ILP-based Interprocedural Path Analysis

Henrik Theiling

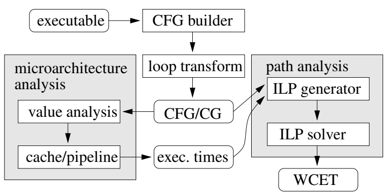
Universität des Saarlandes and AbsInt Angewandte Informatik GmbH
Saarbrücken, Germany

Abstract. Program analysis usually works on control flow graphs (CFGs) and on a call graph (CG). The standard CGs contain one node for each function, but for precise analyses, it may be desirable to distinguish function invocations by their execution history.
This distinction is useful, e.g., to improve the precision of worst-case execution time (WCET) analysis for real-time systems. Our WCET analysis supports these advanced techniques for interprocedural analysis. The first part of the WCET analysis, i.e, the prediction of microarchitecture behaviour, uses Abstract Interpretation, for which tools already support the methods for function distinction by execution history.
The second part of WCET prediction is the worst-case path analysis, which can be performed using the well-established technique of Implicit Path Enumeration using Integer Linear Programming.
So far, support for arbitrary interprocedural analysis techniques in one framework was not discussed in literature. This paper closes this gap.

1 Introduction

Worst-case execution time (WCET) analysis can be split into two steps. The first step is the microarchitecture analysis. Like most program analyses, it works on control flow graphs, containing basic blocks as nodes, and on call graphs. The result of the first step is a worst-case execution time for each basic block of the program under examination.

The microarchitecture analysis often consist of a chain of sub-analyses for different parts of the hardware, like cache and pipeline analyses.



In our research project[1], we analyse the WCET of real-time systems. For all the microarchitecture sub-analyses, Abstract Interpretation (AI) (*see* [2]) is used.

The second part of a WCET analysis is the worst-case path analysis. Based on the results of the microarchitecture analysis, it computes the predicted WCET of the program.

[1] Transferbereich 14 'Runtime Guarantees for Real-Time Systems', supported by Deutsche Forschungsgesellschaft

Path analyses can be implemented in several ways. Because of good precision and speed, we use Implicit Path Enumeration (IPE) (*see* [11]), which uses Integer Linear Programming (ILP) to find the WCET.

For improving the precision of program analysis, function invocations and loops can be distinguished by their execution history, e.g., by the *call stack* as distinctive features, which we call execution *contexts*.

To make use of the contexts in analyses, nodes of the analysis graphs are pairs of original nodes and assigned contexts, so nodes in the original graph are potentially transformed into several nodes in the analysis graphs, due to the context distinction.

Our tool PAG (*see* [14]) for writing analyses using AI comes with interprocedural analysis methods, so these advanced techniques are directly usable by our microarchitecture analysis.

Previous work extended IPE to interprocedural analysis techniques (*see* [26, 27]), showing that the combination is possible in principle. It was shown to work for one context computation method (which was sufficient for showing the principle).

This paper shows how *arbitrary context computation techniques* can be combined with IPE as long as the contexts are computed statically. This makes our WCET tool open to other approaches not yet programmed or thought of, and to other distinction methods, e.g., those used by other work groups, contributing a lot to genericity of our tool.

ILP-based path analysis (*see* [11]) generates an *objective function* and a set of constraints.

Loops in the call graph pose a problem, because the structure of the original call graph without contexts and the call graph the analysis works on (with context) is different w.r.t. to the structure of loops: entry and back edges of loops in the original call graph are not necessarily entry and back edges of loops in the call graph with context.

The biggest problem is that contexts at the entry edge and at the back edges of some loop are usually different, because back edges have a longer execution history (because they are traversed after the entry edges).

This paper will deal with the trade-off between analysis precision and speed. High precision by using contexts is desired, but a distinction by the whole execution history usually is too expensive. For best results, context computation should be most flexible and should be limitable and adjustable for differents programs under examination. We will outline an ILP-based path analyses for arbitrary static context computations.

The paper is structured as follows. Section 2 will give technical definitions, Section 3 shows how ILPs are generated. Section 4 gives an overview of the experimental results, Section 5 compares this work to related work, and finally Section 6 draws conclusions.

2 Basics

Program analyses usually work on two graphs: the Control Flow Graph (CFG), containing nodes for each basic block and edges for intraprocedural control flow, and on the

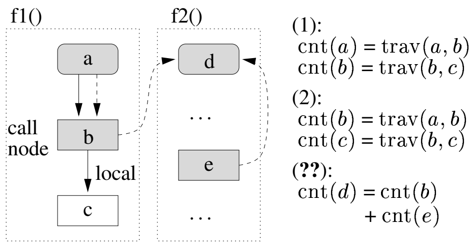


Fig. 1. CFG and CG of a call of a recursive function. All nodes are part of CFG, shaded ones are part of CG. Rounded boxes identify function start nodes. Solid edges are in the CFG, dashed edges in the CG.

Call Graph (CG), containing nodes for each function and function call and edges for all function invocations.

Our CFGs contain a *local* edge after each call node, because the call graphs will not contain flow information about function returns, but it is needed for the generation of correct constraints (*see* Section 3). This is shown in Figure 1.

Definition 1. *Let $\mathcal{P}$ be the* ***program*** *under examination.*

Let F be the set of ***functions*** *of $\mathcal{P}$. Let $V_f, f \in F$ be the* ***basic blocks*** *of each function f (the V_f are pairwise disjoint). $V = \bigcup_{f \in F} V_f$ then is the set of all basic blocks of $\mathcal{P}$.*

Let $\mathit{CFG}_f = (V_f, E_f), f \in F$ be the ***control flow graph*** *of function f.*

Let $\mathrm{Calls} \subseteq V$ be the set of nodes containing ***function calls*** *of $\mathcal{P}$ and let $\mathrm{Starts} \subseteq V$ be the set of function* ***start nodes****. $\forall c \in \mathrm{Calls}$ let $\mathrm{target}(c) \in \mathrm{Starts}$ be the* ***target*** *of the function call c. $\mathrm{target}(c)$ defines the interprocedural call edges of the call graph. Note that our call graphs do not contain return edges.*

For a path in graph G from node v_1 to v_2, we will write $v_1 \rightarrow^*_G v_2$.

Definition 2. *Let $\mathit{CG} = (\hat{V}, \hat{E}), \hat{V} = \mathrm{Calls} \cup \mathrm{Starts}, \hat{E} \subseteq \hat{V} \times \hat{V}$ be the* ***call graph*** *of $\mathcal{P}$, where $\hat{E}$ is defined as follows. $\hat{E} :=$*

$$\{(c, s) : c \in \mathrm{Calls}, s = \mathrm{target}(c)\} \cup \bigcup_{f \in F} \{(s, c) : s \in \mathrm{Starts}, c \in \mathrm{Calls} : \exists s \rightarrow^*_{\mathit{CFG}_f} c\}$$

The first set in the definition of $\hat{E}$ consists of *call edges*, the second set consists of edges from the start node to all reachable call nodes in the same function.

It is required that call nodes have exactly one incoming edge in the CFG. (This can be ensured by inserting additional empty nodes for the call nodes that contradict this requirement.) This way, together with the above definitions, each call node c also has exactly one outgoing edge in the CFG, namely the local edge. In the CG, c also has exactly one incoming edge (from the the start node) and one outgoing edge (defined by $\mathrm{target}(c)$) in the CG. By this, each traversal of these edges corresponds to exactly one execution of c.

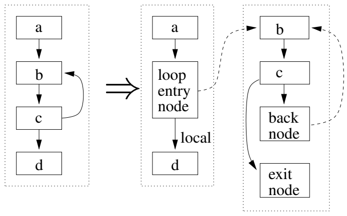
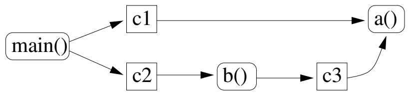


Fig. 2. CFG modifications by loop transformation. Dotted rectangles show functions. The loop transformation introduces a new function and new call nodes for each loop and transforms the loop into a recursive function. Dashed lines represent edges in the call graph, that are introduced by the transformation.

Example in C:

```
void a() {
    ...    // basic block b1
}
void b() {
    a();  // invocation c3
}
int main() {
    a();  // invocation c1
    b();  // invocation c2
}
```



CG (without context). Start nodes are labelled with the name of the function and a pair of parentheses, call nodes are labelled as shown in the comment in the C source code. Note that our CGs contain no return edges, only call edges.

Loops: We handle loops and recursion uniformly in our approach. This is done by transforming all loops into recursive functions by making the loop body a function on its own and inserting interprocedural edges accordingly. Figure 2 depicts this and also clarifies the node names that we will use in the following text:

loop entry nodes are call nodes that invoke a recursive function from outside of that loop.

back nodes are call nodes that invoke a recursive function from inside of that loop.

Definition 3. *Let L be the set of **loops**, and $\forall\, l \in L$: let $\mathrm{Entries}(l) \subset \mathrm{Calls}$ be the set of **loop entry calls** (the sources of each loop entry edges of l), and let $\mathrm{Backs}(l) \subset \mathrm{Calls}$ be the set of **back calls** (the sources of each back edge of l)*

Contexts: Our framework provides the possibility to statically categorise different executions of basic blocks in different classes. These classes are called contexts and are usually computed by considering the execution history of a function.

Contexts are computed in different ways, depending on the desired level of precision and the acceptable computation effort for the specific analysis problem. The computation depends on the execution history by which the basic block is reached. The computation of contexts will be called a *mapping*. Each basic block must only be assigned a finite number of contexts.

Mappings that depend on the history of calls (often represented by a call stack), i.e., on the invocation history of a function, are said to follow the *call graph approach*. $\mathsf{CallString}(k)$ is a mapping that uses a suffix of length k of the call history. The most simple mapping method is $\mathsf{CallString}(0)$, which assigns the same, empty context to every basic block, thus making no distinction by execution history. k, the context length bound, is used to make the context mapping computable and to limit the complexity of the mapping to make the analyses perform well.

In Example 2, basic block `b1` in function `a()` is executed with two different call stacks: `c1` and `c2 ∘ c3` depending on the control flow that led to the invocation of function `a`. Starting from $\mathsf{CallString}(2)$, we thus get two contexts for basic block `b1`.

A mapping will be seen more or less as a black box that computes graphs we use for analysis based in the original CFGs and CG.

Nodes in these CFGs and CGs with contexts are pairs of basic blocks and contexts. We show the presence of context information by writing these graphs with a suffixed '$\star$':

Control flow graph:	$\mathsf{CFG}^\star = ({V_f}^\star, {E_f}^\star)$
Set of all nodes:	$V^\star = \bigcup_{f \in F} {V_f}^\star$
Call Graph:	$\mathsf{CG}^\star = (\hat{V}^\star, \hat{E}^\star)$

The sets $\mathrm{Calls}^\star$ and $\mathrm{Starts}^\star$ are defined for nodes in CG in all contexts:

$$\begin{aligned} \mathrm{Calls}^\star &= \{(v,c) : (v,c) \in \hat{V}^\star, v \in \mathrm{Calls}\} \\ \mathrm{Starts}^\star &= \{(v,c) : (v,c) \in \hat{V}^\star, v \in \mathrm{Starts}\} \end{aligned}$$

The graphs the mapping computes are not totally arbitrary. Nodes and edges in $\mathsf{CG}^\star$ always have corresponding nodes and edges in CG.

$$\begin{aligned} (v,c) \in \hat{V}^\star &\Rightarrow v \in \hat{V} \\ (v_1,c_1) \rightarrow_{\mathsf{CG}^\star} (v_2,c_2) &\Rightarrow v_1 \rightarrow_{\mathsf{CG}} v_2 \end{aligned}$$

Further, edges inside functions do not change context:

$$\forall f \in F, v_1, v_2 \in V_f : (v_1,c_1) \rightarrow_{\mathsf{CG}^\star} (v_2,c_2) \Rightarrow c_1 = c_2$$

Figure 3 depicts the Example 2's $\mathsf{CG}^\star$. It can be seen from the figure that the $\mathsf{CallString}(2)$ $\mathsf{CG}^\star$ is maximally precise w.r.t. control flow, i.e., no control flow from two different calls ever joins in deeper calls. In contrast to this, in the $\mathsf{CallString}(0)$ $\mathsf{CG}^\star$ on the right, control flow from the calls `c1` and `c2` joins when `a()` is invoked from `c3`.

Consider the following program:

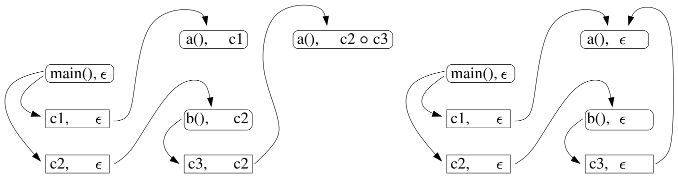


Fig. 3. CG*: each node is a pair of basic block and context. a) left: CallString(2), b) right: CallString(0). b) is isomorphic to the CG of this program.

```
void a (int x)
{   ...
    a(x-1);    // c2
    ...
}

int main (int, char**)
{
     ...
     a(5);     // c1
     ...
}
```

The contexts for CallString(2) and CallString(1) are as follows:

	CallString(2)	CallString(1)
for main()	ε	ε
for a()	c1	c1
	c1 ∘ c2	c2
	c2 ∘ c2	

As can be seen, the 'older' parts of the execution history are chopped off. This is the main reason that causes problems for the loop bound computation as will be seen later.

Recursive Example: This section will clarify how different mappings make different CGs*. Consider the following program:

```
void a(int i) {
    ...
         a(...); // c3
    ...
}
int main() {
    a (10);      // c1
    a (20);      // c2
}
```

Figure 4 shows the CG and some CGs* for different context mappings.

VIVU(n,k): For better precision of loop analysis, we have developed a mapping technique that distinguishes the first few iterations from all other iterations, which are joined in one context. By this, we *virtually unroll* loops. To do this, the CallString(k) approach is modified to add a *saturated counter* to each context element in order to count how often the corresponding call occurs on the call stack via which a basic block is reached. Saturation of the counter is indicated by the symbol $\top$. Loop back nodes do not occur in contexts anymore, but instead, the corresponding counter is incremented. VIVU(n,k) is the mapping where n distinctions are the maximum, i.e., the counter may have the values $1,\ldots,n-1,\top$, and where the contexts may not be longer than k elements.

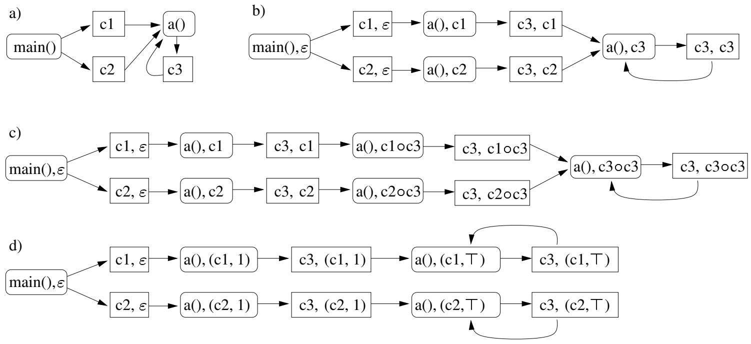


Fig. 4. a) CG of Example 2. b) CG* with CallString(1) mapping: the distinction by the two calls from `main()` is always lost in `a()`, because of the recursion. c) CG* with CallString(2): regardless of the maximum length, `a()` is never distinguished by the call from `main()`. d) CG* with VIVU$(2, 1)$ mapping: the two calls in main remain distinguished in spite of the recursion.

Figure 4 also shows a VIVU(n, k) mapping for the example recursion to show how the mapping works.

Most significantly, the counter element of VIVU(n, k) prevents that the CG* contains joints for every recursion, because the contexts are not shifted, but simply a counter is incremented. This yields better precision w.r.t. execution history.

Times and Execution Counts: In the following we will use the following symbols.

Definition 4. *The **execution time** per execution of a node in a graph $G = (V, E)$ will be written $\mathrm{t}(v), v \in V$. In the ILP, these symbols are constants provided by previous analyses.*

*The **execution count** of a node in G will be written $\mathrm{cnt}(v), v \in V$. The traversal count of an edge in G will be written $\mathrm{trav}(e), e \in E$. In the ILP, these symbols are variables and will be derived from a solution of the ILP.*

*The **minimum loop execution count** per entrance of that loop from a call node $v \in V$ will be written $\mathrm{m}(v)$. The **maximum loop execution count** per entrance will be written $M(v)$. In the ILP, these symbols are constants provided either by previous analyses or by the user.*

All these symbols can be used for all kinds of graphs, i.e., with or without context information.

3 ILP

This section briefly describes how an ILP is generated for worst case path analysis. The techniques are described in detail in previous work, e.g. in [11, **?**].

In the following, the generation of ILPs on graphs with context information is outlined.

Objective Function: The objective function sums up execution cycles of the program. Each basic block's total execution time contributes to this sum. The total execution time per block is the product of the time of one execution and the number of times it is executed. The sum is to be maximised to compute the WCET of the program: $\max : \sum_{v \in V^\star} \mathrm{cnt}(v) \cdot \mathrm{t}(v)$.

Our framework also allows the user to weigh edges of the program for more flexibility and higher precision. This is especially interesting for the pipeline analysis.

Program Start Constraint: Let v_0 be the start node of the program. Since the WCET for *one* execution of the program is to be derived, its execution count must be 1 (we do not permit recursion back to the program entry): $\mathrm{cnt}(v_0) = 1$.

Structural Constraints: For all nodes, we sum up the outgoing and incoming control flow. The following constraints are generated from the CFGs*.

$$\forall f \in F, v \in V_f{}^\star, \{(v, v') \in E_f{}^\star\} \neq \{\,\} : \mathrm{cnt}(v) = \sum_{(v,v') \in E_f{}^\star} \mathrm{trav}(v, v') \quad (1)$$

$$\forall f \in F, v \in V_f{}^\star, \{(v', v) \in E_f{}^\star\} \neq \{\,\} : \mathrm{cnt}(v) = \sum_{(v',v) \in E_f{}^\star} \mathrm{trav}(v', v) \quad (2)$$

Call nodes are also handled correctly by these constraints (*see* Figure 1).

Care has to be taken at nodes with no incoming edges (e.g. start nodes of functions) or no outgoing edges (e.g. exit nodes of functions), since it is usually wrong to generate constraints of the form $\mathrm{cnt}(v) = 0$, as these nodes are really potentially executed. Therefore, these constraints are excluded in the description above.

For each unreachable node $v \in V^\star$, we generate constraints of the form $\mathrm{cnt}(v) = 0$.

From the CG* we have to generate the following equations that state that start nodes are executed as often as the sum of their call nodes.

$$\forall v \in \mathrm{Starts}^\star : \mathrm{cnt}(v) = \sum_{(v',v) \in \hat{E}^\star} \mathrm{cnt}(v')$$

Loop Constraints: Loop constraints bound the number of iterations of a loop. They are specified as the minimum and maximum number of iterations for each invocation of the loop. Because the ILP-based approach adds up the execution counts on the loop entry nodes, the most precise measure is the *ratio between the number of executions of the loop entry node and the number of executions of the start node of the loop*. As defined before, for a loop entry node n_0, the minimum and maximum ratios are called $m(v_0)$ and $M(v_0)$, resp. (*see* Definition 4).

Note that there may be more than one loop entry node for recursive loops (but not for transformed iterative loops). Therefore, the minimum and maximum loop counts are given for each loop entry node, not for each loop.

In the case of graphs without context, loop bound constraints are generated as follows. For each loop l:

$$a := \sum_{v \in \text{entries}(l)} \text{cnt}(v), \quad b := \sum_{v \in \text{back}(l)} \text{cnt}(v)$$
$$\textstyle\sum_{v \in \text{entries}(l)} \text{cnt}(v) \cdot m(v) \leq a + b \leq \displaystyle\sum_{v \in \text{entries}(l)} \text{cnt}(v) \cdot M(v)$$

(Note that $a + b$ is the number of times the start node of the recursive function is executed.)

Here, we add all the minimum and maximum loop counts to generate constraints. This method has poor precision. Consider Example 2 together with Figure 4d. The two loop entries `c1` and `c2` from `main()` are fully distinguished by this mapping. Lets assume that we know that the loop is executed 11 times when entered from `c1` and 21 times when entered from `c2`. Lets further assume that `a()` is much slower (e.g. because of pipeline effects) when entered from `c1`.

If we combined the two loop bounds in one constraint, because we didn't want to distinguish the two loop entries, we could only state that the total is 32 iterations for the two calls together. And because in the first invocation, `a()` is slower, the worst-case solution for the ILP would be that the first iteration had 31 iterations, the second only 1. So this is much less precise than having an own loop bound constraint for each of the two invocations.

This is the reason why we want to keep the distinction for loop bound constraints as precise as the mapping allows.

Generating constraints for loop bounds in call-graphs with contexts is complicated. The reason is that the edges that contribute to the loop bound constraints might occur in *different* contexts which depend on each other in non-trivial ways, depending on the used mapping.

The major problem is the *context length bound* that most mappings use to ensure computability and to reduce complexity. This bound leads to collapses of context distinctions in the control flow from different loop entry edges. This then leads to the fact that edge traversal counts e.g. for the back edges sum up for different loop entries. Our previous work presented in [26] and [27] handled a simpler case, because no length bound was involved in mapping computation.

To generate correct loop bound constraints, it is necessary to consider all loop entries that join in a later part of the $\mathsf{CG}^\star$. This will require a graph algorithm that collects the appropriate edge contexts.

Idea: The loop bound constraints that have to be generated are very similar to the ones shown for graphs without context. One difference is that the nodes are now pairs of basic block and context. Because the constraints compare non-adjacent nodes in the $\mathsf{CG}^\star$, it is quite unclear which contexts to use for the nodes. Let us neglect this problem for now and assume these contexts are known.

For each loop l and each loop entry node $v_0 \in V^\star$, we need the correct set of back nodes with the precise contexts $\mathcal{B}(v_0) \subseteq V^\star$ and the correct set of entry nodes with the precise contexts $\mathcal{V}(v_0) \subseteq V^\star$ to generate the loop bound constraints as before:

$$a := \sum_{v \in \mathcal{V}(v_0)} \mathrm{cnt}(v), \quad b := \sum_{v \in \mathcal{B}(v_0)} \mathrm{cnt}(v)$$

$$\sum_{v \in \mathcal{V}(v_0)} \mathrm{cnt}(v) \cdot m(v) \leq a + b \leq \sum_{v \in \mathcal{V}(v_0)} \mathrm{cnt}(v) \cdot M(v)$$

The problem of the algorithm is thus reduced to finding $\mathcal{V}$ and $\mathcal{B}$. The most simplistic approach would be to ignore the contexts and sum up *all* loop entry calls in all their contexts and all back nodes in all their contexts. However, we lose a lot of precision in doing so, therefore, we want to search for those $\mathcal{V}$ and $\mathcal{B}$ that are most precise for the given mapping, i.e., those that allow the maximal number of distinctions between loop invocations.

Consider Figure 4 again. There is only one loop l (in the CG in 4a), for which $\mathrm{Entries}(l) = \{\mathtt{c1}, \mathtt{c2}\}$ and $\mathrm{Backs}(l) = \{\mathtt{c3}\}$.

If in $\mathsf{CG}^\star$, l is still distinguished by the invocation in `main`, then finding the back nodes that correspond to, e.g. `c1`, is easy, since `c1` must be part of the context. In Figure 4d, these are the back node $(\mathtt{c3}, (\mathtt{c1}, 1))$ (back node of the first iteration) and $(\mathtt{c3}, (\mathtt{c1}, \top))$ (back node of all other iterations). This was the basis of the computations performed in previous work (*see* [27]).

It becomes more complicated when the distinction is lost as in Figure 4b. If we want to generate loop bound constraints for the loop entry node $(\mathtt{c1}, \varepsilon)$, we will have to compute the correct set of back nodes for this entry node. The nodes $(\mathtt{c3}, \mathtt{c1})$ and $(\mathtt{c3}, \mathtt{c3})$ are clearly back nodes for this call, as they are reachable from that loop entry node.

Unfortunately, $(\mathtt{c3}, \mathtt{c3})$ also belongs to the set of back nodes that is reachable from $(\mathtt{c2}, \varepsilon)$, so we cannot handle the two loop entry nodes separately, since the execution count of the back node $(\mathtt{c3}, \mathtt{c3})$ contributes to both invocations. This must, therefore, be accounted for by summing up *in one constraint* the execution counts of *all corresponding loop entry calls* for the back nodes that we have found.

For the set of found back nodes for the call `c1`, we have to search for all the other call nodes whose corresponding set of back nodes intersects with the one we just found. If we do this in the above example, we end up with both initial loop entry nodes to the loop, $(\mathtt{c1}, \varepsilon)$ and $(\mathtt{c2}, \varepsilon)$. To find these, we simply search backward in $\mathsf{CG}^\star$ for a path from the back nodes to the loop entry node.

Finally, we need to extend the set of back nodes to those reachable from all the newly found loop entry nodes. In the example, we find the additional back node $(\mathtt{c3}, \mathtt{c2})$.

Intuitively, for nested loops, care has to be taken. Indeed, reachability of nodes must be computed *without regarding return edges* (from the called function back to the caller). Otherwise, the loop we are searching back nodes for might be reachable in a different context by first using the return edge and then re-entering the loop in a different context.

Our call graphs do not contain return edges, so this does not give rise to any problems (*see* Definition 2).

Algorithm: The algorithm for finding $\mathcal{V}$ and $\mathcal{B}$ is defined as follows. Let l be a loop and v_0 be a loop entry call for which we want to generate a loop bound constraint. First we define the preliminary set of back nodes $\mathcal{B}'(v_0)$, then we define $\mathcal{V}(v_0)$ and $\mathcal{B}(v_0)$.

$$\begin{aligned}
\mathcal{B}'(v_0) &:= \{(b,c) \in \hat{V}^\star : b \in \mathrm{Backs}(l), v_0 \rightarrow^*_{\mathsf{CG}^\star} (b,c)\} \\
\mathcal{V}(v_0) &:= \{(b,c) \in \hat{V}^\star : b \in \mathrm{Entries}(l), \exists v \in \mathcal{B}'(v_0) : (b,c) \rightarrow^*_{\mathsf{CG}^\star} v\} \\
\mathcal{B}(v_0) &:= \{(b,c) \in \hat{V}^\star : b \in \mathrm{Backs}(l), \exists v \in \mathcal{V}(v_0) : v \rightarrow^*_{\mathsf{CG}^\star} (b,c)\}
\end{aligned}$$

4 Experimental Results

We use a very efficient implementation based on hash tables for storage of the graphs and the sets that are involved. By this, it is no problem to generate $\mathcal{B}$ and $\mathcal{V}$ in just a few seconds even for large graphs with thousands of nodes in the $\mathsf{CG}^\star$. Usually, loops are very local, so the algorithm does not need to search long paths to find the desired nodes. Furthermore, only a fraction of the nodes in the call graph are loop nodes, so the scope of reachability search is limited. Moreover, back nodes are very often located in the same function as the loop header, so reachability search can be performed very locally.

In order to measure only the path analysis, programs were considered without microarchitecture analysis. Of cause, for real-life examples, we use the microarchitecture analysis of our framework (*see* [6]).

Further, for comparing a wide range of mappings, from simple ones to very complex ones, in the first step of experiments, we used a small program for an embedded controller (which operates and updates a liquid crystal display) and analysed it with very different mappings. The structure of the program involves 4 loops and the nesting level is maximally 3. By using different, complex mappings, the biggest ILPs became much larger than those of our real-life applications' ILPs with some thousand constraints. (Due to the loop nesting level of 3 and the expected solving time being quadratic for nice ILPs, the number of contexts is expected to increase cubically when increasing the unroll count, and the run-time of the solve by the power of 6. So this is a good way to increase complexity without using large programs.)

Figure 5 compares mappings and their sizes (number of contexts) with the generation and solving times of the corresponding ILP. The tests were done on an AMD Athlon XP1900+ machine with 1,600 MHz clock and 512 MB main memory running SuSE Linux 7.2 (kernel 2.4.10). The largest example, $\mathsf{VIVU}(5,16)$ had 45,000 nodes and 58,000 edges in the $\mathsf{CFG}^\star$ which yielded over 100,000 variables in the ILP before the ILP simplification.

The figure is organised as follows: first, the loop unrolling level is increased to generate ILPs of considerable size. Then, the context length limitation is used in order to limit the number of contexts. Long unroll counts meaning good loop analysis precision can then be analysed in a reasonable amount of time. In the third part of the table, it is shown that call string approaches are also analysable with IPE.

mapping	#ctxts	#constr.	gen.time[s]	solv.time[s]
VIVU(∞, 1)	8	49	< 1	< 1
VIVU(∞, 5)	176	773	1.5	< 1
VIVU(∞, 10)	1,151	4,783	9	7
VIVU(∞, 12)	1,933	7,969	16	39
VIVU(∞, 13)	2,432	9,997	22	69
VIVU(∞, 14)	3,011	12,347	28	111
VIVU(∞, 15)	3,676	15,043	36	176
VIVU(∞, 16)	4,483	18,109	45	409
VIVU(3, 12)	386	1,261	2.5	< 1
VIVU(3, 13)	444	1,443	3.5	< 1

mapping	#ctxts	#constr.	gen.time[s]	solv.time[s]
VIVU(3, 14)	506	1,637	4	1
VIVU(3, 15)	572	1,843	4	3
VIVU(3, 16)	642	2,061	5	1.5
VIVU(2, 16)	356	1,450	3	1
VIVU(1, 16)	68	233	< 1	< 1
CallString(0)	0	49	< 1	< 1
CallString(5)	119	281	< 1	< 1
CallString(10)	594	1,081	2	< 1
CallString(15)	1,794	2,781	6	2
CallString(20)	4,094	6,031	12	18

Fig. 5. For a fixed embedded controller program, the mappings, the number of contexts, number of non-trivial constraint, and the time to generate and solve the ILP are shown. Of course, large unrolling counts result in a lot of contexts due to the nested loops. (Note that a constraint simplification reduces the ILP to non-trivial constraints only.)

Our approach was implemented in order to improve the analysis time of large executables. W.r.t. the path analysis, the standard examples that are widely used for experiments (e.g. FFT, Bubblesort, Matrix Multiplication, Circle Drawing Algorithm, Prime Test) all resulted in an optimal 100%.

In a second step, we ran tests of real-life programs from safety critical embedded hard real-time systems for the Motorola ColdFire MCF 5307 (*see* [17]) architecture. Due to the complex nature of these programs, the real WCET is not known.

So the most interesting measure is the analysis speed for real-life executables.

The framework (*see* [6]) we used for finding the run-time of the path analysis comes with microarchitecture analyses for value analysis (*see* [22, 7]), (instruction and data) cache analysis (*see* [5, 27]), and pipeline and memory bus analysis (*see* [21]). These can be parameterised (at least) with a specific mapping. Loops are transformed at the beginning of the analysis suite, just after the ELF executable's CFG has been reconstructed (*see* [24, 25]). The loop transformation also computes the sets $\mathrm{Entries}(l)$ and $\mathrm{Backs}(l)$ for each loop l.

The pipeline and memory bus analysis, which runs just before the path analysis, yields execution times for each basic block/context pair. The path analysis then reads these results and also the loop-transformed control flow graph, has the needed $\mathsf{CG}^\star$ computed by the given mapping and then generates constraints. In the last step of this generation, loop bound constraints are generated using the algorithm described above. After the ILP has been generated, `lp_solve`[2] is used to solve the ILP.

For us, it was most interesting to see how fast the ILP-generator is, and also how long the ILP takes to be solved. The solving step has quite a high theoretical worst-case run-time, namely exponential, due to the used branch-and-bound technique (which can be reduced to polynomial run-time, if the ILP was relaxated to an LP). With respect to this theoretical worst case run-time, an ILP with some thousand variables and constraints is quite threatening. Relievingly, the generated ILPs will be very similar to network flow structures and are, therefore, quickly solvable.

[2] `lp_solve` was written by Michel Berkelaar and is freely available at `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

For further speed tests, we used twelve test programs from a real-life hard real-time application that all had a size in the order of about 50kB. The target architecture was the Motorola ColdFire MCF 5307 processor (*see* [17]). We used several mappings for testing, including our most precise mapping, namely VIVU without context length restriction, which resulted in the largest ILPs.

The largest $\mathsf{CFG}^\star$ had over 15,000 nodes (basic blocks) and over 20,000 edges. The largest $\mathsf{CG}^\star$ had over 2,300 nodes (function starts and calls) and over 4,500 edges. The ILP generation for any of these executables never took longer than 30 seconds on an Athlon processor with 700 MHz and 768 MB main memory, so the algorithm can be said to perform very well.

The resulting ILPs had some thousand non-trivial variables (more than 8000 plus over 28000 pairs of variables that were removed due to constraints of the form $x = y$) and also some thousand constraints (more than 6800). Still, solving took less than a minute even for our most precise mappings. So this also shows very good performance.

5 Related Work

Work about ILP-based path analysis was first published by Li et.al. in [9, 10, 8] who extended his work to include non-direct-mapped cache analysis in [11]. For caches with high associativity, a pure ILP-approach is not feasible using that technique due to complexity problems. In [26] an alternative approach was presented that split off the microarchitecture modelling from the path analysis by ILP. The method is usable for simple mappings. In contrast to this, this new paper presents how ILPs can be computed for all statically computed mappings. Analysis techniques using contexts are described in detail in [16, 15].

Ottoson and Sjödin also used ILP-based path analysis with a different objective function weighing the edges instead of the nodes (*see* [19]). In total, the basic technique can be said to be well-established. However, to our best knowledge, no other work group yet published the use of basic block contexts for precision improvement of ILP-based path analysis.

Lim et. al. have proposed a method of WCET computation called extended timing schema (*see* [12, 13]). The approach also does not handle contexts.

In [4], Engblom and Ermedahl present a technique of constructing an ILP from a scope graph and a flow graph in order to generate structural and loop constraints. Their approach handles a simpler case of constraint generation than this paper, since their scope graphs are rather trees, so they do not have any joints in the graph resulting from collapsed scope distinction inside a loop that is entered from different scopes (their scope graph is a perfect hierarchy, which we do not have in the general case). Since our context computation algorithms provide the method of bounding complexity, and therefore producing complex relationships between contexts, handling this was an important design goal of our algorithm.

Another technique of path search has been proposed by Stappert et.al. (*see* [23]) where instead of extensive path search or implicit path enumeration by ILP, an approach based on acyclic directed graphs (DAGs) is proposed. The result is a fast way of computing the WCET using a well-known graph algorithm (Dijkstra). The techniques for context distinction are the same as in [4], so the differences mentioned in the previous paragraph are true for this approach, too.

Further, the paper uses simulation to predict microarchitecture behaviour, which is not applicable to architectures like the ColdFire MCF 5307 whose execution depends very much in execution history. Using safe techniques of microarchitecure modelling with provably correct result, however, seems to be possible for their approach as well, since the analysis phases for microarchitecture behaviour prediction and for path analysis are decoupled as in our approach.

6 Conclusion

In this paper, we have presented an extension to the established technique of implicit path enumeration using ILP for WCET prediction. Our approach makes it possible to add a very fine-grained tuning mechanism for analysis precision and speed by arbitrary static execution context distinction. This method of basic block distinction was used already for our microarchitecture behaviour predictions as these use the PAG framework for analysis where the methods are integrated. This paper contributes by providing this for ILP-based path analysis as well.

7 Acknowledgements

I would like to thank Reinhard Wilhelm, Florian Martin and Christian Ferdinand for their support and for fruitful discussions. Thanks are also due to Reinhold Heckmann for proof-reading and for giving many valuable hints.

References

1. R. Arnold, F. Mueller, and D. Whalley. Bounding Worst-Case Instruction Cache Performance. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 1994.
2. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977.
3. J. Engblom. Processor Pipelines and Static Worst-Case Execution Time Analysis. PhD Thesis, Acta Universitatis Upsaliensis, 2002.
4. J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Dec. 2000.
5. C. Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD Thesis, Universität des Saarlandes, 1997.
6. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, 2001.

7. C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, H. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems — The USES Approach. In *Proceedings of Informatik '99 – Arbeitstagung Programmiersprachen*, Paderborn, 1999.
8. Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.
9. Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS)*, 1995.
10. Y.-T. S. Li, S. Malik, and A. Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1995.
11. Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS)*, 1996.
12. S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7), 1995.
13. S.-S. Lim, J. Hee Han, J. Kim, and S. Lyul Min. A Worst Case Timing Analysis Technique for Multiple Issue Machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, 1998.
14. F. Martin. *PAG Reference Manual*. Universität des Saarlandes, 1995.
15. F. Martin. *Generation of Program Analyzers*. PhD thesis, Universität des Saarlandes, 1999.
16. F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In *Proceedings of the International Conference on Compiler Construction (CC'98)*. Springer-Verlag, 1998.
17. Motorola. *Coldfire Microprocessor Family Programmer's Reference Manual*, 1997.
18. F. Mueller, D. B. Whalley, and M. Harmon. Predicting Instruction Cache Behavior. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.
19. G. Ottoson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1997.
20. P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1, 1989.
21. J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1999.
22. M. Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diploma Thesis, Universität des Saarlandes, 1997.
23. F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proceedings of the 4th International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2001)*, Atlanta, Georgia, USA, November 2001.
24. H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Cheju Island, South Korea, 2000.
25. H. Theiling. Generating Decision Trees for Decoding Binaries. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tools for Embedded Systems*, Snowbird, Utah, USA, June 2001.
26. H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.
27. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Seperate Cache and Path Analyses. *Real-Time Systems*, 18(2/3), May 2000.