Real-World Loops Are Easy to Predict

Raphael Ernani Rodrigues

Department of Computer Science, UFMG, Brazil
raphael@dcc.ufmg.br

Abstract. The Trip Count of a loop determines how many iterations this loop performs. Several compiler optimizations yield greater benefits for large trip counts, and are either innocuous or detrimental for small ones. However, predicting exactly the trip count of a loop is an undecidable problem in general. Thus, such problem is usually approached through heuristics, which tend to be computationally expensive. In this paper we argue that in most cases there is no need to resort to expensive methods and that in many cases the trip count prediction does not need to be sound. In that sense, we propose a lightweight trip count prediction heuristic. Our method identifies the pattern on which the induction variables of each loop are updated between two iterations and generate symbolic expressions that represent the trip counts of the loops. Such expressions can be evaluated at runtime with O(1) complexity and allow blocks of code to be conditionally executed, depending on the expected trip count. We argue that such technique is useful for speculative optimizations, very common in the world of just-in-time compilers. For instance, if we predict that a loop will iterate for a long time, we can perform more aggressive JIT optimizations. Furthermore, we show that despite the simplicity of our technique, we have accurately predicted nearly 90% of all the interval loops found in millions of lines of C code. The interval loops represent approximately 67% of the total number of loops of the programs.

1 Introduction

Loops represent most of the execution time of a program. For that reason, there is a well-known aphorism that says that "all the gold lays in the loops". Thus, compiler optimizations made inside loops have their benefits multiplied by the number of iterations actually executed. As a consequence of that fact, there is a vast number of works in the literature that are specialized in loop optimizations [19,11,14].

Some optimizations, however, are highly sensitive to the number of iterations of a given loop. For instance, if a given loop iterates a few times in an interpreter, an aggressive optimization made by a Just-In-Time (JIT) compiler may not even pay for the compilation overhead. On other hand, if the same loop iterates thousands of times, the JIT compilation might use more expensive techniques and still have a better end-to-end performance. The number of iterations a loop actually executes is called *Trip Count*. Here we use the same concept of trip count as described by Wolfe *et al.* [19, pp.200].

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 124–138, 2014.

However, in most cases this number is only known at runtime. Rice [16] has demonstrated that statically predicting this information is an undecidable problem. Therefore, this problem can only be partially solved by heuristics. There is a large number of works in the literature that propose different techniques to solve this problem [18,1,9]. The main disadvantage of them is the high computational cost to predict loops. That characteristic makes these solutions impractical to be applied to JIT compilers, for instance.

In this particular work, we argue that, although predicting exactly the trip count of a loop is undecidable, most of the time there is no need to use computationally expensive state-of-the-art methods to compute (an approximation of) it. We propose a heuristic that extracts patterns of the updates of variables' values and estimates the trip count of loops with symbolic expressions. Those expressions might, then, be evaluated at runtime with O(1) complexity. They allow the program to decide dynamically which piece of code to execute, depending on the actual expected number of iterations.

We support our position with a series of experiments. We have analyzed millions of lines of C code, with thousands of different loops present on well-known benchmarks. Our experiments demonstrate that most of the loops are easy to predict and do not demand expensive techniques to be accurately analyzed. We have collected statistics about the structure of thousands of loops that support our claim. According to our research, 70% of the loops have a very simple and well-behaved structure. We have instrumented those loops and we have exactly predicted the trip counts of 90% of them.

The rest of this paper is organized as follows: Section 2 gives us the basis upon which we develop our technique. Sections 3 and 4 describe our algorithms and explain our engineering choices. We experimentally evaluate the performance of our work in Section 5. In Section 6 we discuss how our work is related with existing efforts in the literature. Finally, in Section 7 we discuss possible future directions of this research and make final remarks.

2 Background

Our analysis combines information contained in the Control Flow Graph (CFG) and in the Data Dependence Graph of the program. From the CFG we can extract information about the structure of the analyzed program, like the points of the program where the loops start and stop, and which variables and instructions directly affect the control flow. From the dependence graph we can extract information about the way that the information flows among the variables. Those information allow us to generate symbolic expressions that estimate the trip count of the loops of the program.

The dependence graph [5] is defined in the following way: for each program variable v, we create a node n_v, and for each instruction i in the program we create a node n_i. For each instruction $i : \ v \ = \ f(\ldots, \ u, \ \ldots)$ that defines a variable v and uses a variable u we create two edges: $n_u \rightarrow n_i$ and $n_i \rightarrow n_v$.

Many variables of a program do not affect the predicates that represent the loops' stop conditions. Thus, we do not consider those variables in our analysis,

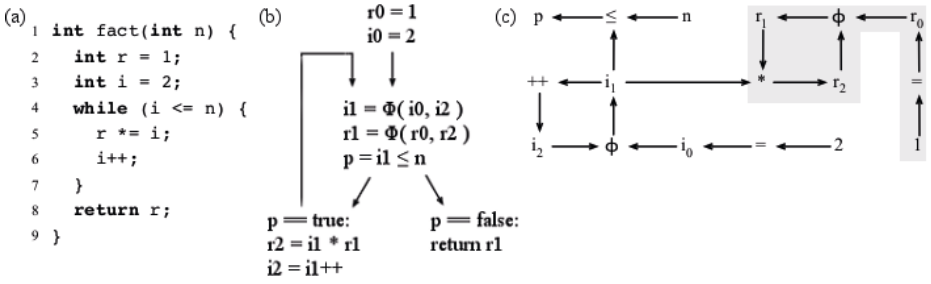

Fig. 1. (a)Example program. (b) CFG of the program, after conversion to SSA form. (c)Dependence graph highlighting nodes that do not affect the loop predicate, after converting the original program into SSA.

because they do not have any impact on the number of iterations of those loops. Therefore, despite of working with a slice of the program that eliminates those instructions, the result of our analysis remains the same. Figure 1 shows a dependence graph for the factorial function and highlights the variables that we can prune before doing our analysis. In this work we use the SSA representation form [3]. We chose this representation form because it is able to give extra precision to our analyses in cases when the same variable is redefined in two different loops.

Natural Loops

According to Appel and Palsberg [2, p.376], a natural loop is a set of nodes S of the control flow graph (CFG) of a program, including a header node H, with the following properties:

- from any node in S there is a path that reaches H;
- there is a path from H to any node that belongs to S;
- any path from a node outside S to a node inside S contains H.

A node PH of the CFG is a pre-header of a natural loop if and only if PH has H as an immediate successor. In this work we normalize the CFG in such a way that every natural loop has one unique pre-header PH, that is executed immediately before the first iteration of the loop. Such normalization gives us a basic block that immediately dominates the loop. This basic block is used by our profiler to initialize the variables that we use to observe the behavior of the loop.

In addition, the *stop condition* of a loop is a boolean expression $E = f(e_1, e_2, \ldots, e_n)$, where each $e_j, 1 \leq j \leq n$ is a value that contributes to the computation of E. Depending on the stop condition we classify the loop into one of the following categories:

- **Interval Loops** - The *stop condition* is an integer comparison instruction that receives two operands e_1 and e_2 and compares them with an inequality ($<$, $\leq$, $>$, or $\geq$).

- **Equality Loops** - The *stop condition* is an integer comparison instruction that receives two operands e_1 and e_2 and compares them with an equality (== or ! =).
- **Other Loops** - Any natural loop that neither is an *Interval Loop* nor is an *Equality Loop*.

Strongly Connected Components
Variables that are redefined during the execution of a loop of the program belong to cycles in the dependence graph. Cycles of redefinitions of variables can be identified by computing the Strongly Connected Components (SCCs) on the dependence graph [17]. The SCCs help us to group the different nodes of the graph that belong to the same redefinition sequence.

After we have computed the SCCs of the dependence graph, we can classify them in the following way:

- **Single-node SCC** - SCCs composed by only one node.
- **Multi-node SCC** - SCCs composed by more than one node. SCCs of this class represent cycles in the dependence graph.

Multi-node SCCs can be divided in two categories:

- **Single-path SCC** - From any node in the SCC there is only one path that starts and ends in the same node and passes through the edges of the SCC at most once.
- **Multi-path SCC** - There is at least one node in the SCC for which there are two or more paths that start and end in the same node and pass through the edges of the SCC at most once.

Single-path SCCs are unconditional sequences of redefinitions of a variable. This pattern of SCC is the easiest to analyze, because it is possible to infer statically what is the effect of one iteration of the loop on the variables of the SCC.

Multi-path SCCs are conditional sequences of redefinitions of a variable. This means that there are conditional branches inside the CFG loop. The amount of branches makes the total number of possible paths to grow exponentially. Thus, this class of SCCs is harder to analyze. Multi-path SCCs can be further classified into two different categories:

- **Single-loop SCC** - The SCC has branches that does not constitute nested loops.
- **Nested-loop SCC** - There are inner loops inside the SCC. In order to avoid non-termination problems, we do not analyze this category of SCC.

Sequences of Redefinitions of Variables
A sequence of redefinitions (SR) is a path in the SCC that starts and ends in the same node and does not repeat any edge. By construction, our dependence graph does not admit self loops, so SRs are always extracted from Multi-node SCCs. A SR can be interpreted to generate the effect of one iteration of the

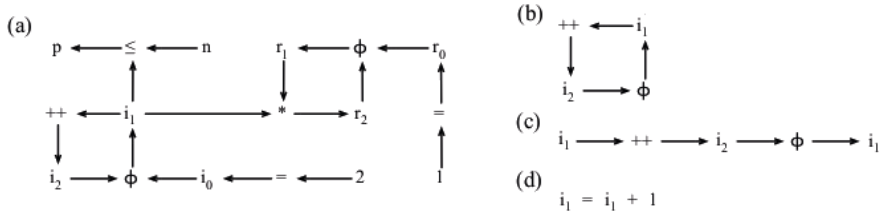

Fig. 2. (a)Dependence graph. (b)Multi-node SCC of the variable i_1. (c)Sequence of redefinitions of the variable i_1. (d)Effect of one iteration on the variable i_1.

program on a given variable. Figure 2 shows an example of SR for one induction variable.

Considering infinite precision, a SR has one of the following classifications:

- **Constant** - after one iteration of the SR, the value of the variable remains the same.
- **Increasing** - after one iteration, the value of the variable is always larger than the initial value.
- **Decreasing** - after one iteration, the value of the variable is always smaller than the initial value.
- **Possibly Oscillating** - after one iteration, we are not able to prove neither an increasing nor a decreasing behavior.

SRs that are classified as *Possibly Oscillating* are placed in this category because at least one of the following reasons is true:

- There is a call instruction in the SR. Currently our analysis is not able to analyze interprocedural SRs.
- There is an operation in the SR that receives an operand X, where the range analysis of X gives a positive upper bound and a negative lower bound.
- The SR depends on SCCs that have been classified as *Possibly Oscillating*.

We can classify a Multi-node SCC within the same categories of a single SR. For that, the classification of all the SRs of the SCC must be combined using a meet operation in the lattice shown in Figure 3. Therefore, the classification of a SCC is the least upper bound of the classifications of the SRs that the SCC contains.

Vectors

In order to achieve good precision without sacrificing efficiency, we propose an abstraction called vectors to predict trip count. We place each numeric variable v of the analyzed program on the real number line, in the point corresponding to the value stored by v. Whenever the value that v stores is changed, we move v to another point of the real line, corresponding to its new value. By doing that, we have observed that some variables have a well-defined behavior along loop iterations, that we can translate into patterns of movement. The vectors are,

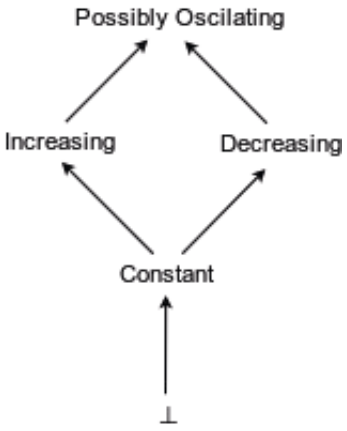

Fig. 3. Lattice of SR classifications

then, the structures that help us to understand those patterns of movement. We borrow the concept from linear physics vectors (magnitude, direction).

A vector is the step given by a variable v after one complete iteration through a SR p. Before the execution of p, we have v_0 stored in v. After the execution of p, v will be redefined with a new value, v_p. We can understand this redefinition of v as a move on the real number line. The step given by v in the line is $v_p - v_0$. Thus, a vector of a variable v extracted from a SR p is $\Delta v_p = v_p - v_0$.

The sign of Δv_p indicates the direction of the vector (i.e. the direction to where we are moving v). Vectors may be defined by symbolic expressions involving other variables of the program. This characteristic generates a chain of dependencies that brings the need to process the SCCs in topological order. If the SCC of a variable n is classified as *Possibly Oscillating*, then the vectors that depend on n are unknown.

Patterns of Movement

When variables have their values updated always by vectors with the same characteristics, some patterns of movement are noticeable:

- **Stationary** - variables updated by vectors with length equal to zero.
- **Constant Speed** - variables updated by vectors with constant length. In this case, on each iteration, the variable is moved a constant distance from its previous location, creating a linear behavior.
- **Constant Acceleration** - variables updated by a vector that has a linearly increasing length. This kind of vector is generated by a linear expression involving a *Constant Speed* variable, creating a quadratic behavior.
- **Constantly Increasing Acceleration** - variables updated by a vector that is generated by a linear expression involving a *Constant Acceleration* variable, creating a cubic behavior.

- **More Than Cubic** - variables updated by a vector that is generated by a linear expression involving a *Constantly Increasing Acceleration* or *More Than Cubic* variable, creating a more-than-cubic behavior.
- **Unknown** - Occurs when:
  - Variables are updated with vectors that depend on variables with *Unknown* movement patterns.
  - Variables are updated with vectors that have their length decreased on each iteration.

3 A Trip Count Algorithm Based on Vectors

Most of the loops have their number of iterations controlled by data that come from the input, so a purely static analysis is not precise enough to solve this problem. We propose, then, a hybrid solution involving a static and a dynamic step: we statically analyze the program and generate symbolic expressions that represent the estimated trip counts of its loops. Dynamically, during its execution, the instrumented program evaluates those expressions with O(1) complexity. Other optimizations might use the result of those expressions to make decisions at runtime, depending on the expected trip count. At the end of the compilation process, unused expressions can be trivially removed by a dead code elimination procedure.

Algorithm 1. Trip Count Instrumentation Based on Vectors

Input: Program P
Output: Program P with new instructions that estimate the maximum trip count of the loops

```
1: for all Loop l ∈ P do
2:     if isIntervalLoop(l) or isEqualLoop(l) then
3:         Variable op1 = getFirstOperand(l.getStopCondition())
4:         Variable op2 = getSecondOperand(l.getStopCondition())
5:         Expression step = estimateMinimumStep(op1, op2)
6:         if ∃ step then
7:             Insert instructions that compute the expression |op1 − op2|/step before the first iteration of l.
8:         end if
9:     end if
10: end for
```

Algorithm 1 presents the static analysis needed to generate the trip count expressions using vectors. Our heuristic only covers *Interval Loops* and *Equality Loops*. Once we have collected both operands op_1 and op_2 of the stop condition of a loop l, we have to estimate the step of approximation of the two variables in the real numbers line. In order to estimate the trip count of a loop, we must be able to evaluate op_1 and op_2 before the first iteration. Thus, both operands must be integer expressions that do not produce side effects when evaluated. Finally, if there is a well defined behavior of update of both operands, then $estimateMinimumStep(op_1,\ op_2)$ will return a valid step and we can estimate the trip count. Otherwise, we are not able to estimate the trip count of l.

Algorithm 2. estimateMinimumStep: Estimate the minimum step of approximation of variables in the real line.

Input: Pair of Variables $op_1,\ op_2$
Output: Expression *step* with the minimum step.

```
1: Vector v1 = getMinVector(op1)
2: Vector v2 = getMinVector(op2)
3: if ∃ v1 and ∃ v2 then
4:     return |v1 − v2|
5: else
6:     return null
7: end if
```

Algorithm 2 shows how we estimate the minimum step given the minimum vectors of the two variables that control the stop condition of the loop. The minimum step leads to the maximum trip count. Minor adaptations are required to generate the maximum step and, finally, the minimum trip count. The variables will only have a minimum vector if they have a monotonic behavior i.e. whenever the variables move in the real line, they move in the same direction.

Algorithm 3. getMinVector: Generate the minimum vector of a given monotonic variable

Input: Variable v
Output: Vector $\vec{V}$ with the minimum length.

```
1: Vector V⃗ = ⊥
2: for all RedefinitionSequence rs of v do
3:     Vector Tmp⃗ = evaluateDelta(rs)
4:     V⃗ = joinVectors_min(V⃗, Tmp⃗)
5:     if V⃗ == unknown then
6:         break
7:     end if
8: end for
9: return V⃗
```

Algorithm 3 generates the minimum vector for a given variable. In order to generate such vector, we have to symbolically evaluate every *Sequence of Redefinition* of that variable and join the results in a vector. The join operation has two steps: first we check the direction of both vectors. If the vectors have opposite directions or one of the vectors is *unknown*, the result of $joinVectors_min(\vec{V},\ \overrightarrow{Tmp})$ is *unknown*. Otherwise, we take the vector with the minimum length as the result of the join.

4 A Simplified Trip Count Algorithm Based on Vectors for JIT Compilers

With the massive increase of the usage of the World-Wide-Web and the introduction of many new architectures that must run the same programs, it is essential to have portable programs. Code interpreting provides easy portability of programs, because just the interpreter must be translated into the different

architectures, instead of any program of a given language. However, code interpreting is slow and excessively consumes resources. In this context, Just-In-Time (JIT) compilers are used to overcome the inefficiencies that code interpreters inherently have [15].

JIT compilers work by compiling pieces of code right before they are executed. Whenever the controller thread tries to execute some function that has no native code available, the controller thread calls the compiler before executing the function. That means that the execution stops while the JIT compiler is generating native code. Therefore, the JIT compiler must generate the most optimized possible code in the minimum time, because the total time (compiling + execution) must be lower than the interpreting time, otherwise there is no point in compiling those programs. Because of that, JIT compilers must use extremely lightweight algorithms to keep the compiling time as low as possible.

Here we present a simplification in our trip count prediction heuristic in order to be able to apply it in JIT compilers. As we have observed in Section 5, 90% of the natural loops are either Interval Loops or Equality Loops. Moreover, most of our vectors are constant speed vectors with length equal to one. From those facts, in the simple heuristic we assume that in every Interval or Equality loops the minimum step of approximation of op_1 and op_2 is equal to one. Thus, the estimated trip count is $|op_2 - op_1|$ and we avoid calling $EstimateMinimumStep(op_1,\ op_2)$. Our heuristic generates the expression that estimates the trip count with $O(1)$ complexity. The complexity of the analysis of the whole program is $O(n)$, where n is the number of natural loops of the program.

5 Experimental Results

We have implemented a prototype of our analyses in the LLVM compiler, version 3.3. We have analyzed more than 500 programs, including the benchmarks of the LLVM test-suite and the benchmarks of SPEC 2006 CPU. In this section we will focus the discussion in the results obtained with the analysis of the benchmarks of SPEC 2006 CPU.

Most of the loops of the programs have a simple structure, and that means that the analysis does not need to be complicated in order to cover almost all loops of a program. Table 1 analyzes the structure of natural loops of programs. According to Ferrante [5], natural loops are *single-entry* regions. We have observed that 65.92% of the loops have just one stop instruction, so they are *single-entry* and *single-exit* regions. However, 39.87% of the loops are nested inside other loops. Those numbers tell us that despite of the simplicity of most loops, a considerable amount of them is nested, so loop analyses that does not support nested loops leave a large number of loops uncovered.

We have also identified a pattern in the stop conditions of the loops. Table 2 shows that approximately 85% of the natural loops have a single integer comparison as the stop condition. Moreover, the vast majority of those loops are interval loops, the easiest kind of loop to analyze. We have also observed similar proportions while analyzing the rest of our benchmarks. Those numbers are

Table 1. Natural Loops in the Control Flow Graph. L: number of natural loops. NL: number of nested loops. SEL: number of loops that have a single exit point.

Program	L	NL	% NL/L	SEL	% SEL/L
433.milc	426	211	49.53%	399	93.66%
444.namd	623	418	67.09%	593	95.18%
447.dealII	6526	2695	41.30%	3412	52.28%
450.soplex	742	181	24.39%	554	74.66%
470.lbm	23	10	43.48%	23	100.00%
401.bzip2	238	85	35.71%	150	63.03%
403.gcc	4614	1357	29.41%	3202	69.40%
429.mcf	50	9	18.00%	39	78.00%
445.gobmk	1288	482	37.42%	913	70.89%
456.hmmer	881	245	27.81%	740	84.00%
458.sjeng	267	62	23.22%	201	75.28%
462.libquantum	98	13	13.27%	90	91.84%
464.h264ref	1870	1008	53.90%	1784	95.40%
471.omnetpp	465	66	14.19%	249	53.55%
473.astar	119	37	31.09%	104	87.39%
483.xalancbmk	3106	259	8.34%	1611	51.87%
Total	21336	7138	33.46%	14064	65.92%

Table 2. Classification of Natural Loops according to their stop conditions. L: number of natural loops. IL: number of *Interval Loops*. EL: number of *Equality Loops*. OL: number of *Other Loops*.

Program	L	IL	% IL/L	EL	% EL/L	OL	% OL/L
433.milc	426	417	97.89%	5	1.17%	4	0.94%
444.namd	623	494	79.29%	7	1.12%	122	19.58%
447.dealII	6526	4597	70.44%	604	9.26%	1325	20.30%
450.soplex	742	572	77.09%	101	13.61%	69	9.30%
470.lbm	23	23	100.00%	0	0.00%	0	0.00%
401.bzip2	238	201	84.45%	29	12.18%	8	3.36%
403.gcc	4614	2103	45.58%	1954	42.35%	557	12.07%
429.mcf	50	17	34.00%	28	56.00%	5	10.00%
445.gobmk	1288	1098	85.25%	131	10.17%	59	4.58%
456.hmmer	881	697	79.11%	109	12.37%	75	8.51%
458.sjeng	267	117	43.82%	128	47.94%	22	8.24%
462.libquantum	98	88	89.80%	6	6.12%	4	4.08%
464.h264ref	1870	1789	95.67%	19	1.02%	62	3.32%
471.omnetpp	465	283	60.86%	82	17.63%	100	21.51%
473.astar	119	108	90.76%	1	0.84%	10	8.40%
483.xalancbmk	3106	1687	54.31%	752	24.21%	667	21.47%
Total	21336	14291	66.98%	3956	18.54%	3089	14.48%

favorable to our heuristics, because we take advantage of the simplicity of the loops to produce precise results with simple algorithms.

Table 3 shows the statistics collected while analyzing the dependence graphs of the programs. 85.40% of the Multi-Node SCCs are Single-Path SCCs. That means that there is only one SR for the variables of such SCCs. Moreover, just 7.70% of the Multi-Node SCCs have nested cycles and do not fulfill the requirements of our analysis. All the presented data confirms that the programs have a structure that is suitable for our heuristics to produce accurate results.

In order to estimate the trip count of a loop, our prototype must be able to infer the values that Op_1 and Op_2 store before the first iteration. Op_1 and Op_2 are the operands of the stop condition of the loop. This information is not always

Table 3. Classification of Strongly Connected Components in the Dependence Graph. SN: number of *Single-Node* SCCs. MN: number of *Multi-Node* SCCs. SP: number of *Single-Path* SCCs. MP: number of *Multi-Path* SCCs. SL: number of *Single-Loop* SCCs. NL: number of *Nested-Loop* SCCs.

Program	SN	MN	SP	% SP/MN	MP	SL	% SL/MP	NL	% NL/MN
433.milc	2507	426	409	96.01%	17	11	64.71%	6	1.41%
444.namd	5879	781	604	77.34%	177	6	3.39%	171	21.90%
447.dealII	79169	7249	6077	83.83%	1172	505	43.09%	667	9.20%
450.soplex	13032	807	683	84.63%	124	53	42.74%	71	8.80%
470.lbm	94	24	23	95.83%	1	1	100.00%	0	0.00%
401.bzip2	3610	214	171	79.91%	43	16	37.21%	27	12.62%
403.gcc	123775	5121	4513	88.13%	608	276	45.39%	332	6.48%
429.mcf	1022	54	40	74.07%	14	7	50.00%	7	12.96%
445.gobmk	17675	1555	1283	82.51%	272	163	59.93%	109	7.01%
456.hmmer	12215	946	825	87.21%	121	67	55.37%	54	5.71%
458.sjeng	3337	276	221	80.07%	55	38	69.09%	17	6.16%
462.libquantum	1439	123	100	81.30%	23	8	34.78%	15	12.20%
464.h264ref	21502	1946	1841	94.60%	105	27	25.71%	78	4.01%
471.omnetpp	12383	470	379	80.64%	91	39	42.86%	52	11.06%
473.astar	2591	138	118	85.51%	20	7	35.00%	13	9.42%
483.xalancbmk	57181	3024	2486	82.21%	538	373	69.33%	165	5.46%
Total	357411	23154	19773	85.40%	3381	1597	47.23%	1784	7.70%

Table 4. Trip Count Instrumentation. IL: interval loops. IIL: instrumented interval loops. EL: equality loops. IEL: instrumented equality loops.

Program	# IL	# IIL	% IIL/IL	# EL	# IEL	% IEL/EL
433.milc	417	391	93.76%	5	3	60.00%
444.namd	494	469	94.94%	7	1	14.29%
447.dealII	4597	3535	76.90%	604	77	12.75%
450.soplex	572	422	73.78%	101	48	47.52%
470.lbm	23	23	100.00%	0	0	-
401.bzip2	201	186	92.54%	29	7	24.14%
403.gcc	2103	1798	85.50%	1954	192	9.83%
429.mcf	17	7	41.18%	28	1	3.57%
445.gobmk	1098	1040	94.72%	131	56	42.75%
456.hmmer	697	664	95.27%	109	39	35.78%
458.sjeng	117	106	90.60%	128	17	13.28%
462.libquantum	88	77	87.50%	6	1	16.67%
464.h264ref	1789	1411	78.87%	19	8	42.11%
471.omnetpp	283	238	84.10%	82	29	35.37%
473.astar	108	80	74.07%	1	1	100.00%
483.xalancbmk	1687	1403	83.17%	752	108	14.36%
Total	14291	11850	82.92%	3956	588	14.86%

possible to be inferred, because sometimes one of the operands is the result of a call to other function. In cases like this, we do not know the value of the operand before the loop starts and, consequently, we are not able to estimate its trip count. Thus, both operands must be integer expressions that do not produce side effects when evaluated. Table 4 shows the number of loops of which we are able to infer the trip count. For instance, we were able to estimate the trip count of 85.50% of the interval loops of the benchmark 403.gcc, while we were able to instrument just 9.83% of its equality loops. We have investigated this and we observed that most of the 403.gcc's equality loops are bounded by comparisons

between pointers. The same was observed in other programs. Because of that, we should focus only in the interval loops.

We have developed a profiler that collects the estimated trip count and the real trip count during an actual execution of the benchmarks. The result of our profiler lets us to observe how accurate are our heuristics. We have split our accuracy results into seven categories according to the actual number N of iterations:

- $[0, \sqrt{N}]$: Occurs when the estimated trip count is less or equal the square root of the actual trip count. For example, if we estimate that a loop will iterate 2 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]\sqrt{N}, N/2]$: Occurs when the estim ted trip count is greater than the square root of the actual trip count but is less or equal its half. For example, if we estimate that a loop will iterate 4 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]N/2, N[$: Occurs when the estimated trip count is greater than the half of the actual trip count but is less than the trip count. For example, if we estimate that a loop will iterate 8 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $[N, N]$: Occurs when the estimated trip count equals the actual trip count. For example, if we estimate that a loop will iterate 10 times and it iterates 10 times during its execution, this loop will fall into this category.
- $]N, 2 * N]$: Occurs when the estimated trip count is greater than the actual trip count, but is less or equal to two times the actual trip count. For example, if we estimate that a loop will iterate 16 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]2 * N, N^2]$: Occurs when the estimated trip count is greater than two times the actual trip count, but is less or equal to the power of two of the actual trip count. For example, if we estimate that a loop will iterate 32 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]N^2, +\infty]$: Occurs when the estimated trip count is greater than the power of two of the actual trip count. For example, if we estimate that a loop will iterate 128 times and it iterates 10 times during its execution, this loop will be classified into this category.

Table 5 shows the comparison between the estimated trip count and the actual trip count that we have collected with our profiler. The subtotal lines contain only the SPEC CPU benchmarks, while the total lines also include more than 300 benchmarks distributed with LLVM. While running the programs, each time a loop stops, we collect the actual trip count and compare it with the estimated trip count. Thus, the numbers that we presented is the number of *instances* of loops, instead of the number of natural loops. We did this because we may predict correctly the trip count for some instances and may predict wrongly for other instances of the same CFG loop. Table 6 shows information about the programs

Table 5. Trip Count Profiler - Trip count estimated using vectors

Program	$[0, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N[$	$[N, N]$	$]N, 2*N]$	$]2*N, N^2]$	$]N^2, +\infty]$
milc	14	0	0	435,514,912	38,360	9,984	1,032,930
namd	0	0	0	21,602,695	8,064	3,168	0
soplex	1,851	367	122	186,943	12,782	10,219	43,338
lbm	0	0	0	53,397	0	0	0
bzip2	8,616,650	2	311,724	13,204,855	14,195,603	1,128,948	28,939,274
gcc	433,588	17	326	17,240,735	1,851,284	278,164	336,422
mcf	96,576	87	42	555	2,643,736	634,623	1,705,369
gobmk	8,392	20	400	651,081	70,492	117	20,141
hmmer	0	0	0	31,551,408	8,512,797	3,893,744	3,273,429
sjeng	0	620	2,565,378	41,787,788	3,423,766	7,917	1,038,075
libquantum	0	0	0	8,182,095	0	1	0
h264ref	6,749,850	0	0	311,274,945	13,427,840	57,300	228,711
astar	7,147	0	0	74,614,711	602,244	2,550	609,708
Subtotal	15,914,068	1,113	2,877,992	955,866,120	44,786,968	6,026,735	37,227,397
Subtotal (%)	1.50%	0.00%	0.27%	89.95%	4.21%	0.57%	3.50%
Total	25,525,142	2,078	2,922,080	4,134,074,825	163,974,403	11,363,892	400,209,181
Total (%)	0.54%	0.00%	0.06%	87.25%	3.46%	0.24%	8.45%

Table 6. Trip Count Profiler - Trip count estimated using simplified heuristic

Program	$[0, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N[$	$[N, N]$	$]N, 2*N]$	$]2*N, N^2]$	$]N^2, +\infty]$
milc	14	0	0	435,514,912	38,360	9,984	1,032,930
namd	0	0	0	21,602,688	8,065	3,174	0
soplex	1,851	367	112	186,939	12,784	10,231	43,338
lbm	0	0	0	53,333	0	64	0
bzip2	5,270,006	2	311,724	14,386,219	15,987,072	1,502,759	28,939,274
gcc	420,390	17	326	17,252,944	1,841,701	283,373	343,054
mcf	96,576	87	42	555	2,643,736	634,623	1,705,369
gobmk	8,392	20	400	651,081	70,492	117	20,141
hmmer	0	0	0	31,551,408	8,512,797	3,893,744	3,273,429
sjeng	0	620	2,565,378	41,787,788	3,423,766	7,917	1,038,075
libquantum	0	0	0	8,182,095	0	1	0
h264ref	367,010	0	0	302,394,768	12,636,622	5,636,387	10,703,859
astar	7,147	0	0	74,614,711	602,244	2,550	609,708
Subtotal	6,171,386	1,113	2,877,982	948,179,441	45,777,639	11,984,924	47,709,177
Subtotal (%)	0.58%	0.00%	0.27%	89.22%	4.31%	1.13%	4.49%
Total	10,762,387	2,094	2,882,136	3,996,856,652	227,506,781	53,384,130	441,012,280
Total (%)	0.23%	0.00%	0.06%	84.46%	4.81%	1.13%	9.32%

that had their trip counts estimated using the simplified heuristic, following the same rules used to build table 5.

By analyzing table 5 we can observe that our heuristic is very precise. 87.25% of the trip counts that we have predicted were the same as the actual trip count. Furthermore, we have observed that more than 99% of the estimated trip counts were equal or greater than the actual trip counts. When we analyze the results obtained with the simplified heuristic, we also find some impressive numbers. As expected, the vector heuristic has better results than the simplified heuristic, but the difference was small. Our predictions were exact in 84.46% of the cases, despite of the extreme simplicity of the algorithm. We also have observed the same over-approximation that we have noticed with the complete vector heuristic. However, it is important to keep in mind that we are not able to instrument 100% of the loops of the programs. In this experiment we only consider Interval Loops, that account for 67% of them.

6 Related Works

It is possible to estimate the trip count of loops in many different ways, in a trade-off between speed and precision. In order to estimate the trip count of loops, many authors have used abstract interpretation [4,10,6,8]. Others have used symbolic execution to achieve similar goals [13,12]. Although those techniques are quite powerful, they are also computationally expensive. Thus, their application is limited by the size of programs to be analyzed. Nevertheless, the high complexity does not mean perfect precision. Some of those works have restrictions with regards to the structure of the analyzed loops. For instance, some of them only analyze loops with a single path and are very conservative while analyzing nested loops. Our work aims to find a better balance between speed and precision.

In an effort similar to ours, Gulwani *et al.* [7] have developed a new approach to estimate the number of iterations of a loop. They have proposed the *Control-Flow Refinement*, a conversion of the programs into a suitable representation, that allowed them to handle programs that other algorithms were not able to analyze. That representation allowed them to find symbolic bounds for 90% of the programs they have analyzed. However, they still rely on expensive techniques. For instance, their implementation requires a theorem prover. Such tools often rely on solutions to NP-complete problems. Differently to their work, here we present two heuristics to estimate the number of iterations of loops that use simpler techniques. Despite of the simplicity of our algorithms, our results show that we offer a good precision without resorting to expensive techniques.

7 Conclusion

In this paper we have discussed the prediction of the number of iterations of loops. We have indicated the usefulness of such information to decide at runtime which code to execute based on the estimated trip count of loops. We also have classified the loops into a taxonomy proposed by us, which allowed us to better understand where the most promising optimizing opportunities are. In addition, we proposed the Vectors, an abstraction inspired by physics to represent patterns of updates of variables on the real line. Furthermore, we have proposed two heuristics to estimate the trip count of loops, based on our Vectors. Finally, we have evaluated the precision of our heuristics on some test suites, observing 87.25% of accuracy while analyzing interval loops.

References

1. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010)
2. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java, 2nd edn. Cambridge University Press (2002)

3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Kenneth Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
4. Ermedahl, A., Gustafsson, J.: Deriving annotations for tight calculation of execution time. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 1298–1307. Springer, Heidelberg (1997)
5. Ferrante, J., Ottenstein, K., Warren, J.: The program dependence graph and its use in optimization. TOPLAS 9(3), 319–349 (1987)
6. Gulavani, B.S., Gulwani, S.: A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 370–384. Springer, Heidelberg (2008)
7. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: ACM Sigplan Notices, vol. 44, pp. 375–385. ACM (2009)
8. Gulwani, S., Mehra, K.K., Chilimbi, T.: Speed: precise and efficient static estimation of program computational complexity. In: ACM SIGPLAN Notices, vol. 44, pp. 127–139. ACM (2009)
9. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In: 27th IEEE International Real-Time Systems Symposium, RTSS 2006, pp. 57–66. IEEE (2006)
10. Halbwachs, N., Proy, Y.-E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods in System Design 11(2), 157–185 (1997)
11. Kennedy, K., Allen, R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
12. Liu, Y.A., Gomez, G.: Automatic accurate time-bound analysis for high-level languages. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 31–40. Springer, Heidelberg (1998)
13. Lundqvist, T., Stenström, P.: Integrating path and timing analysis using instruction-level simulation techniques. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 1–15. Springer, Heidelberg (1998)
14. Park, E., Cavazos, J., Pouchet, L.-N., Bastoul, C., Cohen, A., Sadayappan, P.: Predictive modeling in a polyhedral optimization space. International Journal of Parallel Programming 41(5), 704–750 (2013)
15. Plezbert, M.P., Cytron, R.K.: Does “just in time"=“better late than never"? In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 120–131. ACM (1997)
16. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society 74(2), 358–366 (1953)
17. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
18. Tetzlaff, D., Glesner, S.: Static prediction of loop iteration counts using machine learning to enable hot spot optimizations. In: 2013 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 300–307. IEEE (2013)
19. Wolfe, M.J., Shanklin, C., Ortega, L.: High performance compilers for parallel computing. Addison-Wesley Longman Publishing Co., Inc. (1995)