Analysis of Loops

Florian Martin Martin Alt Reinhard Wilhelm Christian Ferdinand

Universität des Saarlandes, Postfach 151150, 66041 Saarbrücken
Fon: +49-681-302-5571, Fax: +49-681-302-3065
{florian|alt|ferdi|wilhelm}@cs.uni-sb.de

Programs spend most of their time in loops and procedures. Therefore, most program transformations and the necessary static analyses deal with these. It has been long recognized, that different execution contexts for procedures may induce different execution properties. There are well established techniques for interprocedural analysis like the call string approach. Loops have not received similar attention in the area of data flow analysis and abstract interpretation. All executions are treated in the same way, although typically the first and later executions may exhibit very different properties.

In this paper a new technique is presented that allows the application of the well known and established interprocedural analysis theory to loops. It turns out that the call string approach has limited flexibility in its possibilities to group several calling contexts together for the analysis. An extension to overcome this problem is presented that relies on a similar approach but gives more useful results in practice.

The classical and the new techniques are implemented in our Program Analyzer Generator PAG, which is used to demonstrate our findings by applying the techniques to several real world programs.
Keywords: program analysis, program analyzer generator, loops, call string approach, functional approach.

1 Introduction

In data flow analysis the meet over all paths solution is computed or approximated. In the presence of loops this means that for the body of a loop the data flow value on the first entry is combined with the values upon return. Loops may start in a state very different from that encountered in further iterations, so that it could be useful to keep them distinguished in a data flow analysis.

To allow for this a solution is presented which relies on extensions of well known interprocedural analysis techniques like the *call string approach* or the functional approach described by Sharir and Pnueli [8]. This has the advantage that similar problems -loops and procedures- can be treated in the same formal framework. Furthermore this allows to use the existing theory and implementations of the interprocedural analysis for the analysis of loops.

The necessity for better loop analyses has also been claimed by Harrison [5], who proposed to transform loops into procedures to use the techniques for interprocedural analyses. Also in the area of compiler construction there are several optimization techniques for loops (e.g. software pipelining).

The main idea of applying the interprocedural analysis to loops is to extend the procedure concept to a special block structure in the control flow graph for a program. Such blocks can be analyzed like procedures in interprocedural analysis, thus allowing for a separation of the information for different paths through the control flow graph.

As the number of paths is usually infinite, it is not possible to analyze all of them separately. Therefore one has to partition the sets of paths into classes, in order to analyze each of them at once. In the functional approach, this is done by inspecting the data flow information at the entries of blocks dynamically at the analysis time. In the call string approach, the paths are grouped statically in advance.

The solution presented here allows not only for the application of the interprocedural techniques to loops but to arbitrary blocks of code. This paper shows that the classical call string approach for the interprocedural analysis is not always well suited for the analysis of nested loops. We present a new improved technique called VIVU.

The classical and the new approaches are integrated into the program analyzer generator PAG [2]. Practical experiments with several real applications show the applicability and the performance gains of the new approach with respect to the classical approaches.

In the next section the motivation for the extension of the interprocedural analysis techniques is given by taking a closer look at the analysis of loops in the context of a practical application. In Sec. 3 two classical interprocedural techniques are discussed. The VIVU approach is introduced in Sec. 4. In Sec. 5 the classical interprocedural techniques and VIVU are applied to an analysis problem, and the results are compared and evaluated.

2 Motivation

As an example the analysis of loops is considered in the context of cache behavior prediction [1, 4].

Caches are used to improve the access times of fast microprocessors to relatively slow main memories. They are an upper part of the storage system hierarchy and fit in between the register set and the main memory. They can reduce the number of cycles a processor is waiting for data by providing faster access to recently referenced regions of memory. Most modern workstations are equipped with microprocessors that have cycle times of about 2 to 40ns and DRAM (Dynamic Random Access Memory) that has a cycle time of 90ns and more [6].

Cache behavior prediction is a representative of a large class of analysis problems that are of high practical relevance in the area of hard real time systems. These require a timing validation based on bounds of the execution time. Closely related is pipeline behavior prediction for which similar analysis requirements exist.

The goal of cache analysis is to compute a categorization for each memory reference that describes its cache behavior. The categories are described in Fig. 1.

Category	Abb.	Meaning
always hit	ah	The memory reference will always result in a cache hit.
always miss	am	The memory reference will always result in a cache miss.
not classified	nc	The memory reference could neither be classified as ah nor am.

Fig. 1. Categorizations of memory references

Example 1.
Let us consider a sufficiently large data cache and the following program fragment: In the first execution of the loop, the reference to `v` will result in a cache

```
/* Variable v not in the data cache */
for i:=1 to .. do
    ⋮
    y:=v
    ⋮
end
```

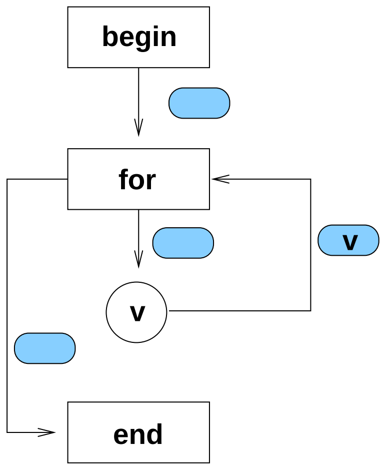

Fig. 2. Motivating example 1

miss, because `v` is not in the cache. In all further iterations the reference to `v` will result in a cache hit, if the cache is sufficiently large to hold all variables referenced within the loop.

The control flow graph for this program is shown in Fig. 2. An empty box means that `v` is not in the cache, and a box with `v` means that `v` is in the cache. In the classical approach, the first iteration and all further iterations are not distinguished. The combination function which is needed to combine data flow information for nodes in the control flow graph with several incoming edges is for the cache analysis similar to set intersection. In the example the combination function is used to combine the data flow information at the entry of the loop with the data flow information from the end of the loop body to obtain a new data flow value at the beginning of the loop. The combined data flow information can not include the variable `v`, because `v` is not in the cache when the loop is entered. The reference to the variable `v` will be classified as nc. For a WCET (Worst Case Execution Time) computation this is a safe approximation, but nevertheless not a good one.

3 Interprocedural Analysis

The key idea of interprocedural analysis is that in general a procedure called more than once will have different data flow elements calculated for these different *dynamic calls*. The different data flow values for the calls of a procedure are called *calling contexts*. The most precise analysis results are obtained if the procedure is analyzed separately for each calling context. But this does not only increase the complexity of the analysis, but may even lead to nontermination of the analysis, if it can't be guaranteed that the procedure is analyzed only with a finite number of different calling contexts. The strategies to overcome this problem and to reduce the complexity of the analysis differ in the two following classical approaches [8].

If for a procedure different calling contexts are known this can be used in a subsequent optimization pass to create several optimized versions of the procedure which are specialized according to different contexts. If it is statically known from the analysis which call contexts are produced by a call site then this call can be replaced by a call to the version of the procedure specialized according to these contexts.

3.1 Functional approach

The functional approach doesn't try to reduce the complexity of the analysis. It analyzes each procedure once for every call context that arises during the computation. This can be done by tabulating the different call contexts for each procedure with the corresponding data flow element for the exit of the procedure. So the tables for the procedures can be seen as functions which map incoming data flow values to outgoing data flow values. They represent abstract versions of the procedures.

Each time the iteration algorithm reaches a call node it looks up the call context in the table of the called procedure. If an exit value is found, this is used as the result of the call. If no exit value is found its calculation is triggered. Since procedures can be (simultaneously) recursive this can trigger the calculation of other values.

The tables for every procedure can't grow infinitely if the abstract domain is finite. No other good sufficient termination conditions are known to us, if we assume that there exists no compact representation for the abstract functions. The disadvantage of the functional approach is that at the end of the analysis the correspondence between paths and table entries is not known, so an efficient specialization is not possible.

3.2 Call string approach

In the call string approach a partition of the dynamic calls for each procedure is constructed in a pre pass of the analysis. During the analysis each procedure is analyzed once for each class of dynamic calls.

The members of a class of dynamic calls for a procedure are chosen according to the suffixes of the path they have taken through the dynamic call tree. (A *call string* is a concatenation of calls c.) The hope is that only those dynamic calls are in the same set which have a similar data flow behavior.

The idea can be understood as simulating the call stack of an abstract machine which contains frames for each procedure call that has not yet finished.

The tracing of the call strings is done by encoding them into the domain of data flow values: if the original domain is D then a domain $D^* : \Gamma^K \rightarrow D$ is constructed, where Γ^K is the set of all call strings with length at most K. Then the transfer functions $tf : D \rightarrow D$ are extended appropriately to a function $tf^* : D^* \rightarrow D^*$.

This approach has three advantages compared to the functional approach: first it is possible to deal with data flow domains of infinite cardinality. Second, it is easily possible to cut down the complexity of the analysis by selecting small values for K. And third, it is easily possible to find for each call context the set of paths through the call graph and therefore to replace a procedure by specialized versions.

The disadvantages are: the call string approach can be less precise than the functional approach. And it is not very practical to encode the call strings into the analysis domain.

4 Extending Interprocedural Analysis

In the interprocedural analysis pieces of code (procedures) are analyzed several times for the different incoming data flow values (calling contexts). This is done to get better analysis results for pieces of code that are executed in different contexts. As the motivating example of Sec. 2 has shown also the body of loops are executed several times with different contexts. Therefore the techniques for the interprocedural analysis are now extended such that they can be applied also to loops.

To do so the concept of procedures is generalized to *blocks*. Blocks have entry and exit nodes, which are the only ways to enter or leave the block. Additionally, there can be edges from inside the block to the entry of other blocks (which correspond to procedure calls in the interprocedural context and are therefore referred to as *call edges*). For each of these call edges there has to be a corresponding edge from the exit of the called block back to a node inside the calling

```
x = 0;
while (x<10) {
   x += 1;
}
```

a)

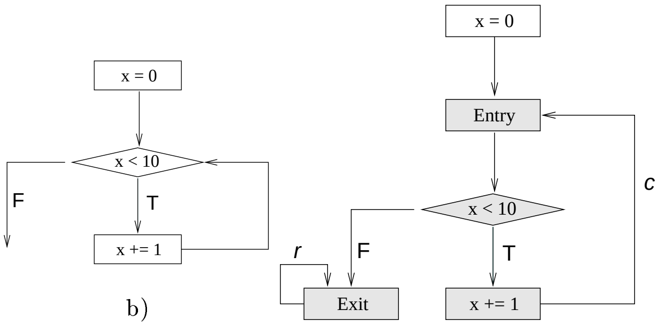

Fig. 3. A loop, the original CFG and the transformed CFG

```
x = 0;
loop();


void loop(void) {
   if (x<10) {
   x += 1;
   loop();
   }
}
```

a)

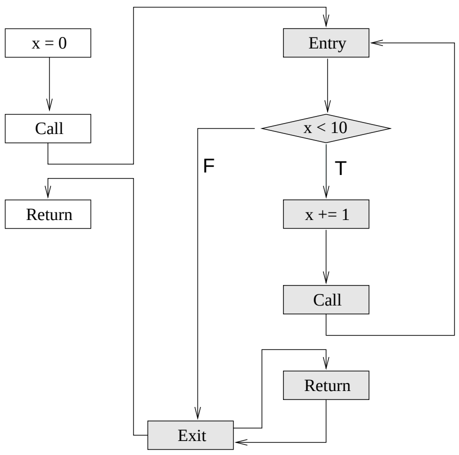

Fig. 4. The transformation from a loop to a recursive procedure

block (which will be called *return edges*). Each node in the CFG belongs exactly to one block.

The transformation of a loop to a block can be seen in Fig. 3. The loop in a) corresponds to the CFG in b) which is transformed to the CFG in c). The description above requires the edge r as correspondence to the edge c in the transformed CFG. This edge r in Fig. 3 c) allows the continuation of the calling block after the called block returns. But as a loop is "tail recursive" there is nothing to do after the return. Therefore the return edge degenerates to a self loop at the exit node. For comparison, Fig. 4 shows the code and CFG of the loop in Fig. 3 expressed as a recursive procedure. But although self loops can

be omitted for a standard data flow analysis, those edges are important for the transformation made in Sec. 4.1.
A program with the described block structure and calling conventions is represented by a supergraph.

Definition 1 Supergraph.
A supergraph $G^* = (G, P)$ consists of a control flow graph $G = (N, E)$ with a set of nodes N and a set of edges E and a partition $P \subset 2^N$ of nodes, where each class P_i represents a *block*. Each block has unique entry and exit nodes $(\mathsf{entry}_i, \mathsf{exit}_i)$. The class P_0 represents the main block and therefore entry_0 has no predecessors and exit_0 no successors. Each edge from a node $n_1 \in P_i$ to entry_j has a corresponding edge from exit_j to an $n_2 \in P_i$. All other edges are intra partition edges.

A block in the supergraph is said to be *(directly) recursive* if it has an edge to its own entry. A set of blocks in the supergraph is *simultaneously recursive* if their subgraphs are strongly connected. A supergraph is called *recursive* if it contains (a set of simultaneously) recursive blocks. In this paper only procedures and loops are used as blocks but the approach is not limited to them. It can be applied to each block of code with more than one entry.

In the remainder of this section the static call graph approach from [2] is introduced and extended to blocks. It transforms a supergraph and a call string length K to an expanded supergraph. This expanded supergraph is a CFG on which the analysis problem can be solved by a standard intraprocedural algorithm. The static call graph approach can also be used for other interprocedural techniques than the call string approach.

4.1 Static call graph approach

To keep different call paths separated, each node in the supergraph is annotated with an array of data flow elements. A pair consisting of a node and an index to the array is called a *location*. For every block P_i, the number of data flow elements at nodes of this block is fixed and called its *multiplicity* written as $\mathsf{mult}(P_i)$. Moreover $\mathsf{mult}(P_0) = 1$, i.e. there is only one location in the main block. mult can be simply extended to nodes by defining: $\mathsf{mult} : N \rightarrow \mathbb{N}$ by $\mathsf{mult}(n) := \mathsf{mult}(P_i)$, iff n is a node in P_i.

A *connector* is a set of functions $\mathsf{con}_\mathsf{c} : \{1, \ldots, \mathsf{mult}(P_i)\} \rightarrow \{1, \ldots, \mathsf{mult}(P_j)\}$ that describe for each call edge c how the locations of the calling block are connected to the locations of the called block. Different strategies like the call string strategy to construct connectors and multiplicities for a supergraph can be chosen. They will be discussed in detail in the next section.

This leads to the definition of the expanded supergraph, a new kind of CFG, whose nodes are locations. The locations are connected along the edges of the supergraph. Within blocks the i-th location is connected with the i-th location of the successors. For call edges c con_c determines how the data flow elements of the calling block are connected to the elements of the called block. At the corresponding return edges the inverse of con_c is applied.

Definition 2. The *expanded supergraph* for a supergraph $G^* = ((N, E), P)$, a function $\mathsf{mult} : N \to \mathbb{N}$ and a connector con_C is defined as

$G_E = (N^*, E^*)$, with

- $N^* = \{(n, i) | n \in N, i \in \{1, \ldots, \mathsf{mult}(n)\}\}$
- $((n_1, i_1), (n_2, i_2)) \in E^*$, iff $e = (n_1, n_2) \in E$ and one of the following conditions hold:
  - i.) $n_2 = \texttt{Entry}\ P_j$ and $i_2 = \mathsf{con}_e(i_1)$
  - ii.) $n_1 = \texttt{Exit}\ P_j$ and $i_1 = \mathsf{con}_\mathsf{C}(i_2)$, where c is the corresponding call edge to e.
  - iii.) $i_1 = i_2$ and $n_1 \neq \texttt{Exit}\ P$ and $n_2 \neq \texttt{Entry}\ P$

Figure 5 is an example for an expanded supergraph with $\mathsf{mult}(P_1) = 1, \mathsf{mult}(P_2) = 2$, $\mathsf{con}_{c_1}(1) = 1$, $\mathsf{con}_{c_2}(1) = 2$, and $\mathsf{con}_{c_2}(2) = 2$.

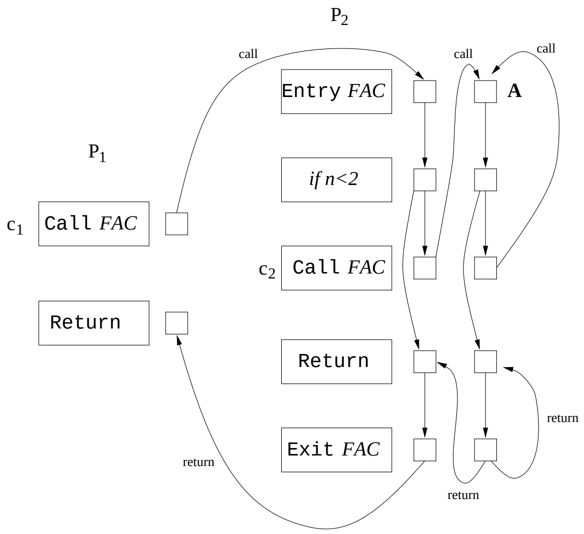

Fig. 5. Connector example

To solve a data flow problem with expanded graphs, it is necessary to find suitable con_C and mult functions. With these it is possible to tune the analysis: the higher the multiplicity the better the precision that can be achieved, but the more time and space are needed.

4.2 Connectors

In this section some methods to determine pairs of multiplicity and connectors are explained.

Simple connector In the simplest case, the multiplicity of each block is one, and the $\mathsf{con_c}$ functions are the identity. So the standard control flow graph is obtained.

Call string connector The call string approach for a fixed length K of the call strings can be simulated with the static call graph approach. This method allows to avoid the calculation of the call strings during the solution procedure by precalculating the paths through the call graph. Also the encoding of the call string in the analysis domain is avoided. This is done by the following static calculation: whenever the analysis passes a call edge (n, entry_i) the call string approach would append the node n to the actual call string. This can be simulated by encoding the call strings as numbers that correspond to locations: every call edge c is assigned a unique number between one and $M-1$, where M is the number of call edges in the given program. Then a call string $\gamma = c_{i_1} \ldots c_{i_K}$ corresponds to a K digit number m to base M ($m = i_1 \ldots i_K$). This number can be converted to the decimal system by multiplying the n-th digit by M^n and add this for all digits ($m' = \sum_{1 \leq j \leq n} i_j * M^j$). So m is between zero and M^K where zero denotes the empty call string ϵ. Fortunately not all of these call strings are valid for each procedure. By deleting all non valid numbers for a procedure the number of locations can be reduced, so that every location is used.

Call string 1 connector The call string approach for $K = 1$ can also be described as follows: count the number of incoming edges to entry_P to take this as the multiplicity of P . Then the $\mathsf{con}_{(n,\mathsf{entry}_P)}$ functions project all data flow elements of n to a single fixed position in the vector of data flow elements of entry_P.

4.3 VIVU

It has turned out in practice that the call string approach is not optimal for nested loops. Even if the length of the call string is chosen to be the level of the nesting depth of the loops many paths are separated which are not interesting.

Figure 6 shows an example for two nested loops. The call edges are labeled with f_1, f_2, o_1, o_2 where f stands for the first calls and o for other calls. Possible call strings which reach loop2 are $f_1(f_2 o_2^* o_1)^* f_2 o_2^*$.

The call string approach with $K = 2$ considers all suffixes of length two of the paths. These (and their interpretation) are:

- $f_1 f_2$ (first iteration of the outer and inner loop)
- $f_2 o_2$ (second iteration of the inner loop)
- $o_2 o_2$ (iteration ≥ 3 of the inner loop)
- $o_1 f_2$ (iteration ≥ 2 of the outer loop first of the inner)

If an analysis problem is assumed for which initialization effects can be expected then it is not important to separate the second and all other iterations, but the

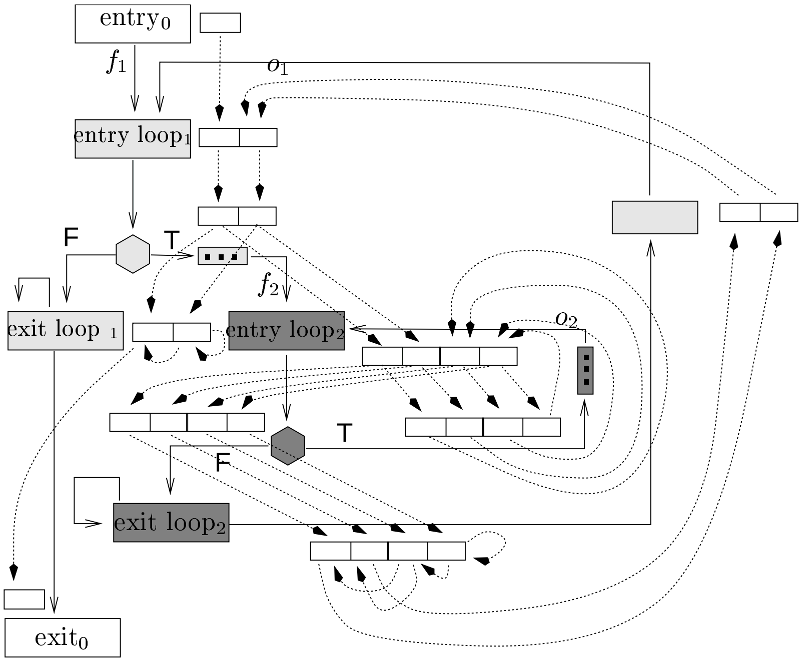

Fig. 6. Call string approach for two nested loops

first from all other iterations. This leads to the following more appropriate separation scheme:

outer loop	inner loop
first	first
first	other
other	first
other	other

For the two nested loops this results in the expanded supergraph in Fig. 7.

For programs with direct recursion the formal mapping and multiplicity construction looks as follows: the multiplicity of a block P_i is the sum of all blocks P_j except P_i that have an edge to the entry of P_i. If P_i is recursive this has to be multiplied by two. For a recursive edge from P_i to P_i the con_c function is defined as

$$\mathsf{con}_c(x) = \begin{cases} \mathsf{mult}(P_i)/2 + x & \text{if } x \leq \mathsf{mult}(P+i)/2 \\ x & \text{otherwise} \end{cases}$$

For the non recursive edges all non recursive edges to the entry of P_i have to

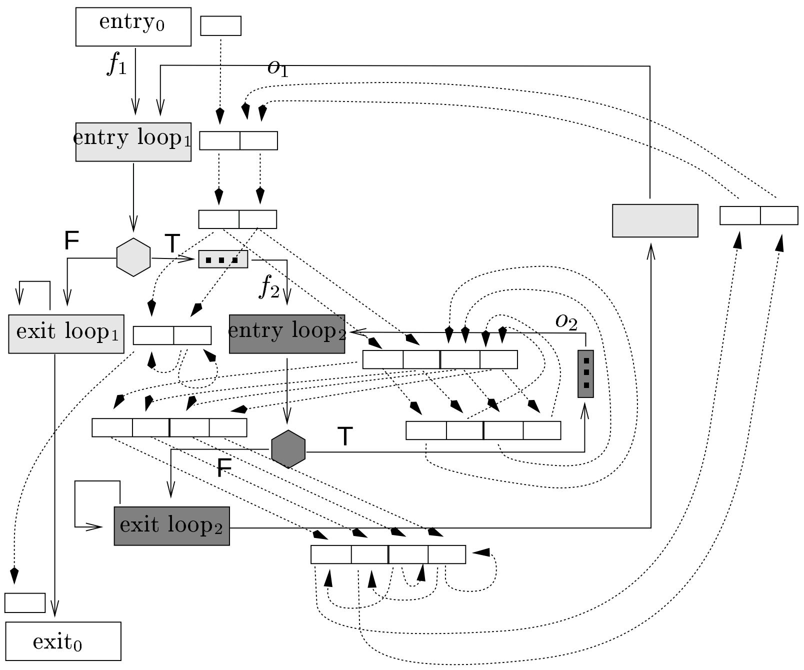

Fig. 7. VIVU for two nested loops

be numbered from 1 to n. Then for the k-th edge c_k

$$\mathsf{con}_{c_k}(x) = x + \sum_{j=1}^{k-1} \mathsf{mult}(P_j)$$

if P_j is the source block of the j-th call.

For non recursive procedures this method simulates full inlining, and for recursive ones it separates the first pass through it from all other passes through it. This is where the name for this mapping comes from: Virtual Inlining (of procedures) and Virtual Unrolling (of loops).

For supergraphs with simultaneously recursive blocks the complete "virtual inlining" has to be given up as follows: first a "call graph" G has to be constructed with the blocks as nodes and an edge between P_i and P_j iff $\exists(n, \mathsf{entry}_j) \in E : n \in P_i$. In this graph the strongly connected components have to be calculated and considered as a single node in a collapsed call graph G'. To this acyclic call graph G' the method described above for the simple recursive case can be applied. The collapsed call graph G' has to be expanded again. In this expansion process every block in strongly connected component inherits the multiplicity from the summary node. An example is shown in Fig. 8.

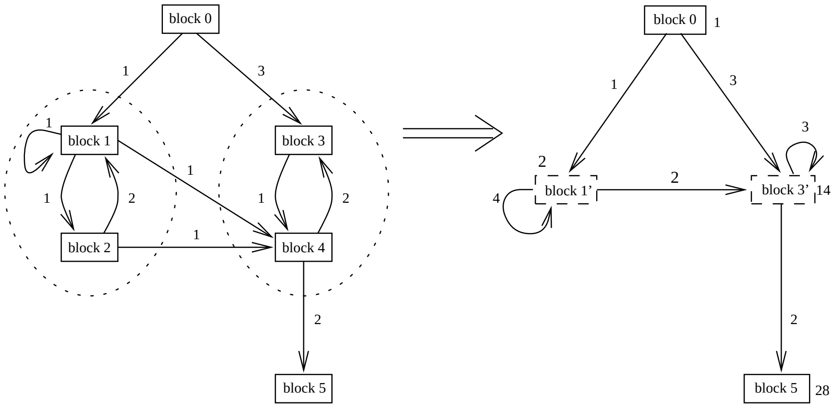

Fig. 8. The calculation of the multiplicity for simultaneous recursive supergraphs. The numbers on the edges mean the numbers of edges to the entry, and the numbers at the nodes are the calculated multiplicities, e.g. the edge `block 2` to `block 1` means that there are two edges from nodes in `block 1` to the entry of `block 2`.

Name	Description	# Instr.
`fdct`	JPEG forward discrete cosine transform	370
`matmult`	50x50 matrix multiplication	154
`ndes`	data encryption	471
`djpeg`	JPEG decompression (128x96 color image)	1760
`fft`	fast Fourier transformation	1810
`matsum`	100x100 matrix summation	135
`stats`	two arrays sum, mean, variance, standard deviation, & linear correlation	456
`dhrystone`	Dhrystone integer benchmark	447

Fig. 9. Test programs

5 Practical Evaluation

The VIVU connector has been applied to the cache behavior prediction analysis for all loops and functions and has shown very good results. Figure 9 shows some of the test programs we used. These are all executables in the `a.out` format. The number of machine instructions is shown in the last column. In Fig. 10 one can see the percentage of instructions for each program that has been classified as hits/misses or neither of them both for the traditional analysis and for the VIVU connector. For most programs there is a definite precision gain of VIVU over the traditional method, and for some programs VIVU allows to predict the cache behavior precisely. Of course VIVU can increase the number of analyzed contexts but the observation from our experiments was that this increase is quite moderate. Indeed all the test programs could be analyzed within a few seconds on a Sun SPARCstation 20. The analysis times are shown in seconds as well for

Name	Traditional				VIVU				Advantage
	ah	am	nc	time	ah	am	nc	time	
`fdct`	74.86%	2.43%	22.70%	0.04	86.90%	13.10%	0.00%	0.03	22.70%
`matmult`	80.20%	12.87%	6.93%	0.05	89.70%	10.30%	0.00%	0.05	6.93%
`ndes`	77.72%	3.25%	19.02%	0.39	89.87%	7.65%	2.49%	0.20	16.53%
`djpeg`	73.51%	2.03%	24.46%	2.46	88.86%	0.60%	10.54%	24.38	13.92%
`fft`	71.71%	3.23%	25.07%	3.30	75.86%	1.94%	22.21%	7.72	2.86%
`matsum`	76.97%	16.45%	6.58%	0.03	85.83%	14.17%	0.00%	0.03	6.58%
`stats`	73.50%	7.69%	18.80%	0.37	79.12%	8.17%	12.71%	0.26	6.09%
`dhrystone`	70.99%	6.09%	22.92%	0.21	80.44%	10.94%	8.62%	0.02	14.30%

Fig. 10. Evaluation of VIVU for cache analysis for a superSPARC instruction cache. The time is the analysis time in seconds.

the traditional approach as for VIVU analyses.

By applying the VIVU connector to the example of Sec. 2 one can see in Fig. 11 that clearly for the first iteration the reference to `v` will be a miss and that for all other iterations it will be a hit.

6 Related work

The technique proposed here to analyze loops should not be confused with several code motion techniques to move loop invariant code out of loops. These are special data flow analyses which fit in the classical data flow framework, whereas the proposed technique is a general framework which can be applied to all data flow analyses in order to obtain more precise analysis results.

Structure based analyses like interval analysis [7] are orthogonal to the technique presented here. They are used to solve data flow problems efficiently in the presence of loops. They can be used to solve a VIVU problem on the expanded supergraph instead of iterative algorithms.

In [9] a property oriented expansion of a program model was developed. This aims toward similar goals as the analysis presented in this paper: Separation of program states at a program point that have different properties and are generated through different program paths. In [9] this is reached by unfolding all paths that result in different properties. To terminate this method requires the set of all properties to be finite. But even then the worst case complexity is worse than the one of the functional approach, since the expansion is not limited to certain call edges but is applied to each node which has more than one predecessor.

The fundamental work for interprocedural analysis was presented in [8] which discusses the functional and the call string approach in a theoretical way. A practical comparison can be found in [3].

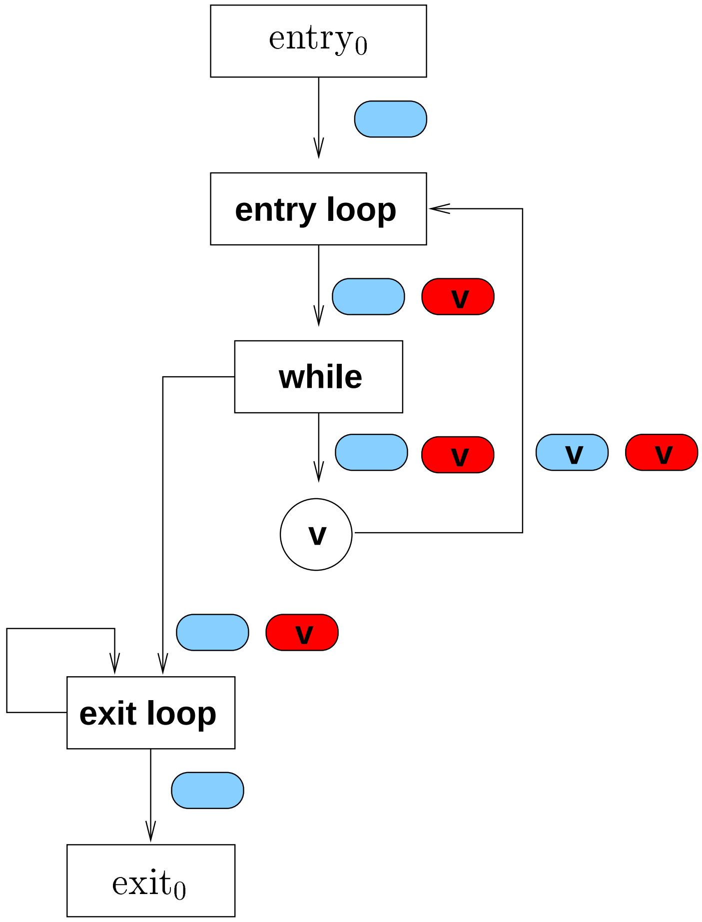

Fig. 11. Example for cache prediction

7 Conclusion

Motivated by poor results of several analyses in the practice we have presented a generalization of the interprocedural analysis for loops and arbitrary blocks.

By extending the existing methods through the static call graph technique it is possible to focus the analysis effort to the points of main interest. Especially for loops it allows to distinguish the first from all other iterations.

The applicability of our methods has been shown by our practical experiments. The newly developed VIVU approach makes it possible to predict for example the cache behavior of programs within much tighter bounds than the conventional analysis methods.

References

1. Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In Radhia Cousot and David A. Schmidt, editors, *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer, September 1996. Long version accepted for SAS'96 special issue of Science of Computer Programming.
2. Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In Alan Mycroft, editor, *SAS'95, Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 33–50. Springer, September 1995.
3. Martin Alt and Florian Martin. Practical comparision of call string and functional approach in data flow analysis. In Herbert Kuchen, editor, *Arbeitstagung Programmiersprachen*, volume 58 of *Arbeitsberichte des Institutes für Wirtschaftsinformatik*. Westfälische Wilhelms-Universität Münster, July 1997.
4. Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 37–46, June 1997.
5. Williams Ludwell Harrison III. *Personal communication on Abstract Interpretation, Dagstuhl Seminar*, 1995.
6. J.L Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.
7. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–315, September 1986.
8. Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
9. Bernhard Steffen. Property-oriented expansion. In Radhia Cousot and David A. Schmidt, editors, *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 1996.
10. Stephan Thesing, Florian Martin, and Martin Alt. *PAG User's Manual*, 1997.
11. Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. International Computer Science Series. Addison–Wesley, 1995.