai.mit.edu`.

[3] G. Benson, "An Algorithm for Finding Tandem Repeats of Unspecified Pattern Size", *Proc. RECOMB98 Second Annual International Conference on Computational Molecular Biology* (S. Istrail, P. Pevzner, M. Waterman, eds.), 1998, p. 20-29.

[4] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm", *Communications of the ACM* 20(10), 1977, pp. 762-772.

[5] S. Brin, J. Davis and H. Garcia-Molina, "Copy Detection Mechanisms for Digital Documents", *Proc. ACM SIGMOD International Conference on Management of Data*, (*SIGMOD Record*) 24(2), 1995, pp. 398-409.

[6] K.-W. Chiang, S. Nahar and C.-Y. Lo, "Time-Efficient VLSI Artwork Analysis Algorithms in GOALIE2", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8(6), 1989, pp. 640-648.

[7] B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling", *SIGMETRICS Conference on Measurement and Modeling of Computer Systems* 22(1), 1994, pp.128-37.

[8] C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", *Symposium on Principles of Programming Languages*, 1998, pp. 184-196,

[9] M. R. Corazao, M. A. Khalaf, L.M. Guerra, M. Potkonjak and others, "Performance Optimization Using Template Mapping for Datapath-Intensive High-Level Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15(8), 1996, pp. 877-888.

[10] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani and J. Ullman, "Computing Iceberg Queries Efficiently", *Proc. International Conference on Very Large Databases*, New York, August 1998.

[11] D. A. Forsyth and M. M. Fleck, "Finding People and Animals by Guided Assembly", *Proc. International Conference on Image Processing*, 1997, vol. 3 pp. 5-8.

[12] S. Grier, "A Tool that Detects Plagiarism in PASCAL Programs", (12th SIGCSE Technical Symposium on Computer Science Education, St. Louis, Feb. 1981), *SIGCSE Bulletin* 13(1), 1981, pp. 15-20.

[13] M. Haertel, et al,"The GNU diff program", Available by anonymous FTP from `prep.ai.mit.edu`, 1999.

[14] P. Indyk, R. Motwani, P. Raghavan and S. Vempala, "Locality-Preserving Hashing in Multidimensional Spaces", *Proc. 29th ACM Symposium on the Theory of Computing*, 1997.

[15] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching", *Proc. ACM/IEEE Design Automation Conference*, 1987, pp. 341-347.

[16] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik and others, "Watermarking Techniques for Intellectual Property Protection", *Proc. ACM/IEEE Design and Automation Conference*, 1998, pp. 776-781.

[17] R.M. Karp and M.O. Rabin, "Efficient randomized pattern-matching algorithms", *Technical Report* TR-31-81, Aiken Computation Laboratory, Harvard, 1981.

[18] D. E. Knuth, J. H. Morris and V. R. Pratt, "Fast Pattern Matching in Strings", *SIAM Journal on Computing* 6(2), 1977, pp. 323-350.

[19] R. A. Krutar, "Conversational Systems Programming (or Program Plagiarism Made Easy)", *Proc. 1st USA-Japan Computer Conference*, Oct. 1972, pp. 654-661.

[20] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. International Symposium on Microarchitecture, pp.330-5, 1997.

[21] U. Manber, "Finding Similar Files in a Large File System", *Proc. Winter USENIX Conference*, 1994, pp. 1-10.

[22] M. Niewczas, W. Maly and A. Strojwas, "A Pattern Matching Algorithm for Verification and Analysis of Very Large IC Layouts", *Proc. International Symposium on Physical Design*, 1998, pp. 129-134.

[23] M. M. Novak, *Correlations in Computer Programs*, *Fractals* 6(2), 1998, pp. 131-138.

[24] M. Ohlrich, C. Ebeling, E. Ginting and L. Sather, "SubGemini: Identifying SubCircuits Using a Fast Subgraph Isomorphism Algorithm", *Proc. ACM/IEEE Design Automation Conference*, 1993, pp. 31-37.

[25] A. Parker and J. O. Hamblen, "Computer Algorithms for Plagiarism Detection", *IEEE Transactions on Education* 32(2), 1989, pp. 94-99.

[26] P. G. Salmon and R. J. Tracy, "Computer-Generated Computation Exercises", *Behavior Research Methods and Instrumentation* 7(3), 1975, p. 307.

[27] N. Shivakumar and H. Garcia-Molina, "Building a Scalable and Accurate Copy Detection Mechanism", *Proc. 1st ACM International Conference on Digital Libraries*, 1996, pp. 160-168.

[28] S. Singhe and F. J. Tweedie, "Neural Networks and Disputed Authorship: New Challenges", *Proc. International Conference on Artificial Neural Networks*, London, 1995, pp. 24-28.

[29] K. L. Verco and M. J. Wise, "Plagiarism a la Mode: a Comparison of Automated Systems for Detecting Suspected Plagiarism", *Computer Journal* 39(9), 1996, pp. 741-750.