McGill University
School of Computer Science
COMP 621

Faster Intraproceducal Analyses

Report No. 2013-04

Bhaskar Pilania
Mentor: Prof. Laurie Hendren

April 29, 2013

Contents

List of Figures

List of Tables

Abstract

Program analysis plays an important role in compiler optimizations, software mantainance and information flow. Slower program analysis may lead to high execution time of programs and hurts the efficiency. In this paper we have tried to address the problem of slow intraprocedural analysis of McLab. We have first redefined the problem in following subproblems. Currently McSAF performs all program analysis computations in hash based data structure. Accessing and performing computations on these datastructures is costly as hash based datastructures are prone to collisions. Besides this, McSAF recomputes flow variables i.e. inflowSets and outFlowSets regardless of change in them since the last statement. This behaviour takes up extra memory and wastes computation time even when it is not required. Another problem is, McSAF also does not store kill and gen sets of program statements as the program is analyzed. This behaviour pushes the user to recompute them each time when a relevant analysis is to be done on these gen and kill sets.

To address the problems, in our first step, we have introduced a bit vector based data structure for computation of inFlowSets and outFlowSets. Since bit vectors are effective at exploiting bit-level parallelism in hardware they are found to perform the computations about 35% faster than hash based data structures. Another introduced optimization is storing the gen and kill sets for each node in the AST so that user can access them at any further time if it requires to do an analysis using them. This optimization makes any further analyses on gen and kill sets faster as they are computed only once. Another optimization is achieved in terms of memory where the new implementation prevents the recomputation of flow variables i.e. inFlowSets and outFlowSets when gen and kill sets are same i.e. flow variables have not changed. This optimization saved considerable amount of memory and a small amount of computaton speed.

Towards the end, the paper discusses the results in more detail and suggests the furute work.

1 Introduction and Motivation

McLAB[2] is a research project that aims to provide compilers, language conversions and virtual machines for dynamic scientific languages. McLAB has a static analysis framework, McSAF[2] which is intended to simplify and fasten the development of new compiler tools for MATLAB. McSAF analysis framework is equipped with a very basic functionality for program analyses and the design of McSAF is unlikely to perform optimally in terms of analysis speed. Slow program analysis lead to longer execution time of the desired task and hurts the efficiency. In wake of these problems, it becomes important to make these analyses faster. This project aims to design and develop faster versions of analyses framework for McLAB to enhance execution performance of McSAF and thereby performance of McLAB.

2 Background and Related Work

Due to high program execution time of MATLAB, computer scientists have been trying and implementing different methods to reduce the MATLAB program execution time. One such relevant work is MaJIC(Matlab Just In Time Compiler) which compiles and optimizes the code behind the scene in real time by employing a combination of just in time and speculative ahead of time compilation[1]. MaJICs dynamic (JIT) compiler uses extremely fast type inference engine and a relatively naive, but fast, code generation engine. Before compilation, MaJIC dynamic JIT compiler tries to gather maximum possible runtime information which allows compiler to save on time consuming optimization steps. MaJIC also performs speculative ahead-of-time compilation. This is accomplished as compiler looks at source code and guesses the run-time context most likely to occur in practice. Also, MaJIC maintains repository of precompiled codes which serves for ahead of time compilation.

Another product having the same aim is McLAB[2] which is developed by Sable research group of McGill University. As already discussed, McSAF analyses speed can be improved for certain analyses so that it can perform faster analyses.

3 Specific Problem Statement

Existing McSAF implementation[2] provides with following two kinds of basic analyses:
1. Depth First Analysis
2. Structural analysis
2.a. Forward Analysis
2.b. Backward Analysis
Out of these analyses Structural analysis is used to compute complex information to approximate runtime behaviour. These analyses provide basic operations like merge, copy on flow-data.
The problem which McSAF faces is that it performs all the computations in hash based data structures namely HashMapFlowMap and HashSetFlowSet. Though hash based data structures performs get operations in constant time, this constant can get very high as the number of statements in matlab program grow to a very high value. This is primarily because as as number of nodes grow, the probability of having more than one statement at the same hash bucket increases. Which implies hash based data structure is only suitable when there are very less number of insertions in hash and thus less collisions.
Another problem of McSAF is that it recomputes inFlowSets and outFlowSets for each node of the program even though they are sure to be same as that of previous statement. This behaviour is not good as these computations are unnecessary and cost the analyses in terms of computation time and cloning space.
Another identified concern is that these computed inFlowSets and outFlowSets at each node of the program are not stored anywhere. This can be a overhead if user want to preform an analysis again then user will have to recompute these inFlowSets and outFlowSets. Instead there should be some mechenism in place to save these computated values at the first time so that they can be used anytime futher if required.
Another possible area of improvement was to save the precomputed inFlowSets and outFlowSets in a file so that these can be accessed any time at a later date for running the analyses.
The project expected to build on above mentioned structural analysis framework keeping in mind the possible areas of improvement so that McSAF can perform faster intraprocedural analyses. More specifically, the problem demanded to design and implement operations that could reduce run time by minimizing standard computations required during runtime and doing and storing these computations during compile time.

4 Solution Strategy and Implementation

Following solution strategies have been implemented to get performance gain on McLab analyses.

1. Since some differing analyses tend to require similar behaviour and may require same set of gen and kill sets, one of the major speedup can possibly be gained by precomputing gen and kill set for each statement and storing them in the tree for later use in relevant analyses.This has been achieved by storing gen and kill sets in ASTNode against their respective nodes. For example let there be a node,
a = b + c;
kill Set = {a}
gen Set = {b, c}

These gen and kill sets are now being saved in AST in below metioned fields.

```
protected BitSet genBitSet;
protected BitSet killBitSet;
```

To access these fields following getter and setter methods are provided.
```
public BitSet getGenBitSet() {
return genBitSet;
}
public void setGenBitSet(BitSet genBitSet) {
```

```
this.genBitSet = genBitSet;
}
public BitSet getKillBitSet() {
return killBitSet;
}
public void setKillBitSet(BitSet killBitSet) {
this.killBitSet = killBitSet;
}
```
As explained above, once these fields are set, they can be reused in any relevant analyses the user wants to perform using gen and kill sets and it greatly reduces the analyses time.

2. As we discussed earlier that computations on hash based datastructures i.e. HashMapFlowMap and HashSetFlowSet can be very expensive. This is because for each time to perform computation, the hashmap is accessed and all merge, remove operations are performed on these maps. To improve on this front we have implemented a new data structure based on Bit Vectors. The reason for using bit vectors is that a bit array is effective at exploiting bit-level parallelism in hardware to perform operations quickly. Operations like merge, intersection, not etc can be performed in atomic time i.e. operations are performed on all bits in one go. This is explained below.
Suppose for doing Live variable analysis, we have to perform operation on following node to compute inFlowSet.
inFlowSet = ? // To be computed
kill set = {a} a = b + c gen set = {b,c}
outFlowSet = {}

if a,b,c represents bit vector positions 0,1,2 then out killSet and genSet becomes,
killSet = 100
genSet = 011
For bit vector operations, our equation becomes,
inFlowSets = (outFlowSet AND NOT(killSet)) OR genSet

It can be visualized from above equation that inFlowSets are being computed with atomic operations AND and OR on bit vectors. This parallel computation makes the analyses faster.
To make the analyses even more faster, we are storing NOT(killSet) in the AST. This is being done with following implementation.
```
killBitSet.flip(0,outLength); //computing NOT(KILL)
node.setKillBitSet(killBitSet);//saving NOT(KILL) in the AST
```
Storing these values further reduces the computation equation to,
inFlowSets = (outFlowSet AND killBitSet) OR genSet
This optimization further makes the computations more efficient.

3. As previously discussed, McSAF currently recomputes inFlowSets and outFlowSets for each program statement regardless of its change from the previous values. To improve on this behaviour, we now analyze the gen and kill sets of the statement in consideration and if gen and kill set are equal then it implies that inFlowSets and outFlowSets will be same as for previous computation and thus we should not recompute them. So, we save the reference of previous computed values with the current statement thereby saving on storage space and computation time. This is being achieved by following implementation,
```
BitSet temp = new BitSet();
temp.or(killSet);
temp.xor(genSet);//if gen and kill set are same then their xor will yeild all bts set to 0
if(temp.nextSetBit(0) == -1) { // We search of index of next set bit
flag = 1;// If above condition satisfies then it implies gen and kill are same and so flag is set to 1
```

}

If the flag's value is set as 0 then we recompute the inFlowSets and outFlowSets and if flag value is 1 then we just save the reference to previous values. Following code is the implementation snippet of the same.

```
if(flag == 0){
compCount++; //Number of times computation is performed
currentInSet = newInitialFlow();
currentInSet = currentOutSet.copy();
currentInSet.and(killBitSet);
currentInSet.or(genBitSet);
inFlowSets.put(node, currentInSet.copy());
}
else{
compSaveCount++; //Number of times computation saved
outFlowSets.put(node, currentOutSet);
inFlowSets.put(node, currentInSet);
}
```

4. In another attempt for getting faster analyses, we serialized the inFlowSets and outFlowSets objects and stored them in a file for use at a later date. The impementation of serializing and deserializing the inFlowSets and outFlowSets was done by implementing Java's serializable interface along with file operations. A class named SerializeFlowVariables with was created for assisting the operations of saving to the file.

5 Experimental Framework and Results

Software Used: Eclipse Juno Service Release 1
Operating System: Fedora 17 (64 bits)
Hardware setup: 2.5Ghz dual core Intel processor, 4 GB RAM

Experimantal Setup: The experimental setup aimed to test the following:
1. Correctness of bit vector based implementation
2. Comparision of Bit vector based implementation with Hash based existing implementation in terms of analyses speed and memory used.

To get a better precision of readings, each analyses was run 10 times and the average was taken of total time elapsed.

5.1 Benchmark programs

For testing analyses speed, four benchmarks were picked from the set of benchmarks which were provided to us for Assignment - 2. Besides that another program of over 100,000 lines was taken from internet.
Parameters for choosing the benchmarks:
1. Large number of array accesses.
2. Large number of function calls.
3. Size of program.
4. Number of loops and level of nesting in the loops.

Number of array access/function calls affects the speed due to Kind analysis and thus this was taken an important factor for choosing programs. The purpose of taking a very large sized program was to determine how bad existing hash based datastructure and new implementation based on bit vectors do on a very large sized programs. Since increase in number of loops and if/else statements increase in number of fixed point computations, it also became an important factor for choosing the benchmark programs. Following the set of benchmark programs used and their associated characterstics.

Table 1: Benchmarks and their characteristics

Benchmark Program	Characteristics
Alphtrmed.m	Small program; Have a 6 level nested loop structure.
ImageSharing.m	Mediun sized program; Have a lot of loops so more fixed point computations.
Direct.m	Large program; Have large number of loops and function calls.
erdosRenyi.m	Small program; Have a large number of arrays access, functions calls and structures.
big.m	Huge sized program of over 100,000 lines, most lines repeated(hence will compute same hash).

5.2 Results

Following are the results and discussion on the results obtained in this experiment.
1. Before proceeding with testing the implementation for analysis time, it is important to check the correctness of new implementation. The new analysis should correctly perform fixed point computations, kind analysis for checking a symbol is a function or an array etc. Following is input Matlab file and corresponding live variable analysis using bit vector implementation.

Sample program for checking correctness

```
m = read();
n = read();
i = n;
if (n ¿ m)
while (i == 0)
n = n + i;
end
else
while (i ¿= 0)
m = m * 1;

   end
end
```

Following is its pretty printed analysis:

```
in: { }
m = read();
out: { m, }

   in: { m, }
n = read();
out: { m,n, }

   in: { m,n, }
i = n;
out: { i,m,n, }
```

```
    in: { i,m,n, }
if (n ¿ m)
while (i == 0)
n = (n + i);
end
else
while (i ¿= 0)
m = (m * 1);
end
end
out: { }

    in: { i,n, }
while (i == 0)
n = (n + i);
end
out: { }

    in: { i,n, }
n = (n + i);
out: { i,n, }

    in: { i,m, }
while (i ¿= 0)
m = (m * 1);
end
out: { }

    in: { i,m, }
m = (m * 1);
out: { i,m, }
```

After careful review we found thar the implementation achieves correctness.

2.This experiment was performed to measure the analyses speed of the implemantation. Following table is the summary of all the computation times.

Table 2: Computation times for HashSetFlowSet vs BitVector Datastructure

Benchmark Program	HashSetFlowSet	Bit Vector
Alphtrmed.m	38ms	18ms
ImageSharing.m	42ms	19ms
Direct.m	51ms	30ms
erdosRenyi.m	30ms	7ms
big.m	2980ms	2262ms

Above results suggests us following,
a. For smaller programs, we are getting a average speed up of over 35%. This implies that doing computations on hash based data structure is slow even when there are very few or no collisions. This implies get and set operations on hash based data structure are costly.

On the other hand Bit Vector based implementation is extremely fast as underlying AND, OR, NOT operations take very less time. The fast behaviour is also because in bit vector approach, we can perform computation on all the elements in gen and kill set in one statement while in hash based data implementation we used to iterate over gen and kill sets performing operations like union, intersection etc for individual elements.

b. Results suggests that for very large programs, performance of both the approaches deteriorates but still Bit Vector based implementation turns out to be 24% faster. The reason for such high performance hit of hash based implementation is that as program size increase, number of statements with same hash value increase and thus collisions at hashmap buckets increase. And with collisions, the hash access time complexity surges to log(n) and thus is no more constant. Deterioration of bit vector based approach can be explained by the fact that with large number of live variables, the size of bt vector increase and thus computations on it takes longer.

3. Next experiment was to determine the impact of the implementation on memory. Here an important positive result to report is number of recomputations saved over same inFlowSets and outFlowSets. Though this parameter highly depend on number of such statements in the Matlab program for which inFlowSets and outFlowSets are same.
It was found that number of saved recomputations in a program ranged from 5% to 20%. As already discussed this variation is caused by number of such statements for which inFlowSets and outFlowSets is same. There was very less % of speedup achieved on this optimization but certainly it saves considerable amount of memory.

4. One negative result to report is on serialization of inFlowSets and outFlowSets. It was oberved that for a 20k lines of Matlab file,average analysis time without serialization was 653ms and average analysis time with serialization turned out to be 702ms. This suggests serialization did not help in faster analysis because file operations turned out to be very costly and thus serialization of inFlowSets and outFlowSets is a bad idea.

6 Conclusions and Future Work

The paper aimed to propose a faster and more efficient intra procedural analyses techniques for McLab. For achieving efficiency in terms of speed and memory, four apporaches were implemented.
First was to perform all the computations of inFlowSets and outFlowSets on a bit vector based data structure as it exploites bit-level parallelism in hardware to perform operations quickly. Second was to save the computed gen and kill sets for each statement in the AST so that any further analyses can be performed over these without need to recompute these gen and kill sets. These two approaches combined gave a speed up of over 35% in terms of analyses speed. By all means, this is very efficient over hash based data structures and I believe Sable research group should introduce these approach in McSAF.
Third optimization was achieved in terms of memory where new implementation prohibits the recomputation of inFlowSets and outFlow sets in case whengen and kill sets are same. This optimization save a considerable amount of memory usage and a small amount of computaton speed. This optimization should can also be introduced in the McSAF.
Another important point to note is that file operations are very costly operations and these should be avoided. An attempt to find a seedup by serializing the inFlowSets and outFlow sets was in vain as file operations took huge amount of time.

This paper is a naive implemetation and one of few first steps towards optimization of McSAF. There is still a lot of scope available to improve the performance of McSAF. A good point to start is to start analysing performance bottlenecks in the product and then improvements should be implemented on those lines.

A My Appendix

I have put all the code on github. Link for the repository is https://github.com/bpilania/Comp621Project.git All the code is properly documented, with comments. I have also provided a readme file in the repository which explains all the files in which I have changed/written the code. For any further questions, I can be contacted on bhaskar.pilania@mail.mcgill.ca.

References

[1] George Almasi and David Padua. Majic: Compiling matlab for speed and responsiveness. Technical report, University of Illinois at Urbana-Champaign.

[2] Jesse Doherty. Mcsaf: An extensible static analysis framework for the matlab language. McGill University, Montreal, August 2011.