Demand-driven less-than analysis

Junio Cezar R. da Silva
Universidade Federal de Minas Gerais - UFMG
Avenida Antonio Carlos, 6627
Belo Horizonte, Minas Gerais 31.270-010
juniocezar@dcc.ufmg.br

Fernando Magno Q. Pereira
Universidade Federal de Minas Gerais - UFMG
Avenida Antonio Carlos, 6627
Belo Horizonte, Minas Gerais 31.270-010
fernando@dcc.ufmg.br

ABSTRACT

A less-than analysis is a technique used by compilers to build a partial ordering between the integer variables in a program. Recently, researchers have shown how to use less-than information to improve the precision of alias analyses. The literature describes two techniques to build less-than relations. Both are asymptotically equivalent to computing a transitive closure in a graph. In this paper, we depart from this approach, and introduce an algorithm that builds less-than relations on demand. We claim that such algorithm is more adequate than the current state-of-the-art approaches, as it performs only the necessary work to satisfy the needs of its clients, i.e., alias analyses and optimizations that require less-than information. To validate our idea, we have implemented it onto the LLVM compilation infrastructure. Depending on the client analysis, our implementation may lead to runtime savings of up to 68% on large benchmarks, when compared to the more traditional approach based on the construction of the transitive closure.

KEYWORDS

Less-than Analysis, Compiler, Transitive Closure, Demand-Driven

ACM Reference format:
Junio Cezar R. da Silva and Fernando Magno Q. Pereira. 2017. Demand-driven less-than analysis. In *Proceedings of SBLP 2017, Fortaleza, CE, Brazil, September 21–22, 2017,* 8 pages.
DOI: 10.1145/3125374.3125379

1 INTRODUCTION

Less-than analyses [7, 9] build a partial ordering between the integer variables used in a program. The ability to conclude that a variable X is always less than a variable Y is useful in several different scenarios, including security analyses and performance optimizations. For instance, less-than analyses may be used to validate array bounds checks [2, 7, 8], detect data races in parallel programs [17], improve runtime performance of applications by removing guards [11] and disambiguate pointers in imperative programming languages [9].

Less-than analyses serve other analyses and optimizations that are part of the compiler's tool box. We call any program pass that requires less-than information a *client* of that analysis. Different clients are not expected to use the less-than analysis in the same way. For instance, loop optimizers will require ordering information between induction variables and loop-invariant code. Restrictifiers [1, 18], in turn, shall build relations between pointers. In other words, each client might query relations between different pairs of variables. Although obvious, we claim that previous work does not capitalize on this simple observation. The current implementations of this service build a complete table of relations between program variables. This approach, even though correct, is not efficient, as we show in Section 2. Inefficiency, in this case, stems from the fact that most of the data in the inequality table will not be queried by an average client. On the other hand, building this table on demand is not a trivial endeavour, as it involves processing partially a constraint system typically solved via transitive closure [12, Ch.03].

A constraint system is a common way of representing the relationship among variables in the less-than domain [7–9]. The particular family of constraints used, in this case, can be solved via the well-known cubic algorithm that builds transitive closures of graphs. There are efficient implementations to build transitive closures, yet, given the size of their output, they are still superlinear [10, 13, 20]. We believe that the transitive closure is an unnecessary price that clients must pay to use a less-than analysis. Thus, our goal is to detach the implementation of a less-than analysis from the need to build a transitive closure between the integer variables that exist in a program. Ultimately, this decoupling tends to increase the performance of our algorithm with regards to the traditional implementation.

In this paper, we achieve our objective by making the less-than analysis *partially demand-driven*. This strategy combines the generation of constraints by an inspection of the whole program in a pre-processed stage with less-than information generated on-demand. With the combination of these two techniques, we are able to construct relational less-than sets for any variable that is either used in arithmetic operations or is used to store the result of such operations. As shown by Maroua *et al.* [9], a potential use of these less-than sets is for the disambiguation of pointers. We have engineered our analysis towards this direction and, as shown in section 4, we are able to observe runtime improvements in a wide range of applications.

We have implemented and tested our analysis in LLVM [6], an industrial quality compilation framework. We have used this framework to evaluate our implementation on more than 20 benchmarks from the SPEC CPU 2006 test suite [5]. As shown in section 4, our implementation works by answering *true* or *false* to queries in the form of: *Is variable x less than variable y*? The *Loop Invariant Code Motion (licm)* pass – present in the LLVM compiler – is an example of a client application that may benefit from this kind of query. While working with *licm*, we were able to reduce by 30% the runtime of Maroua *et al.*'s implementation of the less-than analysis. In

SBLP 2017, Fortaleza, CE, Brazil

DOI: 10.1145/3125374.3125379

some benchmarks such as *dealII*, *gcc* and *milc* this speedup reaches 61%, 117% and 171% respectively. Moreover, we also demonstrate empirically that the queries performed by *licm* involve a number of less-than sets that is substantially lower than the total number of sets constructed by traditional approaches. This observation indicates that our demand-driven algorithm is more adequate to the *modus operandi* of typical compiler optimizations.

2 OVERVIEW

In this section we show how less-than information is used in practice, and we explain why the current approaches to build less-than relations still offer room for improvement.

2.1 Less-Than analysis: usage example

Do pointers A and B refer to the same memory location? This is a common question that arises while software engineers are reasoning about a source code or while compilers are analyzing a program. Alias analysis is a class of code analyses designed to answer this kind of question. In the positive case, where the analysis discovers that these two pointers hold references to the same memory location, they are said to alias. In this section, we shall use pointer disambiguation to motivate our work.

```
1   struct S {
2     int acc;
3     int size;
4     int *vet;
5   };
6
7   int fun1(struct S *stt, struct S *stt2) {
8     int i;
9
10    for(i = 0; i < stt->size; i++) {
11      stt->acc += (stt->vet)[i];
12    }
13
14    stt2->acc += (stt2->vet)[stt2->size - 1];
15
16    return stt2->acc + stt->acc;
17  }
```

Figure 1: An alias analysis, supported by less-than information, can disambiguate pointers in the loop inside `fun1`.

We will use the function illustrated in Figure 1 to demonstrate the need for alias analysis and how it can be effective to generate more efficient programs. This example contains a simple routine that updates the values of accumulators within two structures and, then, returns the sum of these two variables. A problem with function `fun1` is that it requires four memory accesses, at each iteration of the loop in line 10, to retrieve the values stored in the struct members `acc, size` and `vet`. Memory accesses are expensive operations, therefore a way the compiler could optimize that function is by decreasing the number of these operations executed while the program is running.

Figure 2 shows an optimized version of the loop seen in Figure 1. This code only performs four memory accesses, for all possible iterations of the loop, to load or store the same values from the members of struct `stt`. Figure 2 is the result of a well-known code optimization: *loop invariant code motion*. However, compilers may only apply it when they know that the memory locations accessed to fetch

```
1   int tmp1 = stt->size;
2   int tmp2 = stt->acc;
3   int *tmp3 = stt->vet;
4   for(i = 0; i < tmp1; i++) {
5     tmp2 += tmp3[i];
6   }
7   stt->acc = tmp2;
```

Figure 2: Loop present in line 10 of Figure 1, after loop invariant code motion (`licm`) was applied by the compiler.

those values are pairwise different. In this example, if the memory locations dereferenced to retrieve the values of `stt->acc` and `stt->size` were the same, the compiler would not be able to generate the code of Figure 2, because at each iteration of the loop the value of `stt->size` might change, which would lead the loops from the two figures to generate different results.

In programming languages that support pointer arithmetics, such as C and C++, less-than analyses have been shown to be effective to disambiguate pointers [9]. Basically, in this case we say that two pointers, p_1 and p_2, do not alias if the address referenced by p_1 is strictly less than the address referenced by p_2. In Figure 1, a less-than relations based alias analysis would be able to distinguish the memory locations accessed in the loop present in line 10, enabling the generation of the optimized code illustrated in Figure 2. A less-than analysis can differentiate the two fields of the struct because the memory addresses of these members are obtained by the sum of the struct's base pointer plus an offset that is known at compilation time. As a result, once the compiler knows which values correspond to these offsets, it may define if the pointers are accessing or not the same memory location. Figure 3 shows how the compiler would extract the offsets. The figure shows the loop of Figure 1 in LLVM's assembly. The dashed arrows indicate where the pointer's address is

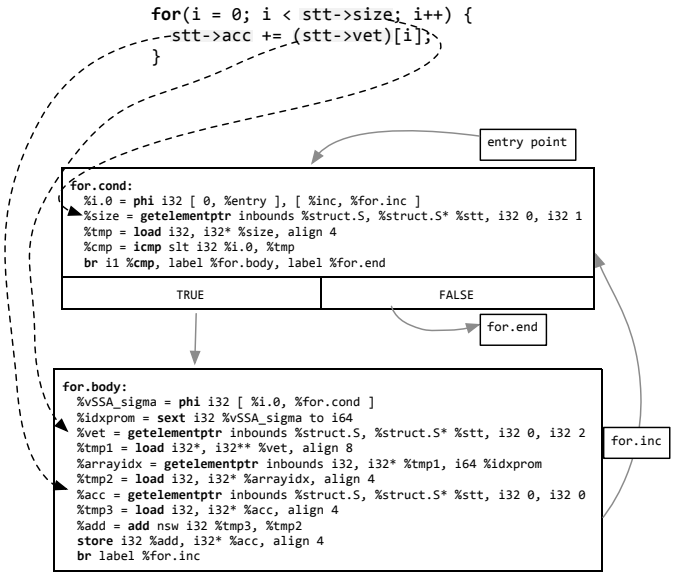

Figure 3: The intermediate representation of the loop in Figure 1. Less-than relations disambiguate struct fields.

being calculated. The special instruction **getelementptr** is used to perform the sum of the struct's base pointer `%struct.S %stt` and the offset, which is 0 for the member `acc`, 1 for `size` and 2 for `vet`. Using this information, the less-than analysis is able to create inequalities out of these operations and distinguish these pointers.

2.2 The Shortcomings with Transitive Closures

The work of Maroua *et. al.* [9] presents the most up-to-date employment of less-than relations to answer alias queries. They provided a snapshot of the potential of such analysis – showing the benefits in precision when compared to previous analyses. But one of the problems faced by their approach is that it consumes an elevated amount of time to build less-than relations. Most of this drawback is due to the fact that Maroua *et al.* create a transitive closure of a graph that has one vertex per integer variable in a program. For large programs, this graph might have thousands of nodes, and computing a transitive closure is well-known to have an $O(V^3)$ time complexity. A transitive closure is a structure that explicitly specifies all possible direct paths within a graph. In other words, a transitive closure of a directed graph G is defined as a second graph $G' = (V, E')$, where V is the set of vertices of G and E' is the set of edges (n_i, n_j) that shows that there is a path between the nodes n_i and n_j in G. Figure 4 shows the closure that represents the function `fun1`, seen in Figure 1. Figure 4 (a) shows the component built out of the less-than relations extracted from the loop in line 10 and (b) shows the component of the closure produced by the relations extracted from operations in line 14.

The transitive closure might lead to unnecessary computations, as it does not consider specific needs of the client that requests less-than information. For instance, two of the connected components of the closure seen in Figure 4 (involving variables `i_fbody` and `stt2`) do not contribute towards the generation of the optimized loop of Figure 2. Therefore, these two components would not be necessary to carry out that optimization. This problem is even more evident if we consider a secondary function, that does not have any relation with `fun1`. Figure 5 illustrates this case. The function presented in this Figure (`fun2`) will provide more variables, that must still be accounted for in the transitive closure, even though `fun2` does not bear impact in the optimization performed in Figure 2.

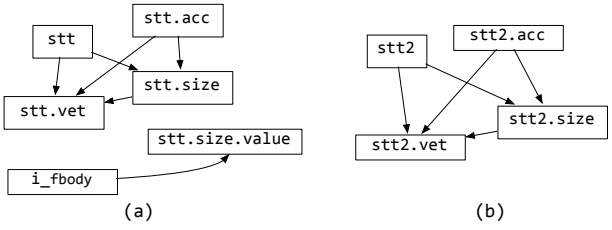

Figure 4: Transitive closure representing the less-than relations extracted from the function `fun1` in Figure 1. Arrows indicate ordering. A variable that is the source of an arrow has its value less than the target one. The nodes *i_fbody* and *stt.size.value* represent, respectively, the values of the variable *i* inside the *for* loop and the struct member *size*. All the other nodes represent pointers to struct members.

```
1   int fun2(int num1, int num2) {
2     int A = num1 + 1;
3     int B = A + 1;
4     if(B > num2) {
5       A += num2;
6       B -= num1;
7     } else {
8       A = num1;
9       B = num2;
10    }
11    return A + B;
12  }
```

(a)

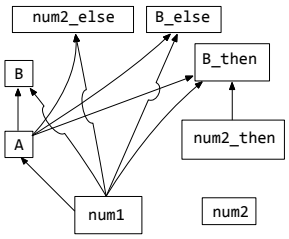

Figure 5: Sample function (a) that would have its transitive closure (b) calculated by the approach used by Maroua *et. al.* [9], even though the creation of this closure does not affect the result obtained in Figure 2. As nodes *A*, *A_then* and *A_else* connect to the same arrows, we decided to merge them into the same node *A* in the graph of letter (b).

In an effort to demonstrate that this problem indeed happens with real applications, we have analyzed the behavior of Maroua *et al.*'s analysis on the benchmark *dealII* from SPEC CPU 2006 [5]. We consider, in this experiment, queries performed by LLVM's *licm* and *instcombine* implementations. The results of this study, illustrated in Figure 6, show that the transitive closure approach produces much more information than client optimizations require. While answering alias queries from the *licm* pass, Maroua *et al.* produced 3.67 times more less-than sets than the client analysis demanded. For the *instcombine* pass – an optimization that tries to combine redundant instructions – this difference between consulted and constructed less-than sets increased to 15.36 times. These sets are the standard way to organize less-than information. They associate a variable v with other variables x, such that $x < v$. For example, the less-than set of variable *B_then*, in Figure 5 (b), contains all variables that have their values less than *B_then*: *num2_then, num1,* and *A*. Some of these variables, such as *B_then*, are not explicitly present within the body of function `fun2`. They appear in the intermediate representation that LLVM produces for that function.

Our technique is able to answer the same alias queries, while performing fewer calculations. Instead of calculating a full transitive

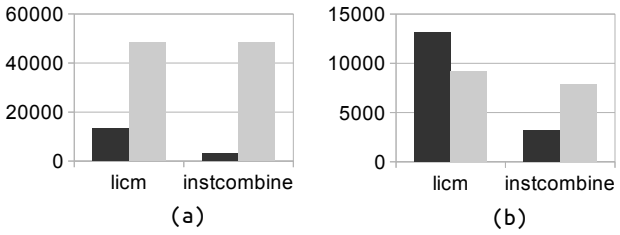

Figure 6: Result of generating less-than sets using a fully pre-processed approach (a) and a demand-driven approach (b) for two client analyses. Dark bars represent the number of less-than sets consulted by the client analyses. Light-gray bars represent the less-than sets constructed in order to answer the alias queries.

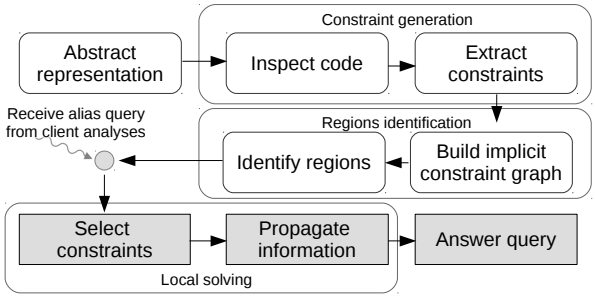

Figure 7: Steps adopted by our less-than relations based demand-driven alias analysis. White rounded boxes represent our pre-processed stage, while the grey squared boxes show the stage we move into on-demand.

closure beforehand, we have embraced a demand-driven approach which allows us to generate specialized less-than information for each source of alias queries. As already shown in Figure 6, we can reduce the number of less-than sets calculated in different optimizations. This happens because the alias queries generated by the clients *licm* and *instcombine* involve only a subset of the integer variables present in the target program. This observation is not exclusive to these two optimizations; rather, it characterizes all the optimizations available in LLVM that we have studied. Furthermore, intuition lets us extrapolate this observation to other compilers. This behavior favours an on-demand approach to build less-than relations. Note that, in order to answer the queries performed by *licm*, our technique needed to calculate a lower number of less-than sets than the number required by that optimization. This behavior is due to the use of *region identifiers*, as will be introduced in section 3.2. This identification system allows us to answer queries without having to pay the cost of calculating any less-than set. From now on, we shall use the notation LT to refer to a generic less-than set and LT(x) to refer to the less-than set owned by variable x.

3 DEMAND-DRIVEN STRATEGY

In this section we shall describe the main steps of our partial less-than demand-driven analysis: (1) constraint generation, (2) regions identification, and (3) local solving. The workflow for these stages is illustrated in Figure 7. At the end of this process, we solve a less-than analysis, which we define as follows:

Definition 1 (Less-Than Analysis)**.** A less-than analysis is a static program analysis that computes, for each variable x, its less-than set $LT(x) = \{x_1, \ldots, x_n\}$. If $x_i \in LT(x)$, then we have that $x_i < x$, at every program point where these two variables are alive together.

Definition 1 describes a *Sparse Analysis*. In compiler theory, there are two ways to implement a data-flow analysis; one using a *Dense* and another using a *Sparse* approach. The dense analysis tries to bind information to pairs, each containing a program point and a variable name. On the other hand, a sparse analysis binds information directly to variables. We make use of a sparse representation over a dense one because of two main reasons. First, it simplifies our strategy: it is easier to associate information with a variable name, instead of having to keep track of the different abstract states that a variable may assume along the program's text. Second, it has been shown that sparse analyses tend to be faster and require less memory than their dense counterparts [3, 14, 15, 19].

Sparse analyses are able to associate information to variables because they run on an abstract program representation that incorporates the *Static Single Information* (SSI) property. In summary, this property defines that an abstract state associated with a variable must be invariant along all program points where this variable is alive [19]. To ensure this property, the abstract representation used by our analysis resorts to *live range splitting*. To split the live range of a variable x, at a program point p, we insert a copy $x' = x$ in p, where x' is a fresh name in the program. Then, we rename every use of x at any point p' that is dominated by p to x'.

Different static analyses use different live range splitting strategies to enforce sparsity. The less-than analysis splits live ranges at three different kinds of program sites: definition points, conditional statements and merging points. The first situation lets us extract less-than information from copy assignments and arithmetic operations. The second allows us to extract less-than information from conditionals. Merging points let us extract or refine information in program points where different information may collide.

Example 3.1. Consider the program in Figure 5. We know that `num2` is less than `B`, at line 5, because the conditional in line 4 must be true at line 5. Similarly, we know that `num2` cannot be less than `B` in line 8. Thus, to ensure that the less-than information is invariant for every variable, we need to rename `num2` and `B` at lines 5 and 8. Each new name will be bound to a new set of less-than facts. For instance, assuming that the new name of those variables, at line 5, is `num2_then` and `B_then`, we have that `num2_then` $\in LT($`B_then`$)$.

The computation of less-than facts depends on a range analysis. We define range analysis as follows:

Definition 2 (Range Analysis)**.** For any program variable x, Range Analysis estimates lower and upper values that bound this variable. Range information is given by $R(x) = [l, u], \{l, u\} \subset \mathbb{Z}, l < u$.

To keep this paper as short as possible we do not go any further in explaining the basis on how range analysis works. This background information is presented by Rodrigues *et al.* [16], who are also the authors of the implementation we use. Thus, henceforth, we shall assume that the implementation of our less-than analysis can count on the existence of a mapping R from variables to their ranges.

3.1 Constraint generation

Following Maroua *et. al.*'s approach, we begin our analysis with a constraint generation step. In this stage, we inspect the whole program once, and generate four kinds of constraints: **init, copy, union** and **intersect**. The process of traversing the program to mine constraints is linear on the size of the program. These constraints are extracted from five different syntactic constructions:

1 - Assignments from constants or *unknown* sources: Instructions such as $x_1 = c$ and $x_1 = \bullet$, whereas $c \subset \mathbb{Z}$ and $\bullet$ represent a source of unknown values, e.g., user input, generate an **init** constraint:

$$x_1 = \bullet \mid x_1 = c \rightsquigarrow LT(x_1) = \emptyset$$

2 - Assignments from variables: For example, $x_1 = y_1$ leads to a **copy**, *copy*(x_1, y_1), constraint:

$$LT(x_1) = LT(y_1)$$

3 - Assignments from arithmetic operations: To handle this kind of assignment, e.g., $x_1 = y_1 + z_1$, our analysis needs a bit of extra information – it needs knowledge about *integer ranges*. As described in definition 2, range analysis defines integer intervals that restrict the values assumed by variables. Continuing with the example above, if we find out that the interval that bounds z_1 contains only positive elements, the condition $y_1 < x_1$ will be true, which leads to an **union**, *union*(x_1, y_1), constraint:

$$LT(x_1) = LT(y_1) \cup \{y_1\}$$

On the other hand, if the interval associated to z_1 contains only negative elements, the condition $x_1 < y_1$ will be true, leading to the reordered **union** , *union*(y_1, x_1), constraint.

4 - Conditional statements: This kind of statement, e.g., ($x_1 < x_2$)?, leads to the creation of both **union** and **copy** constraints:

$$(x_1 < x_2)? \begin{cases} T_{branch} : \langle x_{1t}, x_{2t} \rangle \\ F_{branch} : \langle x_{1f}, x_{2f} \rangle \end{cases} \rightsquigarrow \begin{cases} LT(x_{2t}) = \{x_{1t}\} \cup \\ \quad LT(x_2) \cup LT(x_{1t}) \\ LT(x_{1t}) = LT(x_1) \\ LT(x_{2f}) = LT(x_2) \\ LT(x_{1f}) = \\ \quad LT(x_1) \cup LT(x_{2f}) \end{cases}$$

In this case, we split the live range of variables x_1 and x_2 right after the conditional statement. Inside the *true branch* (T_{branch}), we redefine variables x_1 to x_{1t} and x_2 to x_{2t}. Likewise, we redefine x_1 to x_{1f} and x_2 to x_{2f} in the *false branch* (F_{branch}).

5 - Merging points: these points are syntactically represented with ϕ-functions, a notation borrowed from the classic Static Single Assignment form [4]. For example, the first instruction in the `for.cond` block of Figure 3 is a merging point for the variable i $\rightsquigarrow i.0 = \phi(0, inc)$. Merging points result in **intersect** constraints:

$$x = \phi(x_1, \ldots, x_n) \rightsquigarrow LT(x) = LT(x_1) \cap \ldots \cap LT(x_n)$$

This kind of constraint is also used to allow the analysis to work inter-procedurally, albeit not context-sensitively, by creating pseudo-instructions in the form of $x_f = \phi(x_1, ..., x_n)$. These instructions relate a formal parameter x_f to each actual parameter x_i, $1 \leq i \leq n$, found in the text of the program.

Example 3.2. Figure 8 contains a graph representation of the function `fun2` of Figure 5. This graph already shows variables with split live ranges. From this graph our analysis extracts the following constraints: $LT(A_0) = LT(num1_0) \cup \{num1_0\}$; $LT(B_0) = LT(A_0) \cup \{A_0\}$; $LT(B_{0f}) = LT(B_0)$; $LT(B_{0t}) = LT(B_0) \cup LT(num2_{0t}) \cup \{num2_{0t}\}$; $LT(num2_{0t}) = LT(num2_0)$; $LT(B_{0f}) = LT(B_0)$; $LT(num2_{0f}) = LT(num2_0) \cup LT(B_{0f})$; $LT(A_1) = LT(B_{1t}) = undef$; $LT(A_2) = LT(num1_0)$; $LT(B_{1f}) = LT(num2_{0f})$; $LT(A_3) = LT(A_1) \cap LT(A_2)$; $LT(B_2) = LT(B_{1t}) \cap LT(B_{1f})$.

3.2 Region identification

To speedup the propagation of less-than information, we introduce the notion of *region identification*. With this goal, we define *Constraint Graphs* as follows:

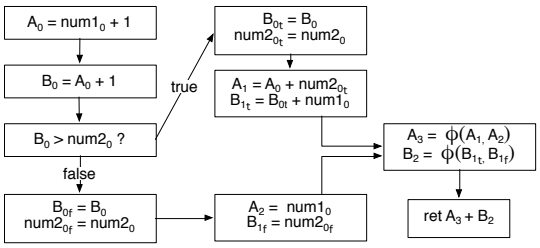

Figure 8: Graph representing the control flow of function `fun2` present in Figure 5.

Definition 3 (Constraint Graph)**.** A constraint graph is a graph G = (N,P,E,X), where N is a set of nodes, composed by variables in the constraint system. P is a set of special nodes, named ϕ nodes, E is a set of edges and X is a set of special edges, which we call ϕ edges. We have two kind of edges that may connect variables u to v: (1) an *equality* edge that derives from **copy** or **union** constraints that only involves LT sets; or (2) a *less-than* edge, which derives from **union** constraints involving both LT sets and single variables. The ϕ edges derive from **intersect** constraints and connect the variables in the right hand side of the constraint to a ϕ node and link this special node to the variable in the left hand side. Henceforth, we shall call connected components in the constraint graph **regions**.

Regions give us the opportunity to answer *some* less-than queries immediately, without having to traverse the constraint graph. We let $id(y)$ imply the region id associated to the variable y. Based on this notation, region identification works as follows: if two variables – x and y – are inside the same region and, consequently, have $id(x) = id(y)$, then there is the possibility of either $x \in LT(y)$ or $y \in LT(x)$. To verify if one of these conditions is valid, we propagate information through our constraint solving step (Section 3.3). Note that this technique does not allow us to assert beforehand that $x \in LT(y)$ or $y \in LT(x)$. On the other hand, if these two variables are enclosed by regions with different ids, then we guarantee that $x \notin LT(y)$ and $y \notin LT(x)$. This last observation helps us to speed up our analysis due to the fact that we avoid propagating less-than information. As we will see in section 4, information propagation is the step with the highest execution cost, since it is based on a worklist algorithm and computes transitive closures.

Example 3.3. Figure 9 contains the constraint graph for `fun1` and `fun2`, from Figures 1 and 5. Regions are marked in grey in the

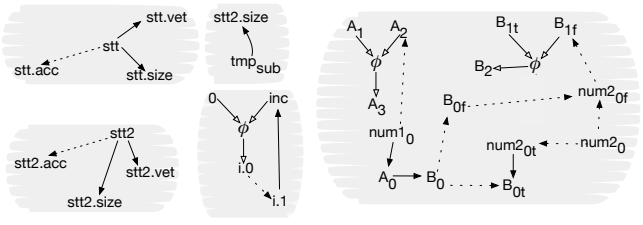

Figure 9: Regions for a program containing both functions `fun1` and `fun2` from figures 1 and 5. Each grey area corresponds to a region with an unique id.

$$constraint(x_1, y_1), e.g., union(x_1, y_1) \text{ or } copy(x_1, y_1)$$

$$case\ 1: \begin{cases} id(x_1) = \emptyset \\ id(x_2) \neq \emptyset \end{cases} \rightsquigarrow id(x_1) = id(x_2)$$

$$case\ 2: \begin{cases} id(x_1) \neq \emptyset \\ id(x_2) = \emptyset \end{cases} \rightsquigarrow id(x_2) = id(x_1)$$

$$case\ 3: \begin{cases} id(x_1) = \emptyset \\ id(x_2) = \emptyset \end{cases} \rightsquigarrow \begin{cases} id = new\ id \\ id(x_2) = id(x_1) = id \end{cases}$$

$$case\ 4: \begin{cases} id(x_1) \neq \emptyset \\ id(x_2) \neq \emptyset \end{cases} \rightsquigarrow unify(\ id(x_2),\ id(x_1)\ )$$

Figure 10: Rules for associating variables to regions. If two variables belonging to regions with different IDs are found to be related, then we *unify* those regions. In a nutshell, this operation maps the two different IDs to a same common region, which we call a *super-region*. Figure 9 only shows super-regions. These rules are also valid for the intersect constraint, which may relate more than two variables.

figure. Dashed arrows represent equality edges and solid arrows represent less-than edges.

Theorem 3.4 (Inclusion Condition). *A variable x is member of the set LT(y), if, and only if there is a directed path in the constraint graph departing from x leading to y, and this path goes across at least one less-than edge. Furthermore, if this path crosses a ϕ node, there must be paths departing from x and reaching such ϕ node from every incoming ϕ edge.*

> **Proof:** Necessity: if there is a directed path from x to y, and this path goes across at least one less-than edge, then there must exist an **union** constraint relating x to y or any other nodes in between these two extremes. Sufficiency: the proof works by induction. On the base case, we have a less-than edge linking x to y. By case analysis on the constraint rules, there must exist a constraint such as $LT(y) = LT(x) \cup \{x\}$. **Extension**: constraints created by ϕ-functions force intersections of abstract information. □

Theorem 3.4 gives us Corollary 3.5. This corollary, in turn, lets us speedup the process to answer less-than queries.

Corollary 3.5. $id(x) \neq id(y) \rightarrow x \notin LT(y) \wedge y \notin LT(x)$

For the sake of simplicity, Figure 7 separates region identification and constraint generation. However, these processes take place simultaneously and, as mentioned in the same figure, the constraint graph is implicit in our problem representation. Immediately after a constraint is extracted from the program, its variables are either inserted into an already existing region or in a new one created for them. The rules for binding variables to regions are illustrated in Figure 10. Notice that the nature of each constraint does not interfere in this process: the sole purpose of constraints in this stage is to establish connections among variables.

3.3 Lazy calculation

As Figure 7 indicates, we propagate information Lazily, in a process we identify as local solving. Our analysis waits until client applications perform alias queries, requesting information about LT sets of pointers, in order to construct such sets. We propagate information through a worklist algorithm. Each variable x of the target region has its LT set initialized to V, the set of variables assigned to the region $id(x)$. The worklist algorithm, then, starts to iterate over a specific constraint set trying to remove elements from each one of them, until a fixed point is reached. Maroua *et al* have shown that a fixed point is indeed reached while information is being propagated [9, Sec. 3.5]. The specific constraint set, mentioned before, is composed by two kinds of constraints: those directly related to the pointer for which LT information is being required, and those related to a source of LT information encapsulated by the same region. The fact that we only propagate information when the target variables are within the same region, and only try to select subsets of constraints from said region, is the major reason we could achieve better runtime for our analysis, as we will demonstrate in the next section.

4 EVALUATION

In this section, we answer the following research questions:

- **RQ1:** What is the benefit in runtime provided by the demand-driven approach presented in this paper?
- **RQ2:** Is there any case in which our technique leads to worse results than a closure-based traditional implementation?
- **RQ3:** How do we compare, in terms of precision, against the closure-based approach?
- **RQ4:** How does our approach compare, in terms of memory consumption, against the closure-based approach?

To provide answers to these questions, we have implemented our analysis in LLVM version 3.7, and have compared it against the implementation of Maroua *et al.* [9], available in the ACM digital library as an artifact submitted to the CGO artifact evaluation committee. Our hardware consists of a quad-core Intel(R) i7-3770 at 3.4 GHz, with 16GB of RAM, featuring Linux Ubuntu 14.04.

Our benchmarks are the 20 C/C++ programs taken from the SPECCPU 2006 suite. This collection contains a range of real world applications such as libraries for video processing, error estimation, compilers and interpreters of programming languages. We have selected three different client analyses present in LLVM to demonstrate how our demand driven approach works alongside individual code optimizations. The first client analysis is *Loop invariant code motion*. This optimization moves unnecessary code outside a loop whenever possible. The second client, *Combination of redundant instructions*, is a code transformation that reduces the amount of instructions of a given program by re-combining them into fewer, simple instructions. The last client, *Dead store elimination*, eliminates redundant store operations. Following the LLVM nomenclature, we shall use the abbreviations *licm, instcombine* and *dse* to identify each client.

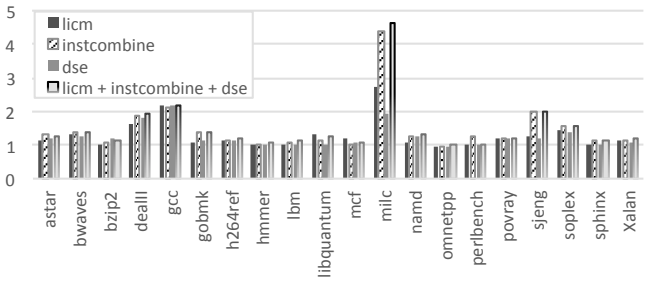

Figure 11: Runtime improvement obtained with our demand-driven analysis, while handling queries from the `licm`, `instcombine` and `dse` passes. The line identified by the number 1 is a baseline that indicates the runtime of the traditional worklist solver. Values above this line indicate a better runtime performance of our algorithm.

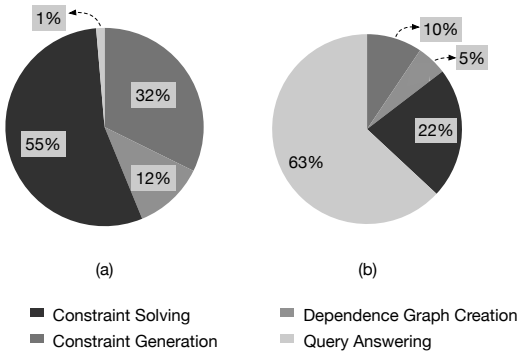

Figure 12: Time consumption distributed among the stages of the less-than analysis. (a) shows the result for *licm, dse,* and *instcombine* and (b) for *aa-eval.*

4.1 Discussion of Results

RQ1: Figure 11 illustrates the runtime results for each benchmark in our test-suite. For the optimizations *licm*, *instcombine*, and *dse* we were able to reach average runtime improvements of 30%, 48%, and 27% respectively, over the implementation of Maroua *et al* [9]. For the largest benchmarks in size: *dealII*, *gcc* and *Xalan*, runtime savings reached up to 68%. We have also analyzed the impact of such optimizations running together, and we observed an average performance gain of 51%. This perceived improvement is due to two reasons. First, these optimizations request *LT* information of few pointers when compared to the whole universe of possible requests. For example, *licm* makes queries related to pointers that are either within loops or correlated with instructions inside those loops. This set represents only 10% of the queries that could be performed over our benchmarks set. Likewise, *instcombine* only queries pointers in instructions with potential to be combined. In this case, queries represent only 4% of the universe of possible inquiries. The more *LT* sets are consulted, the more computations we are expected to perform. The second reason for our better performance is the fact that constraint solving is the most time consuming phase of the less-than analysis. For each code optimization present in LLVM, the process of constraint solving consumed at least 54% of the total time in general benchmarks and up to 85% for the largest one. This stage is exactly the one that our on-demand technique improves.

RQ2: The time consumed to run a less-than analysis may be divided into four stages: (1) constraint generation, (2) dependence graph construction, (3) constraint solving, and (4) query answering[1]. For the cases where constraint solving does not have the highest impact in the analysis runtime, our algorithm is not able to provide considerable improvement upon the closure-based approach. As an example, we consider the Alias Analysis Precision Evaluator, *aa-eval*, present in the LLVM. This client, as its name implies, evaluates the precision of different alias analysis algorithms, by trying to disambiguate every pair of pointers in a program. When serving this client, the most time consuming stage of the less-than analysis is query answering, as outlined in Figure 12. In this case, our demand-driven technique achieved a general improvement of only 9% over the more traditional approach. For some benchmarks, such as *Xalan* and *mcf*, our technique led to slowdowns of 2% and 4%, with a maximum observed value of 16% in the *omnetpp* benchmark. Such regressions are due to the overhead of checking that two pointers are not grouped in the same connected region of the constraint graph. This overhead has not surfaced for the compiler optimizations that we have evaluated; however, *aa-eval* makes it evident, because it queries every possible combination of pointers within a function.

RQ3: As our demand-driven algorithm only uses a subset of constraints within a specific region to propagate information, one may think that our technique could be less precise than Maroua *et al.*'s. However, the fact that we use constraints that represent sources of *LT* information to feed our worklist solver gives us the same results as those obtained with the traditional approach. We preserve precision because less-than facts are propagated to every reachable variable in the constraint graph. We say that two pointers do not alias only if there is a directed path between them in the constraint graph. Under these circumstances, Theorem 3.4 already ensures that we preserve less-than information. We have empirically checked the results from ours and Maroua's implementation, and, as expected, they provide the same answers to alias queries for every combination of benchmark and clients. Furthermore, the binaries that LLVM produced for SPECCPU 2006 were identical, regardless of which implementation of the less-than analysis we have used.

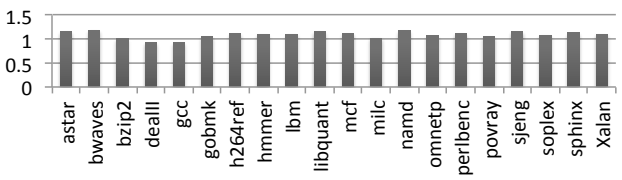

Figure 13: Overhead in memory usage imposed by our demand-driven technique

[1]To keep this paper as short as possible, we decided to omit any reference to the dependence graph created by our approach. This structure is also created by Maroua *et. al.* and both are equivalent in implementation and runtime.

RQ4: In general, our demand-driven technique increases memory consumption when compared with the closure-based approach. Figure 13 shows this increase. On average, we use 9% more memory than Maroua *et al.*'s implementation. We use more memory mainly because we need a separate data-structure to group constraints in different regions.

5 RELATED WORKS

Over the years, many different static analyses have used the less-than domain to demonstrate the relationship between the variables of a program. With this purpose, Logozzo *et. al.* [7] have introduced *Pentagons*, a relational abstract domain that includes a mix of the less-than analysis with integer range of variables. Their domain is very similar to the one we have used in this paper and can be used as well to infer less-than information like we do. Nevertheless, their approach differs from ours, because while we split our algorithm in two stages – pre-processing and querying – they adopt an approach based on the construction of a transitive closure.

Bodik *et. al.* [2] use a strategy that shares some similarities with ours. They also try to solve less-than relations through a demand-driven algorithm. But, differently from our approach, they use a graph interpretation in order to express the less-than relations among variables. Their analysis also differs from ours in terms of performing a LT check. While we say that a variable x is less than y if $x \subset LT(y)$, they answer positive to a check if there exists a path, ideally the shortest, between a source node s and a target node t in their graph. They try to find this path by means of a brute-force depth-first exploration of the graph. Another difference between our approach and theirs it that we make use of range information to improve the accuracy of our analysis.

The work developed by Maroua *et. al.* [9] is the one closest to ours. We share with them all the fundamental steps of creating and solving less-than relations through a worklist algorithm. The differences between their work and ours are the contributions we have presented in section 3. In their original approach, they create less- than constraints, solve them one-by-one and then store the information generated, so as to answer queries from other analyses in $O(1)$. We have modified the step that creates constraints, and have moved the whole resolution step to a demand-driven approach. During constraint creation, we also include a process to generate *Region identifiers*, which we use to prune our iteration space. We only start the constraint solving step if we identify that a client analysis requires it. We emphasize that these differences are not simply due to engineering decisions. To solve constraints on demand, we had to extract structure from them, i.e., we had to model the constraint system as a constraint graph, and then define regions in a pre-processing phase. This pre-processing lets us avoid checking variables that could not be related by less-than information in our constraint system.

6 CONCLUSION

This paper has presented a new strategy, based on a partially demand-driven approach, that is able to solve inequalities of a less-than constraint system. The approach is said to be partially demand-driven, because it combines a pre-processing stage, which generate constraints, with a querying stage, which works on-demand and is responsible for solving constraints. We have used our less-than analysis to disambiguate pointers, following the technique introduced by Maroua *et al.* [9]. Our experimental results lead us to believe that this approach is effective and useful, when compared to the current state-of-the-art implementations of less-than analyses. We speculate that our ideas could benefit other static analyses as well. There are several different static analyses, such as alias analysis, and rapid type analysis, which compare information about pairs of elements. We believe that the idea of grouping constraints into different regions could be used to speed them up. For example, optimizations that could be solved in a potentially more efficient way, given these ideas, include elimination of array bound checks and detection of data races in parallel programs. We leave these possibilities as work that we hope to explore in the future.

REFERENCES

[1] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime Pointer Disambiguation. In *OOPSLA*. ACM, New York, NY, USA, 589–606.

[2] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: eliminating array bounds checks on demand. In *PLDI*. ACM, New York, NY, USA, 321–333.

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *POPL*. ACM, New York, NY, USA, 55–66.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *POPL*. ACM, New York, NY, USA, 25–35.

[5] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[6] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE, Washington, DC, USA, 75–.

[7] Francesco Logozzo and Manuel Fähndrich. 2008. Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses. In *SAC*. ACM, New York, NY, USA, 184–188.

[8] Francesco Logozzo and Manuel Fähndrich. 2010. Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses. *Sci. Comput. Program.* 75 (2010), 796–807.

[9] Maroua Maalej, Vitor Paisante, Pedro Ramos, Laure Gonnord, and Fernando Magno Quintão Pereira. 2017. Pointer Disambiguation via Strict Inequalities. In *CGO*. IEEE, Piscataway, NJ, USA, 134–147.

[10] Ian Munro. 1971. Efficient determination of the transitive closure of a directed graph. *Inform. Process. Lett.* 1, 2 (1971), 56–58.

[11] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2014. Validation of Memory Accesses Through Symbolic Analyses. In *OOPSLA*. ACM, New York, NY, USA, 791–809.

[12] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2005. *Principles of program analysis*. Springer, New York, NY, USA.

[13] Gerald Penn. 2006. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science* 354, 1 (2006), 72–81.

[14] G. Ramalingam. 2002. On Sparse Evaluation Representations. *Theor. Comput. Sci.* (2002), 119–147.

[15] Andrei Rimsa, Marcelo d'Amorim, and Fernando Magno Quintão Pereira. 2011. Tainted Flow Analysis on e-SSA-form Programs. In *CC/ETAPS*. Springer-Verlag, Berlin, Heidelberg, 124–143.

[16] Raphael Ernani Rodrigues, Fernando Magno Quintao Pereira, and Victor Hugo Sperle Campos. 2013. A Fast and Low-overhead Technique to Secure Programs Against Integer Overflows. In *CGO*. IEEE, Washington, DC, USA, 1–11.

[17] Radu Rugina and Martin Rinard. 2000. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *PLDI*. ACM, New York, NY, USA, 182–195.

[18] Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2016. Restrictification of Function Arguments. In *CC*. ACM, New York, NY, USA, 163–173.

[19] André Tavares, Benoit Boissinot, Fernando Pereira, and Fabrice Rastello. 2014. Parameterized construction of program representations for sparse dataflow analyses. In *CC*. Springer, Grenoble, France, 18–39.

[20] B-F Wang and G-H Chen. 1990. Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. *IEEE Transactions on Parallel and Distributed Systems* 1, 4 (1990), 500–507.