Improving Bit-Vector Representation of Points-To Sets Using Class Hierarchy

Hamid A. Toussi and Ahmed Khademzadeh

Abstract—**Points-to analysis is the problem of approximating run-time values of pointers statically or at compile-time. Points-to sets are used to store the approximated values of pointers during points-to analysis. Memory usage and running time limit the ability of points-to analysis to analyze large programs.**

To our knowledge, works which have implemented a bit-vector representation of points-to sets so far, allocates bits for each pointer without considering pointer's type. By considering the type, we are able to allocate bits only for a subset of all abstract objects which are of compatible type with the pointer's type and as a consequence improve the memory usage and running time. To achieve this goal, we number abstract objects in a way that all the abstract objects of a type and all of its sub-types are consecutive in order.

Our most efficient implementation uses about 2.5× less memory than hybrid points-to set (default points-to set in Spark) and also improves the analysis time for sufficiently large programs.

Index Terms—**Programming languages, points-to analysis, points-to sets, data structures, bit-vectors, class hierarchy, Java.**

I. INTRODUCTION

Points-to analysis is the problem of approximating run-time values of pointers statically or at compile-time. The result of this analysis may be used for program optimization, debugging or understanding. One of the most famous and widely used algorithms to solve this problem, which we have used in our implementations, is Andersen style or inclusion based points-to analysis [1].

Points-to sets are used to store the approximated values of pointers during the analysis. Previous works have implemented efficient and compact points-to sets which make the overall analysis more efficient [2]-[8]. Points-to sets which are represented based on bit-vectors [4], [5], [9] and points-to sets which are represented and manipulated based on BDD relations are shown to be more efficient [2], [3], [6], [7], [8], [10]. In this paper we utilize bit-vector representation of points-to sets.

Points-to analysis produces several points-to sets that are similar in the sense that many of their members are common [5]. Based on this fact, both BDD based and bit-vector based methods try to have more efficient representation of points-to

sets by trying to share common members.

In a pure bit-vector implementation of points-to sets, one bit is allocated for every abstract object (allocation-site) that is reachable from root methods. Reachability of allocation-sites can be determined using a conservative call-graph. This call-graph can be created using either CHA [11], RTA [12] or VTA [13] methods.

Heintze have implemented a shared bit-vector representation of points-to sets in order to improve memory and time efficiency [5]. These sets consist of two parts, a shared bit-vector (base part) and an overflow list. The base part is shared among two or more points-to sets and the overflow lists is maintained to include 20 or fewer members. This representation benefits from the fact that many points-to sets are similar. Hirzel also used this implementation of points-to sets in his work [9]. Lhotak and Hendren [4] used a variant of this technique. Their set had no mechanism to share common subsets. Once any set gets larger than some specified-size (e.g. 16 members), it becomes a pure bit-vector points-to set.

In strongly typed languages like Java [14] more precise points-to results and, as a consequence, smaller points-to sets can be achieved by using type filtering during points-to set propagation. This method which is known as online type filtering [15] also makes the analysis faster [4].

Previous works, we have mentioned above, haven't made use of types in order to improve size of points-to sets, e.g., in the pure bit-vector implementation of a points-to set associated with a variable of type T, one bit is allocated for every abstract object of the program. By noticing the types, we are able to allocate bits only for a subset of all the abstract objects which are of compatible type with the variable (i.e. abstract objects of type T or its subtypes). To be able to make use of this fact, we reordered abstract objects of programs, using class-hierarchy so as to assign abstract objects of compatible types, successive numbers.

We have a number of implementations and the best of them employs the above-mentioned idea and also uses Spark's hybrid points-to set's idea to gain more efficiency.

There is also another related work, sparse bitmaps, which is employed by GCC [16] and LLVM [17] compilers. Every sparse bitmap consists of a linked list of elements and each element is a bit-vector of size eight words. No element isallocated in a sparse bitmap unless it has at least one non-zero bit set. In order to calculate the memory usage of sparse bitmaps when it is combined with our desired ordering, we did not need to implement a combined version of our method and sparse bitmaps since the memory gain can be calculated by examining bit-vectors after the propagation and finding out how many elements could be saved. This combined version could have some minor memory

improvement compared to our current implementation, but considering the cost of linked list manipulation and minor memory usage improvement, we believe that this memory gain is not worth it (see section VII).

Some background information comes in Section II. As stated above, we would first number the abstract objects in our desired order and then we need to implement a bit-vector (we call it Ranged Bit Vector) which takes these ranges into account. An overall picture of the work is given in Section III. Numbering is explained in Section IV and Ranged Bit Vector is described in Section V. Section VI concludes our implementations and we show our experimental results in Section VII.

II. BACKGROUND

Andersen or inclusion based points-to analysis which is used in this paper, consists of a set of deduction rules. Points-to propagation is the process of applying Andersen's rules repeatedly until reaching a fix-point where all the constraints are satisfied, and as a consequence, a safe solution to the points-to analysis problem is found. This set of constraints is inferred from the input program and are represented by Pointer Assignment Graph (PAG) in Spark.

PAG consists of different nodes and edges. Pointers of the input program are represented by variable nodes. They represent local variables, method parameters and static fields which can hold pointer values. Allocation nodes are abstract objects (i.e. every allocation node abstracts a set of run-time objects) and every allocation node corresponds to an allocation-site of the input program.

Edges mainly represent flow of points-to sets. For example an edge from the variable node v to variable node w shows the statement *w=v* and makes the propagator add all allocation nodes in points to set of v to points-to set of *w*.

There are other edges and nodes in the PAG (store edges, load edges, allocation edges, concrete field nodes and field dereference nodes). See [4], [18] for more details about them.

III. OVERVIEW

Suppose there are 1000 allocation-sites within all the reachable methods of a program and from these, there are only 200 allocation-sites of type A (i.e. instantiation or instantiations of class A at the allocation-site), or one of its sub-types. If we have a variable v of declared type A, and we are supposed to allocate a bit-vector representation of points-to set for it, we can allocate 200 bits (one bit per each allocation-site of type A or its sub-types) instead of 1000 bits. To achieve this, we have to number allocation nodes so that all of the allocation nodes of a declared type, and all of its sub-types, fall within the same interval and be consecutive in order (e.g. the interval [201, 400] in the example above). Recall that for every allocation-site one allocation node is created.

Every non-interface type is mapped to an interval as described above. An interface type may be associated with more than one interval (one interval for every top subclass that has implemented the interface). Every Ranged Bit Vector has an interval and is able to handle different operations.

During points-to set propagation, points-to sets for two kinds of nodes (variable nodes and concrete field nodes. See [4]) are created. For each of these nodes, a specialized bit-vector implementation of points-to set is allocated based on type of the node (i.e. declared type of variables in variable nodes and fields in concrete field nodes).

To allocate the specialized points-to set, the interval or intervals associated with the node's type are required so allocation node manager (the module we added to Spark to manipulate intervals and allocation nodes in our desired way) is queried and type of the node is sent to it as a parameter to achieve these intervals. Finally, the points-to set for the node is allocated. This points-to set has a Ranged Bit Vector for every interval associated with the type. This is the description of Ranged Points-To Set, another variant of it is also implemented in this work (Ranged Hybrid Points-To Set). See Sections VI and V.

IV. NUMBERING

Originally, Spark (the framework we have used to implement our work) numbers allocation nodes as allocates them in the PAG, which is not our desired way of numbering. The procedure in Fig. 1 renumbers the allocation nodes and associates every reference type with an interval. This does not include interface types which may be associated with more than one interval.

The interval's lower bound indicates the number associated with the first allocation node of that type, and its sub-types and interval's upper bound indicates the number associated with the last allocation node within the interval.

```
void dfsVisit(SootClass cl){

  lower = globalCounter + 1;

  for (AllocNode alloc : class2allocs.get(cl)){

    globalCounter = globalCounter + 1;

    globalArray[globalCounter] = alloc;

  }

  // cha is the class hierarchy

  subclasses = cha.getSubClassesOf(cl);

  if (subclasses.isEmpty()){

    // SootClass cl is a leaf in class hierarchy

    type2interval.put(cl.getType(),

            new Interval(lower, globalCounter));

    return;

  }

  for (SootClass c : subclasses)

    dfsVisit(c);

  upper = globalCounter;

  type2interval.put(cl.getType(),

      new Interval(lower, upper));

}
```

Fig. 1. Procedure to renumber allocation nodes and create intervals.

The procedure is basically a depth first traversal of the class hierarchy. The index of an allocation node in theglobalArray represents its corresponding bit in a pure bit-vector implementation. In fact, the globalArray represents the universe of all the allocation nodes.

In this procedure two hash-maps are used. Each reference type is mapped to a linked-list of allocation nodes of that type through the map class2allocs. The map type2interval gets filled during this procedure and it maps every type to its associated interval. Note that globalArray, globalCounter, class2allocs and type2interval are global (static in term of Java). The first invocation of this procedure is dfsVisit(getSootClass("java.lang.Object")). Fig. 2 shows an example class hierarchy and the intervals associated with each type as a result of the invocation.
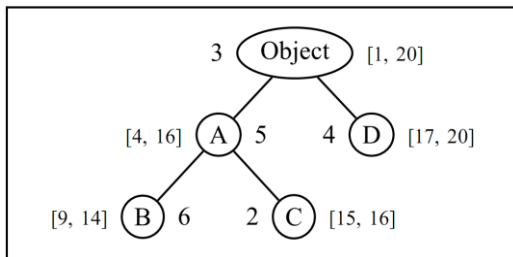

Fig. 2. Each node represents a type in the class hierarchy. The number next to each node shows the number of allocation nodes of that type in the PAG. Intervals associated with each type is created in postorder (i.e. B, C, A, D, Object) and are shown in brackets.

V. RANGED BIT-VECTORS

During the points-to set propagation, many set union operations are performed. For example, an assignment edge from variable node w to variable node v make the propagator add all the members of pt(w) to pt(v) if it is not already satisfied. Note that pt(v) shows the points-to set of variable node v. Considering bit-vectors, this set union operation is implemented using logical or operation.

We have implemented a customized version of bit-vectors which takes the range associated with each bit-vector into account so that the logical or operation can be done efficiently. We have called this implementation of bit-vectors, Ranged Bit Vector. Using the intervals not only saves memory needed for points-to sets, but also helps us to do points-to propagation more efficiently by limiting the size of bit-vectors.

The set union operation pt(v) = pt(v) ∪ pt(w) is actually implemented as:

for (RangedBitVector bv : pt(v).bitvectors)

for (RangedBitVector bw : pt(w).bitvectors)

 or(bv, bw);

In previous works each bit-vector is and with a suitable type mask to enforce type filtering. Type masks are created before the propagation and are used for type filtering during the propagation [9]. The suitable type mask is determined based on type of the points-to set containing the bit-vector. In fact, each type is associated with one type mask. This means, to perform on the fly type filtering [15], one and operation for every set union operation is needed. With our implicit type

filtering mechanism this pass of points-to analysis which was needed to create type masks is eliminated and also we does not need the additional and operation either.

Our method performs type filtering implicitly using the intervals associated with bit-vectors. In another word, an allocation node an with absolute number index is added to bit-vector v if index ≥ v.lower and index ≤ v.upper where v.lower and v.upper are lower and upper bounds of the interval associated with v. In real implementation of our method this assumption is relaxed which will be explained later in this section.

```
boolean or(RangedBitVector x, RangedBitVector y){
  /* The lower bounds are aligned. For example,
     if we are supposed to allocate a ranged
     bitvector of length 10 for the interval
     [10, 20], assuming chunk size is 8, 2 bytes
     will be allocated and the lower bound is
     aligned to 8. */
  if (y.isSubrangeOf(x)){
    /* x is super-range and y is subrange */
    /* Absolute index of lower chunk of the
       subrange */
    L = indexOf(y.lower);
    /* Absolute index of upper chunk of the
       subrange */
    U = indexOf(y.upper);
    /* Relative index of lower chunk of
       the subrange which always is 0 */
    firstChunkY = 0;
    /* Relative index of upper chunk of the
       subrange */
    lastChunkY = U - L;
    /* Absolute index of lower chunk of the
       super-range */
    int temp = indexOf(x.lower);
    /* Relative index of first common chunk
       within super-range */
    firstChunkX = L - temp;
    /* Relative index of last common chunk
       within super-range */
    lastChunkX = firstChunkX + lastChunkY;
  } else if (x.isSubrangeOf(y)) {
    /* y is super-range and x is subrange */
    /* similar to the previous case ... */
  } else {
    return false;
  }
  ret = false;
  for (int i = firstChunkX, j = firstChunkY;
```

```
  i <= lastChunkX; i++, j++) {

    if (!ret) old = x.bits[i];

    x.bits[i] |= y.bits[j];

    if (!ret)

      if (old != x.bits[i]) ret = true;

  }

  return ret;

}


int indexOf(int index) {

  return index / CHUNK_SIZE;

}
```

Fig. 3. The procedure to or 2 ranged bit vectors.

Each Ranged Bit Vector consists of an array of chunks. For example 64-bit chunks (i.e. long[]) or 32-bit chunks (i.e. int[]) in current JVMs. The or operation is implemented by oring corresponding chunks of each bit-vector. To implement Ranged Bit Vector's or operation, we cannot simply or the *i*th chunk of the first bit-vector and *i*th chunk of the second bit-vector since each bit in a Ranged Bit Vector must be interpreted based on its lower bound. Fig. 3 shows Ranged Bit Vector's or operation as we have implemented.

Any two intervals created in the procedure shown in Fig. 1 are disjoint, or one of them is subrange of the other one, so only these cases are handled in Fig. 3. Variable declarations are omitted for saving space. The constant CHUNK_SIZE shows the size of each chunk in the implementation of Ranged Bit Vector.

The overall goal of the procedure shown in Fig. 3 is to add all the members of Ranged Bit Vector y which fall within the interval associated with x, to x and return true if x is changed and false otherwise.

The procedure is conservative in that it aligns the portion which is common to the intervals associated with Ranged Bit Vectors x and y so it can do the or operation chunk-by-chunk (instead of bit-by-bit). This relaxes the assumption we made earlier in this section and as a consequence reduces the precision of type filtering compared to the type filtering performed by using type masking during the propagation. We found this reduction of precision trivial according to the experiment we did. See section VII.

VI. RANGED PONTS-TO SET AND HYBRID RANDED POINTS-TO SET

Mainly, two versions of points-to sets have been added to Spark. One of them is Ranged Points-To Set which has a Ranged Bit Vector for every interval that is associated with its type. Our second major set implementation that is faster and more memory efficient than the first one is inspired from Spark's hybrid points-to set. It contains 16 allocation node references which are initialized to null, whenever all of them are filled up and the set is going to get larger than 16 members, it will become a Ranged Points-To Set.

VII. EXPERIMENTAL RESULTS

We have incorporated our techniques into Spark which is a research framework for points-to analysis. We observed that our work improves the memory allocated to points-to sets by a factor of about 2.5 compared to Spark's hybrid points-to set. It also improves the propagation time for sufficiently large programs compared to Spark's hybrid points-to set which is the fastest and default points-to set in current version of Soot (version 2.4.0) [19].

There is an implementation of Heintze's shared points-to set in Spark which consumes less memory than hybrid's points-to set and our implementation, but it is slower than both of them. See Table I. The results which are shown for Heintze's sets may be smaller than what they really consume since we calculate the results after the propagation but it may take more memory at some point during the propagation.

To evaluate our work we chose 4 programs as follows, jEdit 2.4 (a text editor), JFlex 1.4.3 and SableCC 2.18.2 (both of them are lexical analyzer generators) and Soot 1.2.5 (a framework for analyzing and optimizing Java bytecode).

We used JRE 1.3 as the library to analyze the input programs. All benchmarks are performed on a machine with 2 GB memory (1 GB allocated to JVM) and 2 GHz Intel core 2 Duo CPU running Ubuntu 8.10.

TABLE I: TIME AND SPACE FOR POINTS-TO SET PROPAGATION (SPACE IN MB AND TIME IN SECONDS). COMPARISON BETWEEN SPARK'S HYBRID, OUR RANGED-HYBRID AND HEINTZE'S SHARED POINTS-TO SETS. THE COLUMN RANGE+HYBRID SHOWS OUR IMPLEMENTATION.

program	hybrid		range+hybrid		heintze	
	time	space	time	space	time	space
jedit	12.5	79.5	8.5	31.9	18.6	4.9
soot	10.0	98.4	8.5	34.4	28.1	5.4
jflex	5.0	65.2	5.3	25.9	13.2	4.1
sablecc	1.6	17.2	2.0	8.8	3.1	1.5

Programs jEdit, Soot, JFlex and SableCC consist of 13583, 13741, 11893 and 5299 methods respectively. This shows that our most efficient points-to set (Hybrid Ranged Points To Set) makes the analysis of the two larger programs faster (JEdit and JFlex) and consumes significantly less memory in all four cases when compared to Spark's hybrid set (the fastest and default points-to set of Spark in current version of Soot).

We also compared our implementation to sparse bitmaps [16], [17] in term of space that would be saved if our techniques were combined with the sparse bitmaps (i.e. every element in a sparse bitmap would be a Ranged Bit Vector of size 8×4 bytes and no element would be allocated unless it has at least one bit set). Table II shows the results of this experiment. Considering the results shown in the table and the cost related to the linked list manipulation of elements in sparse bit-maps, we believe that sparse bitmaps are not a really good candidate to be combined with our techniques. One reason could be that the Ranged Bit Vectors are not sparse because of the way we numbered allocation nodes.

Since in our ordering, all allocation nodes of compatible types are consecutive and a points-to set of type T is filled

only with those of compatible type with T, but in the plain sparse method, allocation nodes of type T would be scattered and are not grouped in some interval. Hence the sparse method combined with our way of numbering has higher chance to allocate fewer elements.

TABLE II: This Table Shows the Space That Would Be Saved if Our Techniques Were Combined with Sparse Bitmaps. The Column Range Shows Ranged Points-To Set as Described in Section vi and the Second Column Shows Our Most Efficient Set Implementation as Described in Section vi (Total Set Size /Space That Would Be Saved if Was Used in Combination with Sparse Bitmaps).

program	range	range+hybrid
jedit	49.3/6.1	31.9/6.4
soot	44.7/5.9	34.4/5.9
jflex	37.3/4.6	25.9/4.8
sablecc	9.6/1.1	8.8/1.1

To compare the precision of our intrinsic type filtering with type filtering based on type masking, we considered all variable nodes which are dereferenced. Table III shows the percentage of points-to sets with 0, 1, 2, 3 - 10, 11 - 100, 101 - 1000 and more than 1000 elements in their points-to sets.

According to this table, reduction of precision which is caused by the way we do type filtering (alignment of the common subrange as described in Section V) is trivial.

TABLE III: Precision of Intrinsic Type Filtering and Type Filtering Based on Type Masking - Precision: Intrinsic (Ranged Points-To Set) / Type Masking (Pure Bit-Vector) (% of Total).

	jedit	jflex	soot	sablecc
0	5.48/5.48	4.49/4.49	1.41/1.41	6.37/6.37
1	28.99/29.01	31.70/31.72	37.85/37.87	34.99/34.99
2	7.11/7.12	6.89/6.91	17.90/17.90	10.72/10.80
3-10	47.99/48.02	47.00/47.05	36.17/36.19	39.95/39.95
11-100	7.81/7.87	9.47/9.39	5.28/5.28	7.71/7.63
101-1000	2.53/2.42	0.35/0.35	1.29/1.26	0.22/0.22
1000+	0.0/0.0	0.0/0.0	0.0/0.0	0.0/0.0

VIII. Conclusion

In this paper, we presented and evaluated an improvement to points-to analyses which orders allocation nodes in our desired way and uses our implementation of points-to set. This work also combined our methods with Spark's hybrid points-to set idea and evaluated the performance gain.

Type filtering is done in our method implicitly (it is done by means of the Ranged Bit Vectors). In another word, we do not need the additional logical and operation to enforce type filtering (type masks). This feature along with having smaller bit-vectors made our method more efficient than the previous works. Note that to do the or operation in the units of chunk some justification is performed as you saw in Fig. 3 which is an additional overhead compared to previous works. Despite this overhead, we see an improvement in time for sufficiently

large programs. See Section VII.

We observed that our work consumes 2.5×less memory than hybrid points-to set (default set in Spark). It also improves points-to propagation time for sufficiently large programs compared to the same set. We also compared our set implementation to other state of the art implementations including Heintze's shared bit-vectors [5] and sparse bitmaps [16], [17]. We observed that Heintze's shared bit-vector sets use less memory than both our implementation and Spark's hybrid sets but it is also slower than both of them. We did not find sparse bit-maps a good candidate to be combined with our method and we believe that it is practically less efficient than our method (see Section VII).

Finally, you can find further details like the way allocation nodes of array types are handled in our technical report [18].

References

[1] L. O. Andersen, "Program analysis and specialization for the C programming language," PhD thesis, University of Copenhagen, 1994.
[2] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDDs," in *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ACM Press, pp. 103-114, 2003.
[3] J. Whaley and M. Lam, "Clonning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proc. of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM Press, pp. 131-144, 2004.
[4] O. Lhotak and L. Hendren, "Scaling java points-to analysis using Spark," in *Proc. of the 12th International Conference on Compiler Construction*. Springer-Verlag, pp. 153-169, 2003
[5] N. Heintze, "Analysis of large code bases, the compile-link-analyze model," Draft version (unpublished), November 1999.
[6] O. Lhotak, S. Curial, and J. N. Amaral, "Using ZBDDs in points-to analysis," *Languages and Compilers for Parallel Computing*, Springer-Verlag, 2007, pp. 338-352.
[7] J. Zhu, "Symbolic pointer analysis," in *Proc. of International Conference on Computer Aided Design (ICCAD)*, pp. 150-157, November 2002.
[8] C. Lin and B. Hardekopf, "Semi-sparse flow-sensitive pointer analysis," in *Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 226-238, 2009.
[9] M. Hirzel, D. V. Dincklage, and A. Diwan, "Fast online pointer analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 2, April 2007.
[10] O. Lhotak, S. Curial, and J. N. Amaral, "Using XBDDs and ZBDDs in points-to analysis," *Software, Practice and Experience*, vol. 39, no. 2, pp. 163-188, February 2009.
[11] J. Dean, D. Grove, and C. Chambers, "Optimization of object oriented programs using static class hierarchy analysis," in *Proc. of the 9th European Conference on Object-Oriented Programming (ECOOP 95)*, August 1995.
[12] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proc. of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 324 - 341, 1996.
[13] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. V. Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for Java," in *Proc. of the 15th ACM SIGPLAN Conference on*

Object-Oriented Programming, Systems, Languages, and Applications, pp. 264-280, 2000.

[14] J. Gosling, B. Joy, and G. Bracha, *The Java Language Specification*, third edition, 2005.

[15] B. Ryder, "Dimensions of precision in reference analysis of object oriented languages," in *Proc. of International Conference on Compiler Construction (CC)*, 2003.

[16] GCC, the GNU Compiler Collection. [Online]. Available: http://gcc.gnu.org

[17] LLVM, the LLVM Compiler Infrastructure Project. [Online]. Available: http://llvm.org

[18] H. A. Toussi and A. Khademzadeh, "IPLP resesrch group technical report: Improving bit-vector representation of points-to sets using class hierarchy analysis," Islamic Azad University of Mashhad, July 2010.

[19] V. Sundaresan, P. Lam, E. Gagnon, R. V. Rai, L. Hendren, and P. Co, "Soot - a Java optimization framework," in *Proc. of CASCON 1999,* pp. 125-135, 1999.

Hamid Alavi Toussi was born in Iran (October 1985). He received his BSc. degree in Computer Engineering from Islamic Azad University, Mashhad, Iran in January 2009 and received his MSc. degree in Computer Science from Univesity of Sistan and Baluchestan, Zahedan, Iran in August 2011. His research interests includes Compilers and Programming Languages, Operating Systems and Computer Architecture.

Ahmed Khademzadeh received his BSc. degree in Computer Engineering from Ferdowsi University, Mashhad, Iran in March 2001 and received his MSc. degree in Software Engineering from Iran University of Science and Technology, Tehran, Iran in March 2004. He is currently a PhD. student at Florida Institute of Technology, Florida, USA. His current research involves large-scale machine learning and data mining.