# Chapter 8

# Experimental Results and Practical Uses of Points-to Analysis

In this chapter, we demonstrate the effectiveness of our points-to analysis by providing some experimental results. We also discuss how other analyses implemented in the framework of the McCAT compiler benefit from our points-to information. The experimental results obtained by running the analysis on some standard benchmarks are presented in Section 8.1. In Section 8.2, we discuss other analyses which are currently using the results of our analysis.

## 8.1 Experimental Results

This section contains our experimental results on some standard benchmarks. Table 8.1 contains the following information about the benchmark files:

1. Benchmark: is the name of the benchmark
2. Lines: is the source lines including comments
3. #of stmts in SIMPLE: is the number of C-statements in the SIMPLE representation excluding the comments
4. Min #of var: is the minimum number of variables among all abstract stacks (this also includes the invisible variables and all the fields in case of a structure.)
5. Max #of var: is the maximum number of variables among all abstract stacks (this also includes the invisible variables and all the fields in case of a structure.)

| Benchmark | Lines | #of stmts in SIMPLE | Min #of var | Max #of var | Description |
|---|---|---|---|---|---|
| genetic | 506 | 479 | 33 | 61 | Implementation of a genetic algorithm for sorting. |
| dry | 826 | 212 | 21 | 43 | Dhrystone benchmark. |
| clinpack | 1231 | 920 | 11 | 109 | The C version of Linpack. |
| config | 2279 | 4549 | 19 | 188 | Checks all the features of the C-language. |
| toplev | 1637 | 1096 | 92 | 164 | The top level of GNU C compiler. |
| compress | 1923 | 1342 | 41 | 186 | UNIX utility program. |
| mway | 700 | 869 | 51 | 125 | A unified version of the best algorithms for m-way partitioning. |
| hash | 256 | 110 | 15 | 30 | An implementation of a hash table. |
| misr | 276 | 235 | 10 | 43 | This program creates two MISR's. Their values are compared to see if the introduced errors have cancelled themselves. |
| xref | 146 | 140 | 26 | 61 | A cross-reference program to build a tree of items. |
| stanford | 885 | 889 | 31 | 67 | Stanford baby benchmark. |
| fixoutput | 400 | 391 | 17 | 31 | A simple translator. |
| sim | 1422 | 1768 | 99 | 137 | Finds local similarities with affine weights. |
| travel | 862 | 543 | 28 | 55 | Implements Traveling Salesman Problem with greedy heuristics. |
| csuite | 872 | 781 | 34 | 55 | Part of test suite for Vectorizing C compilers. |
| msc | 148 | 226 | 20 | 73 | Calulates the min spanning circle of a set of n points in the plane. |
| lws | 2239 | 6671 | 64 | 527 | Implements dynamic simulation of flexible water molecule. |

Table 8.1: Characteristics of benchmark files.

6. Description: is a brief description of the benchmark

Note that each function has its own bit-matrix where the size of the matrix depends on the total number of variables involved in the points-to analysis. The columns 'Min #of var' and 'Max #of var' give an idea of the size of this matrix. For example, the largest bit-matrix size for the program 'config' is equal to 188. Assuming that each word is composed of 32 bits, six words are needed to store one row of this bit matrix.

All pointer type variables are initialized to NULL. This initialization increases the number of points-to relationships. Since these relationships are not necessarily initialized by the user, the points-to information contributed by this initialization is not counted. For pointer arithmetic it is assumed that pointers to arrays do not cross array boundaries. For non-array pointer arithmetic a flag is supported that indicates either: (1) the pointer target stays within the presently pointed-to data structure, or (2) the pointer target can be any memory location. For the first case a warning message is issued so that the programmer can determine if this is a safe assumption. The experiments were done with this setting. The relevant statistics is provided in Tables 8.2, 8.3, 8.4, and 8.5. We first analyze Table 8.2 which essentially contains the number of indirect references in a program. This table represents the actual use of points-to information and contains the following items (from left to right):

1. Benchmark: is the name of the benchmark.
2. 1 D rel: is the number of indirect references with the dereferenced pointer definitely points to a single stack location.
3. 1 P rel: is the number of indirect references with the dereferenced pointer possibly points to a single stack location, the other being NULL (if any).
4. 2 P rel: is the number of indirect references with the dereferenced pointer possibly points to two stack locations.
5. 3 P rel: is the number of indirect references with the dereferenced pointer possibly points to three stack locations.
6. $\geq 4$ P rel: is the number of indirect references with the dereferenced pointer possibly points to four or more stack locations.
7. Ind refs: is the total number of indirect references in the program.
8. Replace: is the number of indirect references that can be replaced by a direct reference, by using definite points-to information.
9. To stk: is the number of points-to information used by indirect references, with the pointer target being on the stack.
10. To heap: is the number of points-to information used by indirect references, with the pointer target being on the heap.
11. Tot: is the sum of the previous two columns ('To stack' and 'To heap').
12. Avg: is the average number of points-to information used in resolving an indirect reference in the program, which is the same as the average number of stack locations pointed to by the dereferenced pointer

For each multiple entry column, the first entry provides statistics for indirect references of the form `*x` and `(*x).y.z`, and the second entry provides statistics for indirect references of the form `x[i][j]`, where `x` is a pointer to an array.

The results in this table are generally very impressive. This indicates the effectiveness of our point-to analysis. Following are some specific conclusions derived from this table:

| Bench- mark | 1 D rel | | 1 P rel | | 2 P rel | | 3 P rel | | $\geq 4$ P rel | | Ind refs | Rep- lace | To stk | To heap | Tot | Avg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| genetic | 2 | 5 | 14 | 27 | 0 | 2 | 0 | 2 | 2 | 0 | 54 | 7 | 38 | 30 | 68 | 1.26 |
| dry | 2 | 11 | 37 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 58 | 9 | 21 | 45 | 66 | 1.14 |
| clinpack | 7 | 98 | 0 | 2 | 0 | 39 | 0 | 4 | 0 | 0 | 150 | 101 | 197 | 0 | 197 | 1.31 |
| config | 8 | 3 | 34 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 45 | 3 | 45 | 0 | 45 | 1.00 |
| toplev | 5 | 0 | 110 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 117 | 5 | 171 | 0 | 171 | 1.46 |
| compress | 0 | 0 | 40 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 50 | 0 | 43 | 7 | 50 | 1.00 |
| mway | 31 | 38 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 74 | 0 | 79 | 0 | 79 | 1.07 |
| hash | 2 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 14 | 0 | 7 | 7 | 14 | 1.00 |
| misr | 1 | 3 | 8 | 0 | 27 | 0 | 0 | 0 | 0 | 0 | 39 | 0 | 31 | 35 | 66 | 1.69 |
| xref | 0 | 0 | 20 | 2 | 9 | 0 | 0 | 0 | 0 | 0 | 31 | 0 | 9 | 31 | 40 | 1.29 |
| stanford | 6 | 61 | 74 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 143 | 51 | 119 | 26 | 145 | 1.01 |
| fixoutput | 5 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 5 | 5 | 3 | 8 | 1.00 |
| sim | 0 | 0 | 122 | 231 | 0 | 0 | 0 | 0 | 0 | 0 | 353 | 0 | 34 | 319 | 353 | 1.00 |
| travel | 0 | 20 | 3 | 0 | 32 | 17 | 4 | 1 | 0 | 0 | 77 | 20 | 125 | 11 | 136 | 1.77 |
| csuite | 8 | 13 | 36 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 66 | 21 | 64 | 2 | 66 | 1.00 |
| msc | 6 | 0 | 35 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 41 | 6 | 6 | 35 | 41 | 1.00 |
| lws | 77 | 90 | 54 | 197 | 5 | 0 | 0 | 0 | 0 | 0 | 423 | 110 | 428 | 0 | 428 | 1.01 |

Table 8.2: Points-to statistics for indirect references.

- The average number of stack locations pointed to by the dereferenced pointer in an indirect reference is quite close to one for most programs, where one indicates the best possible case. The overall average is equal to 1.13, while the maximum average for a program is only 1.77. This indicates that the information collected by our points-to analysis is highly precise.
- Overall, 28.80% of indirect references in the programs have the dereferenced pointer definitely pointing to a single stack location. Using the definite information, 19.39% of indirect references, can be replaced by direct references (when the dereferenced pointer does not point to an invisible variable). For this replacement, 67.33% of definite relationships applicable to indirect references, are useful. These are very important results, which support our strategy of collecting both possible and definite information.

  Note that the fact that replacement is not considered for an invisible variable does not mean that the definitely-points-to information for invisible variables can not be useful. As an example, they are useful for register allocator when the life time of keeping a variable in a register can become longer for the variables that have definitely-points-to relationship. They will be also useful in propagating any data flow information from callee to caller.

- A pointer should not be pointing to NULL when being dereferenced during program execution. With this assumption, 90.76% of indirect references have the dereferenced pointer definitely/possibly pointing to a single stack location.
- For certain benchmarks, in particular 'clinpack', 'stanford', 'sim' and 'lws', the majority of definite relationships are for indirect references of the form `x[i][j]`, (where `x` is a pointer to an array). This information is very useful for array dependence analysis, as it reduces the number of array pairs to be collected for subscript analysis [Jus93].
- Overall, 27.92% of points-to relationships used, have heap locations as the pointer target. This underlines the need for a powerful companion heap analysis to identify disjoint accesses to heap locations [Ghi93b].
- There are very few cases with three or more possible points-to relationships for an indirect reference. One of these cases (with more than 4 points-to relationships), occurs when an array of pointers is initialized (in 'toplev').

Table 8.3 further categorizes the points-to relationships with the pointer target on stack (column 'To stk' of table 8.2). It contains the following items (from left to right):

1. Benchmark: is the name of the benchmark.
2. From lo: is the points-to relationships from local variables.
3. From gl: is the points-to relationships from global variables.
4. From fp: is the points-to relationships from formal parameters.
5. From iv: is the points-to relationships from invisible variables.
6. To lo: is the points-to relationships to local variables.
7. To gl: is the points-to relationships to global variables.
8. To fp: is the points-to relationships to formal parameters.
9. To iv: is the points-to relationships to invisible variables.

The statistics in the table show that most of the relationships arise from formal parameters, and are directed to invisible variables or global variables. This indicates that procedure calls generate the majority of points-to relationships.

Table 8.4 contains statistics about the total number of collected points-to information, obtained by adding up the number of pairs valid at each statement in the *simplified* program. It classifies the points-to pairs based on their origin and target in the memory organization. It contains the following items (from left to right):

| Bench-mark | From | | | | To | | | |
|---|---|---|---|---|---|---|---|---|
| | lo | gl | fp | iv | lo | gl | fp | iv |
| genetic | 0 | 0 | 38 | 0 | 0 | 38 | 0 | 0 |
| dry | 0 | 0 | 21 | 0 | 0 | 9 | 0 | 12 |
| clinpack | 0 | 0 | 197 | 0 | 0 | 193 | 0 | 4 |
| config | 3 | 0 | 42 | 0 | 3 | 33 | 0 | 9 |
| toplev | 0 | 0 | 171 | 0 | 0 | 171 | 0 | 0 |
| compress | 11 | 3 | 29 | 0 | 0 | 40 | 0 | 3 |
| mway | 0 | 0 | 79 | 0 | 0 | 5 | 0 | 74 |
| hash | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| misr | 23 | 0 | 8 | 0 | 0 | 0 | 0 | 31 |
| xref | 0 | 0 | 9 | 0 | 0 | 9 | 0 | 0 |
| stanford | 0 | 0 | 119 | 0 | 0 | 103 | 0 | 16 |
| fixoutput | 0 | 0 | 5 | 0 | 0 | 5 | 0 | 0 |
| sim | 15 | 0 | 19 | 0 | 0 | 26 | 0 | 8 |
| travel | 2 | 0 | 123 | 0 | 14 | 57 | 0 | 54 |
| csuite | 12 | 0 | 52 | 0 | 8 | 56 | 0 | 0 |
| msc | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| lws | 0 | 0 | 428 | 0 | 1 | 350 | 0 | 77 |

Table 8.3: Categorization of points-to information used by indirect references.

1. Benchmark: is the name of the benchmark.
2. Stack To Stack: is the number of points-to relationship from stack to stack.
3. Stack To Heap: is the number of points-to relationship from stack to heap.
4. Heap To heap: is the number of points-to relationship from heap to heap.
5. Heap To Stack: is the number of points-to relationship from heap to stack.
6. Avg: is the average number of points-to relationship valid at a statement.
7. Max per stmt: is the maximum number of points-to relationship valid at a statement.

The major observation from this table is the absence of points-to relationships from heap to locations on stack, implying that pointers in heap objects only point to other heap objects, for the given benchmark set. This supports our strategy of separating stack and heap points-to analyses.

Finally, Table 8.5 gives the following measurements for invocation graphs:

1. Benchmark: is the name of the benchmark.
2. ig nodes: is the number of nodes in the invocation graph.

| Bench-mark | Stack To Stack | Stack To Heap | Heap To Heap | Heap To Stack | A v g | Max per stmt |
|---|---|---|---|---|---|---|
| genetic | 3901 | 1066 | 0 | 0 | 10 | 38 |
| dry | 512 | 883 | 198 | 0 | 7 | 24 |
| clinpack | 18987 | 0 | 0 | 0 | 20 | 91 |
| config | 136315 | 18 | 0 | 0 | 29 | 120 |
| toplev | 41539 | 6 | 0 | 0 | 37 | 100 |
| compress | 30502 | 1070 | 0 | 0 | 23 | 82 |
| mway | 16399 | 0 | 0 | 0 | 18 | 76 |
| hash | 577 | 207 | 34 | 0 | 7 | 18 |
| misr | 1314 | 706 | 9 | 0 | 8 | 25 |
| xref | 46 | 506 | 17 | 0 | 4 | 16 |
| stanford | 3137 | 364 | 7 | 0 | 3 | 30 |
| fixoutput | 3111 | 794 | 0 | 0 | 9 | 14 |
| sim | 7048 | 31174 | 1437 | 0 | 22 | 47 |
| travel | 3581 | 1174 | 0 | 0 | 8 | 42 |
| csuite | 4527 | 14 | 0 | 0 | 5 | 26 |
| msc | 221 | 907 | 88 | 0 | 5 | 22 |
| lws | 241291 | 0 | 0 | 0 | 35 | 366 |

Table 8.4: General points-to statistics.

3. call sites: is the number of call-sites in the program.
4. # of fns: is the number of functions actually called in the program.
5. R: is the recursive nodes in the invocation graph.
6. A: is the approximate nodes in the invocation graph.
7. Avgc: is the average number of nodes in the invocation graph per call-site.
8. Avgf: is the average number of nodes in the invocation graph per called function.

The overall average for the number of invocation graph nodes per call-site is 1.45. Thus each call-site appears on an average on two call-chains for our benchmark set. This indicates that our approach of explicitly following call-chains is practical for real programs of moderate size, though it is theoretically exponential in cost. However, to fully support this claim, we need to do further experimentation on larger benchmarks. If the size of the invocation graph becomes unreasonable on such benchmarks, we plan to reduce its size by sharing sub-trees that have the same or similar invocation contexts. This can be implemented by caching the input and output points-to information for each function, and by recognizing when a particular input has already occurred. If the output has already been computed, then the sub-trees can be shared

| Bench-mark | ig nodes | call sites | #of fns | R | A | Avgc | Avgf |
|---|---|---|---|---|---|---|---|
| genetic | 45 | 32 | 17 | 0 | 0 | 1.38 | 2.65 |
| dry | 19 | 17 | 14 | 0 | 0 | 1.06 | 1.36 |
| clinpack | 92 | 42 | 11 | 0 | 0 | 2.17 | 8.36 |
| config | 1068 | 493 | 49 | 0 | 0 | 2.17 | 21.80 |
| toplev | 53 | 29 | 18 | 0 | 0 | 1.80 | 2.94 |
| compress | 45 | 23 | 12 | 0 | 0 | 1.91 | 3.75 |
| mway | 44 | 42 | 21 | 0 | 0 | 1.02 | 2.10 |
| hash | 9 | 8 | 5 | 0 | 0 | 1.0 | 1.80 |
| misr | 8 | 7 | 5 | 0 | 0 | 1.0 | 1.60 |
| xref | 15 | 14 | 8 | 2 | 4 | 1.0 | 1.88 |
| stanford | 64 | 61 | 37 | 6 | 10 | 1.03 | 1.73 |
| fixoutput | 23 | 12 | 6 | 0 | 0 | 1.83 | 3.83 |
| sim | 120 | 47 | 15 | 2 | 8 | 2.53 | 8.00 |
| travel | 39 | 22 | 14 | 2 | 4 | 1.73 | 2.79 |
| csuite | 37 | 36 | 36 | 0 | 0 | 1.00 | 1.00 |
| msc | 6 | 5 | 5 | 2 | 2 | 1.00 | 1.00 |
| lws | 33 | 29 | 17 | 0 | 0 | 1.10 | 1.94 |

Table 8.5: Invocation graph statistics.

and the stored output can be used to continue the analysis.

## 8.2 Practical Applications of Points-to Analysis

Points-to analysis acts as the initial step for many other analyses performed by the McCAT compiler and improves their accuracy. In the following, we briefly explain some of these analyses which are developed by other members of our research group. We summarize the impact of accurate points-to information on their methods.

Section 8.2.1 is on dependence analysis for the ALPHA representation [HGS92]. Section 8.2.2 is on extending the invocation graph (relatively points-to analysis) in the presence of function pointers parameter [Ghi92]. Section 8.2.3 is on generalized constant propagation [LH93]. Section 8.2.4 is on array dependence analysis [Jus93]. Section 8.2.5 is on reaching definition analysis [Sri92]. Section 8.2.6 is on live variable analysis [Sri92]. Section 8.2.7 is on loop unrolling which is implemented as a course project by M. Iarocci and C. Pateras.

### 8.2.1 Dependence Analysis for ALPHA

In this section, we briefly present the work of Hendren, Gao, and Sreedhar on ALPHA [HGS92]. ALPHA is an intermediate representation that allows one to perform sparse analyses. The first step in constructing ALPHA is to generate read/write sets for each statement using the points-to information. The read/write set specifies all the abstract stack location read/written at each statement in SIMPLE. The way to generate this information for pointer variables is to follow the links in the abstract stack of the points-to analysis. This is done using the rules given in Table 8.6 where $s$ is a statement, $PI$ is the points-to information, $W_p(s, PI)/W_d(s, PI)$ is the possible/definite write at $s$ with $PI$ as input, $R(s, PI)$ is the read set at $s$ with $PI$ as input, $P_p(x, PI)/P_d(x, PI)$ is the set of variables that $x$ definitely/possibly-points-to in $PI$, and $P(y, PI)$ is the set of variables that $x$ definitely/possibly-points-to in $PI$.

| $s$ | $W_p(s, PI)$ | $W_d(s, PI)$ | $R(s, PI)$ |
|---|---|---|---|
| `x = y` | $\phi$ | {`x`} | {`y`} |
| `x =` *unop* `y` | $\phi$ | {`x`} | {`y`} |
| `x = y` *binop* `z` | $\phi$ | {`x`} | {`y`,`z`} |
| `*x = y` | $P_p(x, PI)$ | $P_d(x, PI)$ | {`x`,`y`} |
| `*x =` *unop* `y` | $P_p(x, PI)$ | $P_d(x, PI)$ | {`x`,`y`} |
| `*x = y` *binop* `z` | $P_p(x, PI)$ | $P_d(x, PI)$ | {`x`,`y`, tt z} |
| `x = &y` | $\phi$ | {`x`} | $\phi$ |
| `x = *y` | $\phi$ | {`x`} | {`y`} $\cup P(y, PI)$ |

Table 8.6: Read and Write Sets for statement $s$ relative to point-to information $PI$

The example represented in Figure 8.1(a) shows a C-program, Figure 8.1(b) shows its SIMPLE representation, Figure 8.1(c) shows the related abstract stack, and Figure 8.1(d) shows the read/write set related to each statement. Now we have a closer look at statement 4. Clearly `temp0` will be in the read set. All the location(s) that `temp0` is pointing to will be in the write set, in this case variable `c`.

Once the read/write set is computed for each statement, *exposure analyses* are performed. Exposure analyses, which consist of *willbe_read*, *willbe_written*, *was_read* and *was_written*, identify program points where $\alpha$-nodes are introduced. The $\alpha$-nodes essentially capture data-dependences crossing different control regions. Informally, $\alpha$-nodes are like *identity assignments* (e.g. x = x). Once $\alpha$-nodes are inserted, dependence analysis is performed to generate dependence information and the resulting representation is called the ALPHA representation.

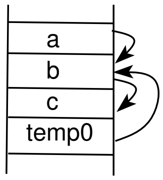
The accuracy of dependence information depends on the points-to information.

```
main() {
    int **a, *b, c ;
    a = &b ;
    b = &c ;
    **a = 7 ;
}
```

(a) A C-program

```
main() {
    int **a, *b, c ;
    int temp0 ;
    a = &b ;      /* stmt1 */
    b = &c ;      /* stmt2 */
    temp0 = *a ; /* stmt3 */
    *temp0 = 7 ; /* stmt4 */
}
```

(b) The SIMPLE representation




(c) Stack Matrix

| S | Write(S) | Read(S) |
|---|---|---|
| `a = &b` | {a} | $\phi$ |
| `b = &c` | {b} | $\phi$ |
| `temp0 = *a` | {temp0} | {a, b} |
| `*temp0 = 7` | {c} | {temp0} |

(d) Read/Write Sets

Figure 8.1: A simple example of dependence analysis.

Without points-to information, one has to make conservative assumptions regarding the dependence information and this in turn generates a 'denser' ALPHA dependence representation.

### 8.2.2 Function Pointer Parameters

The presence of function pointers complicates the interprocedural analysis. Function pointers are special pointer variables which point to functions. These are referenced at run time to call the pointed-to functions. Unlike a normal function call, a call through a function pointer can not be bounded to a unique function at compilation time. The function invoked from such a call-site depends on the address contained in the function pointer when program execution reaches that point. Thus different functions can be invoked from the same call-site in different executions of the program. Under these circumstances, the invocation graph becomes a dynamic property of the program and can not be constructed by a simple pass through the program.

However, a precise and safe static approximation of the invocation graph can still be obtained using flow-analytic techniques. The key idea is that the set of functions invocable from a function pointer call-site is identical to the set of functions that the function pointer can point to at the program point just before the call-site. If the complete points-to information for each function pointer is available at the corresponding function pointer call-site, the invocation graph can be effectively constructed. In this case, different interprocedural analyses can be efficiently performed. To provide precise and useful information, points-to analysis itself needs to be interprocedural. The problem is how to build the invocation graph for points-to analysis itself.

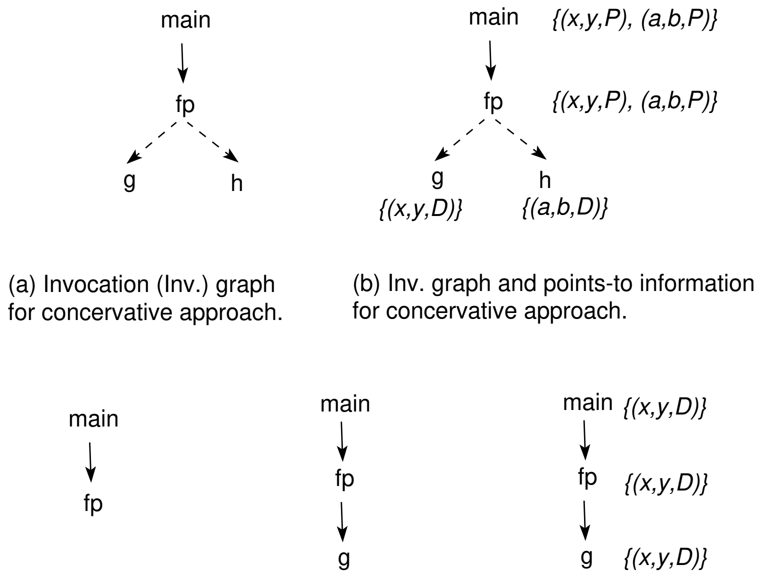
The solution lies in updating the invocation graph while performing the points-to analysis. First, the invocation graph is constructed using the normal depth-first algorithm described in Chapter 4. This construction is left incomplete at the points where a function-pointer-call is encountered. Next, points-to analysis is performed using this incomplete invocation graph. Once encountering a function pointer call, all the functions it can point to according to the current points-to information are determined. The invocation graph is updated accordingly to indicate that all these functions are invocable from the given call-site. The points-to analysis assumes all these functions to be invocable from the call-site under consideration and merges their output points-to sets to compute the points-to information at the program point after this call. Thus the points-to analysis keeps updating the invocation graph, while calculating the points-to information, and constructs the complete invocation graph to be used by other interprocedural analyses.

Consider the example shown in Figure 8.2. Without points-to information, the actual function call(s) for `fp` at statement `S2` is not known, therefore, a conservative approach should be taken which says that `fp` can be all the possible functions in the program (in this case `f` and `g`). The related invocation graph for the worst case is represented in Figure 8.2(a). Figure 8.2(b) represents the invocation graph decorated with the points-to information after the completion of points-to analysis. As it is shown in this figure, in case of having more than one possible case for a function pointer, the result from different calls should be merged. This merging results in some extra points-to information that will propagate an inaccurate result throughout the analysis.

Using the points-to information, the invocation graph is completed at the same time as the points-to analysis. First an incomplete invocation graph is built, as it is shown in Figure 8.2(c). Next, points-to analysis is performed. After encountering the function pointer call `fp()` at statement `S2`, the points-to information (fp,g,D) is used for updating the invocation graph, by adding the call `g` as the child of `fp`. The result is represented in Figure 8.2(d). Next is to follow the newly added call to continue the points-to analysis. This results in a more accurate points-to information that is represented in Figure 8.2(e).

Note that the structured nature of points-to analysis plays an important role in efficient handling of function pointers without requiring any extra pass through the

```
int *x, y ;
int *a, b ;
  main() {
     void  (*fp)() ;
S1:  fp = g ;
     ...
S2:  fp() ;
  }
  g()
  {
      x = &y ;
  }
  h()
  {
      a = &b
  }
```




(a) Invocation (Inv.) graph for concervative approach.

(b) Inv. graph and points-to information for concervative approach.

(c) Incomplete inv. graph

(d) Complete inv. graph at Statement S2.

(e) Complete inv. graph plus points-to info.

Figure 8.2: An example of extending the invocation graph and points-to analysis in the presence of function pointer.

program. More details can be found in [Ghi92].

### 8.2.3 Generalized Constant Propagation

Generalized constant propagation is an analysis that attempts to determine what range of values each variable can get at each point in a given program. Each time a variable is assigned a value, the value so given (if it is not user-input) is a function of existing variables. Thus, since most programs contain a liberal sprinkling of constants throughout, and provided one defines appropriate semantic functions for each possible operation, it should be possible to determine the set of the range of values each variable could contain at any point in its lifetime in any conceivable run of the program by performing a top-down semantic analysis of the program.

If one is to apply semantic functions to compute possible values, one of course needs to know the possible values of the variables serving as input, as accurately as possible. For example, if it is known that `a` could have any value in the range of [0,3] and `b` could have any value in the range of [1,2], then `a+b` can be any value in the range [1,5]. The range [1,5] is calculated from the above values using the semantic function corresponding to simple addition of integers. However, given the ubiquity of indirect referencing in popular languages such as C and Pascal, such information may not even be readily available, let alone accurate. In these cases, there are two options: (i) either one assumes the worst case assumption for each dereference variable (i.e., that each dereference variable could assume any value its type allows), or (ii) one makes use of points-to analysis to determine which variables might be the result of each dereference. In the former case, the quality of the information gathered via generalized constant propagation is substantially reduced, so the later method is naturally preferable. Points-to analysis, then, is of particular importance if generalized constant propagation is to be useful for analyzing any non-trivial programs in such pointer-based languages.

We show the effect of points-to analysis in two examples, one with points-to information and one without. In order to understand the examples, we define the following notations:

- GCP comments: represents the information available after the statement would have been executed.
- $\pm$ Inf : means $\pm$ infinity.
- var:[val1,val2] : means that 'var' can get the values between 'val1' and 'val2'.

The following represents a small program with the result of general constant propagation after each statement of the program. After processing statement `S1`, `count` gets the constant zero. The range of `ptr` is positive numbers, because `ptr` is a pointer variable and pointers can hold only addresses (which are always a positive number). Since variable `i` is not assigned to any value, it can be any value. If the points-to relationship is not available at statement `S3`, the worst-case assumption says that `*ptr` can be either `i` or `count`. Therefore, the constant 1 should be added in the range of the variables that they can get. This results in increasing the range of the variable `count` and leaving the value of `i` unchanged (because infinity plus one is still infinity).

```
/* WITHOUT points-to information */
   int main() {
        int i ;
        int  *ptr ;
        int count ;

S1:     count = 0 ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[-Inf,+Inf]} */

S2:     ptr = &i ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[-Inf,+Inf]} */

S3:     *ptr = 1 ;
        /* GCP = {count:[0,1],ptr:[0,+Inf],i:[-Inf,+Inf]} */
   }
```

The following shows the result of the same program while points-to information is available. At statement S3, the points-to relationship is (ptr,i,D), therefore *ptr is equivalent to i. The result of general constant propagation would be that count is constant zero and i is constant one.

```
/* WITH points-to information */
   int main() {
        int i ;
        int  *ptr ;
        int count ;

S1:     count = 0 ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[-Inf,+Inf]} */

S2:     ptr = &i ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[-Inf,+Inf]} */

S3:     *ptr = 1 ;
        /* GCP = {count:[0,0],ptr:[0,+Inf],i:[1,1]} */
   }
```

This simple example shows the strong affect of points-to information on general constant propagation.

| Benchmark | Exact const | Bounded ranges | Half ranges | Unbounded ranges | Range instances | Description |
|---|---|---|---|---|---|---|
| spanning | 74 | 81 | 0 | 26267 | 26422 | Minimum spanning |
| | 84 | 93 | 648 | 25597 | 26422 | circle calculation |
| 3sorts | 1 | 0 | 29 | 3765 | 3795 | 3 Sorts: Bubble, Quick, |
| | 2 | 3 | 204 | 3586 | 3795 | Quick (Median of 3) |
| determinant | 20 | 18 | 77 | 3375 | 3490 | Matrix determinant |
| | 57 | 113 | 159 | 3161 | 3490 | calculation |
| knight_tour | 19 | 57 | 50 | 2362 | 2488 | Recursive Knight's tour |
| | 30 | 248 | 0 | 2210 | 2488 | solution calculation |
| root_complex | 9 | 15 | 46 | 17548 | 17618 | Complex roots |
| | 59 | 159 | 215 | 17185 | 17618 | of polynomials |
| heap_sort | 34 | 68 | 20 | 6257 | 6379 | Heap sort |
| | 103 | 89 | 82 | 6105 | 6379 | |
| Mersenne_prime | 54 | 104 | 141 | 2906 | 3205 | Mersenne primes |
| | 61 | 240 | 157 | 2747 | 3205 | |
| 2sorts | 13 | 8 | 9 | 2457 | 2487 | 2 Sorts: Bubble, Quick |
| | 21 | 12 | 79 | 2375 | 2487 | |

Table 8.7: Statistics of general constant propagation in the absence (first line) and presence (second line) of points-to analysis.

Table 8.7 represents the statistical result of general constant propagation in the presence and absence of points-to analysis. This table contains the following information about the benchmark files:

1. Benchmark: is the name of the benchmark
2. Exact const: is exact constants, like [3,3]
3. Bounded ranges: both ends of the range are constant, like [0,10]
4. Half ranges: one end of the range is infinite, like [2,Inf]
5. Unbounded ranges: both ends of the range are infinite, like [-Inf,Inf]
6. Range instances (total of the above 4)
7. Description: is the description of the benchmark

Referring to Table 8.7, it is observed that the presence of points-to analysis has generally resulted in a noticable improvement. In specific, there is a large difference between the values of exact constant which is the most important constant information in a program.

### 8.2.4 Practical Array Dependence Analysis

This analysis determines whether dependences exist between *two subscripted references to the same array* in *a loop nest*. This is a two-step process. The first step is to set up a system of dependence equations and inequalities. In the second phase, a decision algorithm determines if the system has a solution [BC86]. The goal of the dependence testing is to disprove dependence of array subscripted pairs as many as possible and as early as possible. If dependence exists, it tries to find *distance and direction vectors*[1]. There are three cases where pointer alias information is used by array dependence analysis. These are listed in the following:

- To distinguish a good-loop[2] in the program. If a loop is not a good-loop, the dependence analysis does not continue.

  At statement `S1`, there is a relationship (a,b,D). Therefore the modification to `*a` is independent of loop index `i`, consequently, the following loop is a good-loop.

  ```
  main()
  {
  int  w[100] ;
  int  *a, b, i ;

     a = &b ;
     for ( i = 1; i <= 99 ; ++i )
      {
  S1:     *a = i * 2 ;
          w[i] = w[i+1];
      }
  }
  ```

  However, in the lack of points-to information, any loop with a modification to an indirect reference would not be considered as a good-loop.

- To replace, if necessary, any indirect reference variable with its corresponding points-to variable(s) along the backward analysis path. This replacement is used to increase the possibility of a successful *canonical transformation* which is a transformation that converts array subscript expressions into *canonical expressions*[3].

[1] *Distance and direction vectors* represent the access pattern between loop iterations.

[2] A *good-loop* is a loop-nest with no statements in between two loop statements and loop indices not modified in the loop body.

[3] A canonical expression is a subscript expression that can be expressed as a linear (affine) function of for-loop variables

In the lack of this information, one can not continue the backward analysis when an indirect reference variable is found. Therefore, the canonical form can not be constructed.

- To detect whether two pointer references point to the same array reference, or a pointer reference points to an array reference. This detection increases the number of array pairs to be analyzed by practical array dependence tester. In the following example, without points-to information, one could not detect that `s[i+1]` (from statement `S1`) and `a[i+1]` are the same location. Therefore, one should take the conservative assumption that says: `s` can refer to any array, consequently, `a` is dependent on all the arrays in its current scope (`b` and `c`).

```
main()
{
    int  a[100], b[50], c[7];
    int  i;
    int  *s;

    s = a ;
    for (i = 1 ; i < 100 ; i++)
S1:     a[i] = s[i+1];
}
```

Readers are refer to [Jus93] for more details on this analysis.

### 8.2.5 Reaching Definition Analysis

Reaching definitions have been formally defined in [ASU86] as: A *definition of a variable* $x$ is a statement that assigns, or may assign a value to $x$. A definition $d$ *reaches* a point $p$, if there is an execution path immediately following $d$ to $p$ such that $d$ is not 'killed' along that path. This is a forward analysis where a program is analyzed from top to bottom [ASU86].

In the case of pointers, points-to information is used to make reaching definition information as accurate as possible. Consider the following example:

```
S1:     *t = 3;
S2:  --->
```

When `S1` is reached, all the variables that `t` points-to are considered. If `t` definitely-points-to a variable, say `a`, it is concluded that the definition `S1` of `a` definitely-reaches

S2. If t possibly-points-to some variables, say a and b, then it is concluded that the definition S1 of both a and b possibly-reaches S2.

If the points-to information were not available, then all the variables of the current scope would be considered to possibly-reach to S2.

The general case of this analysis together with an example is given in Chapter 1. Readers are referred to [Sri92] for more details.

### 8.2.6 Live Variable Analysis

A variable $x$ is *live* at a point $p$ in the program if the value of $x$ at $p$ could be used along some execution path in the program starting at point $p$ [ASU86]. Thus, during live-variable analysis, at every point in the program, the set of variables that are live at that point in the program are computed. This is a backward analysis (the program is analyzed from bottom to top) and is absolutely essential for optimizations like register allocation.

For pointers, the points-to information is used to determine liveness accurately. Consider the following example:

```
                        <---   'a' is definitely-live
  S1:    ... = *t;
```

When the statement S1 is reached, all the points-to information of t are considered. If t definitely-points-to a variable, say a, then it is concluded that both t and a are definitely-live at S1. If t possibly-points-to some variables, say a and b, then it is concluded that a and b are possibly-live and t is definitely-live at S1.

If the points-to information were not available, then all the variables of the current scope would be considered to be possibly-live at S1. This increases the live-ranges of variables and, hence, makes the register allocation less efficient.

Another example of this analysis is given in Chapter 1. Readers are refer to [Sri92] for more details.

### 8.2.7 Loop Unrolling

Loop unrolling refers to the process of making one or more copies of the loop body, whereby the loop control must be suitably modified. Loop unrolling has many advantages including: increasing the number of instructions that can be scheduled in a basic block, decreasing the unnecessary data movement in a memory hierarchy architecture, reducing loop overheads, etc.

One of the conditions for loop unrolling is that the loop index should not be modified inside the body of the loop. Without points-to information, it is difficult to know whether any pointer (whose dereference is being modified in the body of the loop) is indirectly pointing to a loop index or not. For example, consider the following program:

```
    main()
    {
        int a[100], b[100] ;
        int *x, y ;
        x = &y ;
        ...
        for ( i = 1 ; i < 100 ; i = i + 1 ) {
            a[i] = b[i-1]
S1:         *x = *x + i ;
        }
    }
```

Without the points-to information, it is not safe to unroll the loop, because `*x` could be `i`. With points-to information, it is trivial to notice that at statement `S1` the points-to relationship is (x,y,D), and the loop can be unrolled as shown below:

```
main()
{
    int a[100], b[100] ;
    int *x, y ;
    x = &y ;
    ...
    for ( i = 1 ; i < 99 ; i = i + 2 ) {
        a[i] = b[i-1] ;
        *x = *x + i ;
        a[i+1] = b[i] ;
        *x = *x + i + 1 ;
    }
}
```