Engineering a Scalable Boolean Matching Based on EDA SaaS 2.0*

Chun Zhang
State Key Lab of ASIC and System, Fudan University

Yu Hu
Electrical and Computer Engineering Department, University of Alberta

Lingli Wang
State Key Lab of ASIC and System, Fudan University

Lei He
Electrical Engineering Department, UCLA

Jiarong Tong
State Key Lab of ASIC and System, Fudan University

ABSTRACT

Software as a Service (SaaS) 1.0 signifcantly lowers the infrastructure and maintenance cost and increases the accessibility of the software by hosting software via the web. Compared with SaaS 1.0, SaaS 2.0 is more flexible since it leverages software tools from both server and client sides with closer interaction between them. The SaaS 2.0 paradigm provides new opportunities and challenges for EDA. In this paper, we take Boolean matching, one of the core sub algorithms in logic synthesis for field programmable gate arrays (FPGAs), as a case study. We investigate the advantages and challenges of implementing a scalable EDA algorithm under SaaS 2.0 paradigm from a technical perspective. We propose SaaS-BM, a new Boolean matching algorithm customized to take full advantage of the cloud while addressing concerns such as security and the internet bandwidth limit. Extensive experiments are performed under a networked environment with concurrent accesses. Integrated into a post-mapping re-synthesis algorithm minimizing area, the proposed SaaS-BM is 863X times faster than state-of-the-art SAT-based Boolean matching with 0.5% area overhead. Compared with a recent Bloom Filter-based Boolean matching algorithm, our proposed SaaS-BM is 53X times faster on large circuits with no area overhead.

Categories and Subject Descriptors

B.6.3 [**Hardware**]: Design Aids—*Automatic Synthesis*; C.2.4 [**Computer Systems Organization**]: Distributed Systems—*Client/Server*

General Terms

Algorithms, Design, Experimentation, Performance

*This work is partially funded by National 863 Program of China (2009AA012201), Shanghai Pujiang Program 2008, NSERC Discovery Grant and ISTP Canada. Address comments to `llwang@fudan.edu.cn` and `lhe@ee.ucla.edu`.

Keywords

Boolean Matching, Electronic Design Automation (EDA), Cloud Computing, Software as a Service, Field Programmable Gate Array (FPGA)

1. INTRODUCTION

With the cloud computing era approaching, the software industry has encountered disruptive changes. On top of the cloud computing model, software-as-a-service (SaaS) [1], a new model of software development, has been adopted in the electronic design automation (EDA) industry [2], pioneered by companies including Xuropa and PDTi [3]. The SaaS model allows the software vendors to develop, host and operate software for customer use. Rather than purchase the hardware and software to run an application, customers need only a computer or a server to download the application and internet access to run the software. For EDA users, SaaS greatly reduces maintenance and infrastructure costs.

In the SaaS 1.0 paradigm, through internet the vendor provides systematic, secure remote access to an EDA tool without any installation, setup or teardown by the user. The tool is available on demand and provides for an ideal environment within which to carry out many of the tool adoption phases. In the past three years, the SaaS 1.0 model has evolved to SaaS 2.0, where a vendor provides the tool engine (as opposed to an entire tool) through web service and makes the server-side tool and clients more interactive. For example, Think Silicon provides $IPGenius^{TM}$ [4] as a web service to generate IP cores from different configurations.

The paradigm shift to SaaS 2.0 provides new opportunities and also poses new challenges to researchers of EDA community. The entire conventional EDA flow needs to be examined in order to take full advantage of SaaS 2.0. In this paper, we perform a case study on the implementation of a SaaS 2.0-based Boolean matching for FPGA synthesis to investigate the effectiveness of such a paradigm shift in logic synthesis.

Boolean matching is a widely used technique in field programmable gate array (FPGA) technology mapping, post-mapping re-synthesis and architecture evaluations. As it is called many times as a sub-routine in those applications, Boolean matching has been shown to be one of the most time-consuming sub-problems in combinational logic synthesis [12]. In this paper, we proposed SaaS-BM, a new Boolean matching algorithm deployed under the SaaS 2.0

paradigm, which is hundreds of times faster than the state-of-the-art SAT-based Boolean matchers [14] for FPGAs in the literature.

To verify the effectiveness of the proposed SaaS-BM, we integrate it into a post-mapping re-synthesis [15] application minimizing area. Tested on a collection of three different benchmark sets, the re-synthesizer geared with the proposed SaaS-BM under 100 client accesses is 863X times faster than the one with the SAT-based Boolean matching [14], with only 0.5% more area (in terms of number of LUTs). Compared with a recently proposed Filter-based Boolean matching [17], SaaS-BM servicing 100 clients concurrently is 53X times faster for the largest benchmark set circuits with no area overhead.

The reminder of the paper is organized as follows. Section 2 presents preliminaries. Section 3 depicts the main framework of the proposed SaaS-based Boolean matcher. Section 4 describes implementation details of the scalable Boolean matching at server side, and section 5 analyzes the performance issues under a networked environment. In section 6, we present experimental results by integrating SaaS-BM into a post-mapping re-synthesis algorithm minimizing area. Section 7 concludes the paper.

2. PRELIMINARIES

2.1 Boolean Matching

For a programmable logic block (PLB) H that consists of various configurable devices (e.g., LUTs or macro-gates), and a Boolean function f, Boolean matching (BM) either finds a set of configurations for H to implement f, or concludes that H cannot implement f. Ideally, an FPGA Boolean matcher should be scalable to large Boolean functions and complex PLB structures in terms of both runtime and memory, and be flexible with regard to reusability across different PLB structures.

Most of the existing Boolean matching algorithms are based on function decomposition [16] or on canonicity and Boolean signatures [11]. However, the function decomposition technique lacks flexibility and needs to be customized for different PLB architectures, and canonicity-based approaches can only handle functions of limited input size. Due to significant improvement of the modern SAT solvers, SAT-based Boolean matching [15] has also been proposed. While SAT-based Boolean matching (SAT-BM) offers great flexibility in handling various PLB architectures, the computational complexity prevents its applications to large PLBs, even with numerous improvements [14].

Recently, a Filter-based Boolean matching technique was proposed [17]. Instead of computing the Boolean matching on-the-fly, the results are pre-computed and stored in a library, i.e., a lookup table. Bloom Filter is used to implement the library. However, it has two main disadvantages. First, due to its probabilistic nature, Bloom Filter has false positives (e.g., a function can be falsely claimed in the library while it is actually not). Secondly, function implementations (e.g., LUT configurations) are not kept in the library. In either case, an explicit Boolean matching computation is still needed to double-check a function's existence and to generate correct implementations.

3. OVERVIEW OF SAAS-BM

The proposed SaaS-BM is based on the traditional client/server model. Instead of performing the Boolean matching computation at each EDA user's local machine, this time-consuming operation is provided as a *service* at the servers provided by the EDA tool vendors.

On the server side, a scalable Boolean matcher is customized for the cloud computing paradigm where the storage is a relatively cheap resource. Instead of performing the computation of each Boolean matching instance on-the-fly, we use a table lookup-based approach similar to [17]. Matching results are pre-computed and stored in a key/value database, which is indexed by a Boolean function and each function is associated with its implementation (i.e., configurations of programmable devices). When the server receives a Boolean matching request for a given function from the client, a database query is performed. If the function is found in the database, the corresponding implementation associated with this function is returned to the client. Otherwise, a *null* is returned indicating the given function is not implementable.

From the client's perspective, the Boolean matching is simplified to a request and reply interaction with server clusters through Internet. Once a Boolean matching instance is formed locally during any EDA application, client will send the request and then wait for the response from the server. To improve performance, the user-end contains a *cache* implemented by Bloom Filter [17]. As the Boolean matching operation is encapsulated as a black-box operation performed remotely, minor changes on existing EDA tool flows are required.

The proposed SaaS-BM has the following attributes, which make it particularly suited to be deployed in the cloud.

- On the server side, as the number of Boolean matching request increases, the efficiency of the database query increases due to the cache effect of the key/value database.
- Our SaaS-BM addresses the security concern in the design outsourcing. To use our SaaS-BM service, the end-users of the EDA tool do not need to expose their entire designs to the cloud. Instead, the client-side application decomposes the problem into Boolean functions, from which a complete design is hard to reverse-engineer.
- SaaS-BM is scalable to large-scale clouds due to the simplicity of the data structures in the key/value database.

4. BOOLEAN MATCHING AT SERVER SIDE

4.1 Selection of Storage Medium

Database provides a uniform and efficient method for management of massive data sets. Over the past 30 years, relational databases have dominated the data storage and management application domains. While relational databases scale well on a single server node, their complexity becomes overwhelming when one tries scaling to hundreds or thousands of nodes in the cloud [5].

In SaaS-BM, we propose to build the server-side lookup table with the key/value database techniqdue [6][5]. Unlike a relational database, schemas and relationships between tables are not explicitly defined in a key/value database, and therefore it is more flexible when scaling to larger number of server nodes. Both functionalities (i.e., *key*) and their implementations (i.e., *value*) for frequent Boolean functions extracted from benchmark circuits are pre-calculated, stored

Table 1: Library and Testing Set

Library Set	apex2, des, ex1010, pdc, spla clma, elliptic, frisc, s38417, s38584
Testing Set	alu4, apex4, bigkey, diffeq, dsip ex5p, misex3, s298, seq, tseng

and indexed in this database. In this way, all SAT-checking time is eliminated, and the matching time is only limited by database querying performance.

In our application, we choose Berkeley DB (BDB) [6], an efficient open-source implementation of the key/value database. Besides the aforementioned scalability of the key/value database in the cloud, we favor it over the relational databases due to the underlying property of our data. Specifically, we have simple data relationships (i.e., functions with corresponding implementations) and stable query requests (i.e., lookup implementations for given functions), which match directly with the BDB's design perspective. By organizing data in a single two-column table implemented with basic data structures such as hash-table or balanced search tree, BDB provides users with extremely high performance and the flexibility to tune for different applications. In addition, BDB is capable of managing giga-bytes to tera-bytes of data, while the amount of data which can be kept is only limited by hardware.

4.2 Generation of Library

Instead of performing a brute-force enumeration for all K-input functions, which is impossible for large functions (e.g., 9-input functions), we propose to generate the matching library using frequently appeared functions extracted from a selected group of benchmark circuits. As will be shown in Table 2, *Boolean functions in real circuits exhibit repetitivity of occurrence across different benchmark circuits, and therefore we can apply a matching library extracted from one set of circuits to new circuits.*

In our experiments, the functions are extracted from 10 MCNC benchmark circuits, which is called the *library set* in Table 1. To test the effectiveness of our library, functions extracted from another 10 circuits in MCNC benchmark circuits (*testing set*) are checked for their existence against the library.

The detail library generation process works similar to the one proposed in [17]. For frequently appeared functions with up to 9 inputs extracted from the library set circuits using the ABC tool [7], their implementability is checked against certain PLBs using SAT-BM [14], and corresponding matching results (e.g., LUT configurations) are stored into database. Note that in our experiment, we adopt PLB structures shown in Figure 1, as they've been proven to be effective in reducing area at post-mapping re-synthesis [15].

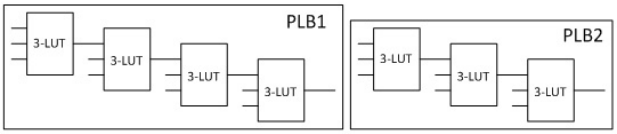

Figure 1: PLB structures used for library generation

Top 2-million frequently appearing functions are checked by SAT-BM [14]. For each function, we keep up to 10,000 of its input permutations into the library as well. Overall, there're 3 billion records (i.e., functions with corresponding implementations) stored in the library, with the size of 250 GigaBytes. It takes about two weeks to generate the library on a Linux server with Quad-Core Intel Xeon 2.33GHz CPU. Note that other Boolean matching (other than SAT-BM) algorithms, e.g., [16], can also be employed to perform the library generation for homogeneous PLBs.

Table 2: Coverage of the generated library. I-hit stands for implementable functions found in the library, so is the case of N-hit for non-implementable functions. I-miss and N-miss lists the number of functions not found in the library.

	Functions		
Type	7-input	8-input	9-input
I-hit	80,460	69,886	48,048
I-miss	2,219	3,465	4,851
I-Coverage	97.3%	95.3%	90.8%
N-hit	11,621	15,680	20,022
N-miss	5,700	10,969	27,079
N-coverage	67.1%	58.8%	42.5%

Table 2 illustrates the effectiveness of the SAT-building method, where the coverage for 100,000 functions randomly extracted from *testing set* circuits are tested. A function is said to be covered if its matching result can be found in the library[1]. As shown in Table 2, over 90% of implementable functions are covered. Following these observations, the Boolean matching problem on the server side can be reduced as follows: *Given a Boolean function, we query its existence in the database. If it is found, the correct implementation is fetched; otherwise, we can regard it as non-implementable without significant quality degradation on existing EDA applications.*

4.3 Key Selection and Compression

The length of the representation for a Boolean function has a tremendous impact on database query performance, as it is used as the key to index the data record. In our experiment, we compare two commonly used representations of a Boolean function, i.e., truth-table and BDD. Interestingly, we found that for functions extracted from library set circuits, the length of BDD representation grows slowly than the truth-table representation. In addition, there exist specialized BDD compression algorithms [13], where each BDD node can be represented using only 1 to 2 bits. As a result, we decide to choose BDD as function representation in our method.

In our library, BDD is dumped out into *Buddy* format [8], where 4 integers (i.e., node-id, node-level, low-edge and high-edge) are needed to describe each BDD node. In most cases only one byte is needed for each integer in binary format, since normally there are less than 256 nodes in the diagram. To further reduce the space requirement, we compress the dumped output with a common compression package – Zlib [9]. Note that we can choose different compression methods depending on the compression time and query performance trade-off. Table 3 shows the performance of the library built in section 4.2. The second column presents compressed key size[2]. The third column of Table 3 lists the library query

[1]While actually only implementable functions for given PLBs need to be kept, however, for the purpose of completeness, the coverage for non-implementable functions are also tested in this experiment.

[2]Although truth-table representation have smaller size for functions with less than 9 inputs, we keep using BDD because it accommodates better to larger functions, which

Table 3: Performance of library queries. Query time is calculated as by averaging the speed of 100,000 functions randomly selected from testing set

	Compressed Size (bytes)	Query time (us)
2-input	22.04	8.58
3-input	29.52	10.04
4-input	35.99	10.29
5-input	43.33	11.21
6-input	51.80	11.57
7-input	62.90	11.58
8-input	76.72	11.77
9-input	93.59	11.97

speed, which is the time we need to perform Boolean matching. As is shown, it is much faster compared to SAT-BM where several seconds are needed to match one 9-input function.

5. BM IN NETWORKED ENVIRONMENT

In order to respond multiple client requests at the same time, the server must provide competitive performance for concurrent access to the database. In BDB [6], a cache is provided to keep newly and frequently appearing queries in memory. Since in real circuits a small set of common functions holds a large percentage of all functions that may appear, such a cache mechanism greatly improves the concurrent query performance by avoiding the slow disk I/O operations. Figure 2 illustrates the impact of BDB cache on query performance. As we increase the number of database queries, the average time for each query decreases by more than two factors of magnitude. Note that the average query time converges to a stable range (i.e., several micro-seconds) when the query number is large enough. In real cases, since the server shall handle much more queries than is shown in Figure 2, we can expect it to be the actual performance for database query.

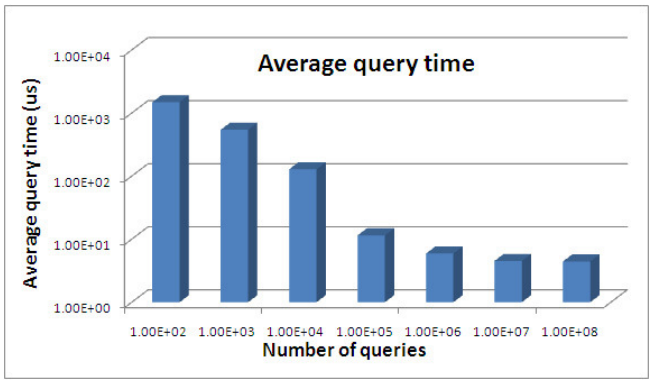

Figure 2: The Impact of BDB cache. The performance is tested under single thread querying the database.

To precisely test the effectiveness of the proposed SaaS-BM paradigm, we build a network environment to simulate its performance, where up to 1,000 clients are concurrently sending matching requests. At the server side, the server keeps listening at a given port for the client's connection request in the main process. Whenever one connection arrives, a separate thread is spawned to handle communications with that client using POSIX pthread library. The connection is established under TCP/IP protocol, which guarantees no data loss through the network. Note that since the database works in a read-only mode during Boolean matching, we disable the LOCK mechanism in BDB to improve performance. To represent clients, we generate multiple processes sending random matching requests on another machine[3].

Table 4 compares the matching time when different number of clients are accessing concurrently. Note that the matching time is only 8 ms even when 1,000 clients are connected at the same time.

Table 4: Average Boolean matching time for 9-input functions under network environment, including both network data transferring and database querying time

Number of users	Matching time (ms)	Slowdown
1	0.01	1x
10	0.04	4x
100	0.64	64x
1000	8.64	864x

Figure 3 plots the average matching time for different clients. Results show that the performance differs in a small range, indicating that different clients get matching service with almost the same quality.

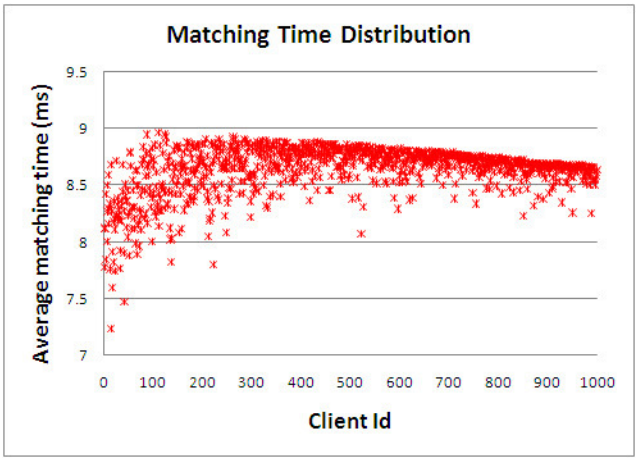

Figure 3: Distribution of query time among different clients

The above results were collected using a hardware infrastructure consisting of a 5400/RPM hard-disk and a 100Mbps network interface card. We believe that the concurrent access performance can be significantly improved in the real cloud, where more advanced hardware is available and the servers are carefully configured (e.g., the cache size and the thread management).

5.1 Using Bloom Filter as Cache at Client Side

The experiment in section 5 is carried out inside a 100Mbps LAN. However, it's more realistic that the client needs to access the server through a public network (e.g., Internet). In that case, we need to take network latency into consideration. With modern broadband network technologies, it is common to achieve a 1Mbps connection speed. As we need

makes it easy to extend our library in the future.

[3]Each client is set with a small interval between two continuous Boolean matching request, to represent the time to perform other operations in real EDA applications

less than 100 bytes to represent the implementation of a 9-input Boolean function, it roughly takes 0.8 *ms* to transfer the data.

To address the problem incurred by network latency, we propose to set up a *cache* at client side, where the most common non-implementable functions against PLBs in Figure 1 are kept. Instead of sending remote request to server for every Boolean function, it is first checked for its existence in the cache. Once found, it is regarded as non-implementable, thus the explicit check on server side is eliminated.

Bloom Filter [17] serves well to build the local cache. A Bloom Filter is a space efficient probabilistic data structure to test for an element's membership in a set. When properly configured, it has several advantages including: (1) constant checking time; (2) low false positive rate; (3) extremely low space cost to store one element. Specifically, with 1% false positive rate, we need only 9.6 bits to store one arbitrary element, regardless of the actual size of the element itself.

In our experiment, the local cache in our experiment keeps the non-implementable functions found during library generation described in section 4.2 and takes a memory space of 2GBs. As is shown in Table 2, over 40% 9-input non-implementable functions are covered. In other words, we can avoid about half of the matching at server side for non-implementable function using such a cache. As function lookup in the local cache costs only about 4 *us*, as a result, tremendous speedup can be achieved.

6. EXPERIMENT RESULTS

To show the effectiveness of the proposed SaaS-BM, a post-mapping re-synthesis minimizing area (i.e., LUT number) [15] is adopted as an application for Boolean matching. The re-synthesis procedure works in a greedy mode, which takes a circuit mapped to 3-LUTs (mapped by ABC [7]) and scans the combinational portion of the circuit in a topological order. During the scanning, new logic blocks are generated by enumerating and combining the logic blocks at inputs of a LUT, and each logic block is checked for its implementability against PLB1 and PLB2 (shown in Figure 1). When an implementable case is found by the Boolean matcher, the logic block is replaced by the corresponding PLB structure if such a replacement reduces the number of LUTs without increasing the logic depth. The algorithm terminates after several iterations (e.g., set by user) of full scan of all LUTs, or until no LUT can be further reduced.

Algorithm 1 Resynthesis-one-iteration(*network*)

```
1: for all node of network in topological order do
2:   cutset = enumerateKfeasibleCut(node)
3:   for all cut in cutset do
4:     for all PLB H in PLB library do
5:       if |cut| ≥ |H| then
6:         continue {No area reduction}
7:       end if
8:       impl = booleanMatching(cut, H)
9:       if impl ≠ NULL then
10:        updateNetwork(cut, H)
11:      end if
12:    end for
13:  end for
14: end for
```

The pseudo code of the re-synthesis procedure is shown in Algorithm 1. For the Boolean matching sub-routine (i.e., function **booleanMatching** called in line 8), three algorithms are implemented: single-client SAT-BM [14], single-client F-BM [17] and SaaS-BM with 100 concurrent clients' access.

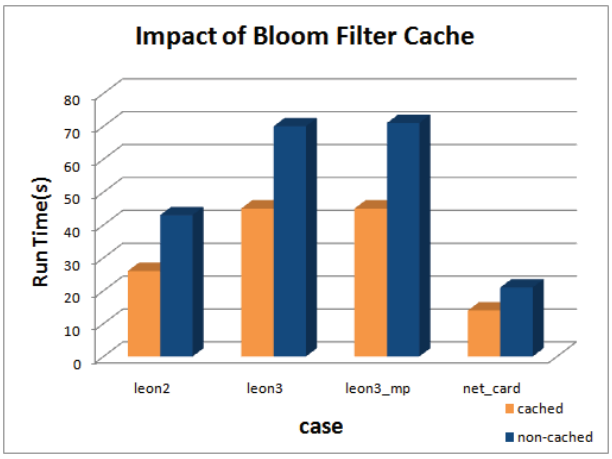

Figure 4: Impact of Bloom Filter Cache

Table 5 compares the run-time[4] and quality of these three matchers. To further explore the effectiveness of the library, two other benchmark sets (IWLS 2005 [10] and Industrial designs) are also tested. Separate comparisons are listed for each benchmark set, while the last two rows average over all three benchmark sets. The *Ratio* in this table is computed as geometric mean. As is shown in the table, the re-synthesizer geared with SaaS-BM is 863X times faster than the one with SAT-BM over three benchmark sets, and with only 0.5% area overhead. Note that we achieve better speedup for large circuits, i.e., for IWLS benchmark circuits, SaaS-BM-based re-synthesizer is 53X times faster than F-BM-based one, with the same area. Considering that for SaaS-BM is capable of performing 100 concurrent Boolean matching operations, this speedup is tremendous.

To further analyze the effect of the Bloom Filter based cache, we compare the run-time with and without such cache. Figure 4 shows the run-time is reduced by 37% when Bloom Filter is applied as cache at client's local machine. Such improvement comes from the fact that, during the re-synthesis, many more non-implementable functions than implementable cases are found (shown in Table 6), so the local cache allows us to quickly prune non-implementable functions without the need for network communications to the server.

Table 6: Number of implementable and non-implementable functions found during re-synthesis

Circuit	Implementable	Non-implementable
leon2	6374	26006
leon3	10202	41020
leon3_mp	6606	42281
netcard	5330	11637

7. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a network based scalable Boolean matching method SaaS-BM. The method changes traditional EDA flows, by constructing a new client/server network service model. Incorporated into a post-mapping re-synthesizer

[4]This is the time to perform Boolean matching, including public network latency

Table 5: Comparison between SAT, Bloom Filter and SaaS based Boolean Matchers

		Runtime (s)			Reduced LUT #			Total LUT #		
		SAT-BM	F-BM	SaaS-BM	SAT-BM	F-BM	SaaS-BM	SAT-BM	F-BM	SaaS-BM
MCNC	alu4	246	7.86	0.16	14	14	14	1742	1742	1742
	diffeq	392	0.05	0.10	4	3	3	1344	1345	1345
	ex5p	0.02	0.02	0.001	1	1	1	1105	1105	1105
	s298	8.89	0.07	0.10	7	7	7	1291	1291	1291
	seq	3.93	0.02	0.003	1	1	1	1522	1522	1522
	Ratio to SAT-BM	1x	1/58x	1/426x	1x	0.94x	0.94x	1x	1.000x	1.000x
	Ratio to F-BM	–	1x	1/7x	–	1x	1x	–	1x	1.000x
Industrial	Ex1	11982	101	18	1721	1139	1139	19105	19687	19687
	Ex2	1578	26	1.84	305	271	271	6326	6360	6360
	Ex3	32156	24	14	674	549	549	7502	7627	7627
	Ex4	10280	86	7.84	669	513	513	15272	15428	15428
	Ex5	671	6.66	0.73	93	79	79	2873	2887	2887
	Ratio to SAT-BM	1x	1/163x	1/1096x	1x	0.79x	0.79x	1x	1.013x	1.013x
	Ratio to F-BM	–	1x	1/7x	–	1x	1x	–	1x	1.000x
IWLS	leon2	63834	1771	26	6673	6456	6456	374701	374918	374918
	leon3	60959	1976	45	10770	10395	10395	566134	566509	566509
	leon3mp	46063	1382	45	6941	6636	6636	341309	341614	341614
	netcard	23626	1342	14	5566	5435	5435	336334	336465	336465
	Ratio to SAT-BM	1x	1/29x	1/1549x	1x	0.97x	0.97x	1x	1.001x	1.001x
	Ratio to F-BM	–	1x	1/53x	–	1x	1x	–	1x	1.000x
Ratio to SAT-BM		1x	1/80x	1/863x	1x	0.89x	0.89x	1x	1.005x	1.005x
Ratio to F-BM		–	1x	1/11x	–	1x	1x	–	1x	1.000x

reducing area, our method is 863X times faster while providing 100 concurrent Boolean matching services and with only 0.5% more area, against the one using with an optimized SAT-BM which performs one single Boolean matching at a time. Compared to the recent single-client Bloom Filter-based Boolean matching, our approach under 100 concurrent client access is still 53X times faster with the same area on large benchmark circuits.

In the future, we plan to optimize the configurations at both client and server side to get better performance. In addition, we'll target more realistic hardware settings (e.g., multiple servers). Thirdly, we'll incorporate the SaaS-BM into other EDA applications.

8. REFERENCES

[1] Software As A Service, http://en.wikipedia.org /wiki/Software_as_a_service.

[2] Cloud Computing, SaaS and Electronic Design, http://www.xuropa.com/blog/2008/12/12/cloud-computing-saas-and-electronic-design-part-4/.

[3] PDTi, http://www.productive-eda.com/.

[4] IP Genius, http://www.ipgeniuscores.com /ipgenius.php.

[5] http://www.readwriteweb.com/enterprise/2009/02/is-the-relational-database-doomed.php.

[6] Berkeley DB, http://www.oracle.com/technology /products/berkeley-db/index.html.

[7] ABC, http://www.eecs.berkeley.edu/ alanmi/abc/.

[8] Buddy, http://sourceforge.net/projects/buddy/.

[9] Zlib, http://www.zlib.net/.

[10] IWLS 2005 benchmark, http://iwls.org/iwls2005 /benchmarks.html.

[11] A. Abdollahi and M. Pedram. A new canonical form for fast Boolean matching in logic synthesis and verification. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 379–384, New York, NY, USA, 2005. ACM.

[12] J. J. Cong and Y.-Y. Hwang. Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20:1077–1090, 2001.

[13] E. R. Hansen, S. S. Rao, and P. Tiedemann. Compressing Binary Decision Diagrams. In *Proceeding of the 2008 Conference on ECAI 2008*, pages 799–800, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[14] Y. Hu, V. Shih, R. Majumdar, and L. He. Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 350–353, Piscataway, NJ, USA, 2007. IEEE Press.

[15] A. Ling, D. P. Singh, and S. D. Brown. FPGA technology mapping: a study of optimality. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 427–432, New York, NY, USA, 2005. ACM.

[16] A. Mishchenko, R. Brayton, and S. Chatterjee. Boolean factoring and decomposition of logic networks. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 38–44, Piscataway, NJ, USA, 2008. IEEE Press.

[17] C. Zhang, Y. Hu, L. Wang, L. He, and J. Tong. Building a faster Boolean matcher using Bloom Filter. In *FPGA '10: Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 185–188, New York, NY, USA, 2010. ACM.