Decapsulation: Estimating Execution Profiles in Object Bases

Donald Kossmann Michael Steinbrunn Alfons Kemper

RWTH Aachen, Lehrstuhl für Informatik III
Ahornstraße 55
W-5100 Aachen, Germany
{ *kossmann* | *steinbrunn* | *kemper* }*@informatik.rwth-aachen.de*

Abstract

In this paper, we propose a novel method—called *decapsulation*—for *statically* estimating the (running time) cost of object base application programs. The approach is based on extracting weighted inter-object reference chains that are potentially traversed during the execution of the program. These reference chains, called *path expressions* at the syntactical level, are obtained by data flow analysis; the weights, which denote the traversal probability, are derived from selectivity estimates of the branching statements, i.e., the *if*-statements. From the inter-object reference profile thus obtained, a Petri net based simulation of the application program is derived. The Petri net is initialized by the object base characteristics, such as the cardinality of the relevant type extensions, the fan-in and fan-out of the objects, etc. The outcome of the simulation constitutes an estimate of the I/O costs, i.e., the number of (different) objects read by the program.

1 Introduction

Many measures of optimization in database systems depend on reliable cost estimates, i.e., good quality estimation of the execution cost of the particular application. For relational systems the cost estimation for evaluating *declarative* queries is understood quite well—witness the advanced cost models described in, e.g., [SAC+79, Yao79]. However, most object-oriented database applications are not submitted in the form of a declarative query but, rather, as a procedural application program coded in an imperative programming language such as GOM [KMWZ91] or C++ [Str90].

Principally, there are three vastly different possibilities to estimate the execution profile of an application:[1]

1. *user specification*: Let the programmer of an operation submit a (more or less) reliable estimate.

[1] Of course, some hybrid combinations of these approaches are also conceivable.

2. *sampling*: Monitor the execution of the application with a sample of the possible input parameters and, thus, obtain an estimate of the average execution cost.

3. *static analysis*: Analyze the application program statically (without actually executing it) and estimate the execution cost with respect to the current object base characteristics.

In this work we propose a method—called *decapsulation*—which is based on the third possibility, i.e., static application program analysis. The approach is based on extracting weighted inter-object reference chains that are potentially traversed during the execution of the program. These reference chains, called *path expressions* at the syntactical level, are obtained by data flow analysis; the weights, which denote the traversal probability, are derived from selectivity estimates of the program branches, i.e., the conditions of *if*-statements. From the inter-object reference profile thus obtained, a Petri net based simulation of the application program is derived. The Petri net is initialized by the object base characteristics, such as the cardinality of the relevant type extensions, the fan-in and fan-out of the objects, etc. The outcome of the simulation constitutes an estimate of the I/O costs, i.e., the number of different objects read by the program.

We decided to follow the approach of estimating the execution cost based on static analysis because of several considerations:

- Even though a knowledgeable user could possibly submit a reliable cost estimate, this estimate is invalidated by the dynamically changing object base configuration. Furthermore, the system should also support less knowledgeable users for whom optimization has to be transparent in any case.

- Monitoring the object base under (regular) operation imposes a certain system overhead which may not be tolerable.

- Static analysis facilitates the instantaneous optimization as soon as an application is coded. On the other hand, a monitoring based approach has to be delayed until a representative sample has been collected either under regular object base application or under simulated, off-line execution of the applications—which, for this purpose, have to be stripped of any statement possibly modifying the object base.

The remainder of this paper is organized as follows: Section 2 gives a closer view on the optimization techniques that can be supported by decapsulation. Furthermore, our architecture of estimating the execution costs is presented, the assumptions are listed, and related work is discussed. Section 3 outlines the key ideas of decapsulation. A complete definition of decapsulation can be found in [KSK93]. Section 4 describes our cost estimation based on a Petri net simulation. In Section 5 some experiments are presented that investigate the quality of the estimation for realistic applications and typical object bases. Section 6 concludes the paper.

2 Focus

2.1 Applications

The range of optimization techniques that can be supported by decapsulation is diverse. Predicting the access behavior of an operation can be used for *prefetching* [WZ86]. In [KKM90] decapsulation was introduced for the materialization of functions. The method introduced there was refined in [GKKM92a] in order to obtain information for clustering. Furthermore, the use of *access support relations* [KM92] can be planned by the information derived by decapsulation.

In Section 4, the cost of an operation is estimated on the basis of decapsulation. An estimation of cost is indispensable in query optimization and the optimization of Boolean expressions. The idea in optimizing Boolean expressions is to evaluate those terms first for which the ratio between influence on the result and evaluation cost is highest. In [KMS92] a heuristics is developed which requires the cost estimate of the terms, which possibly invoke computationally complex operations.

2.2 Architecture

The estimation of the execution profile devised in this work is carried out in two steps (cf. Figure 1): first, the code of every operation is inspected with the help of the *decapsulation* tool. As a result, decapsulation returns a deterministic finite automaton that accepts those path expressions that are possibly dereferenced by the application.

Given this automaton and the characteristics of the object base, in the second step, the execution of the application is simulated. This simulation is carried out on Petri nets and yields an estimation of the number of objects of every type that are accessed by the application. Alternatively, the cost of an application can be determined by *monitoring*; i.e., executing the application and, thereby, counting the number of objects that are read from disk. The architecture of monitoring is illustrated in dotted lines in Figure 1.

From the point of view of efficiency, estimating the cost of an application by program analysis is far superior to monitoring. Monitoring either requires an extensive "training" phase or is carried out on-line inducing a running time penalty of the application. Although the number of path expressions that are possibly dereferenced can grow exponentially with the number of statements of an operation, decapsulation and the subsequent simulation are carried out within seconds even for extreme cases. On the other hand, we expect that the cost estimation obtained by monitoring can be more precise—if a large enough sample is monitored.

2.3 Assumptions

2.3.1 Programming interface and data model

We assume, that an application consists of a set of operations that are coded in a strongly-typed, imperative programming language, e.g., GOM [KMWZ91]. Each operation is a sequence of assignments, *if*-statements, loops (*while*, *foreach*) and operation calls. In the current version of decapsulation, only side-effect free operations can be analyzed.

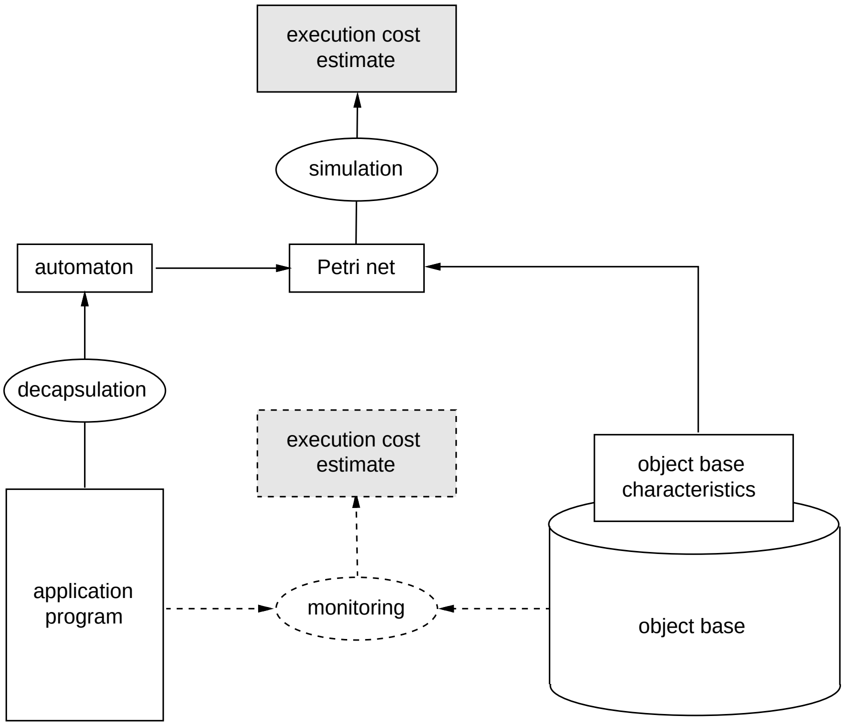


Figure 1: Architecture of estimating the execution costs

Objects are either set or tuple-structured. Tuples are an aggregation of attributes that are accessed by the "." operator. Inter-object reference chains can be represented by so-called *path expressions*; e.g., to compute the value of *Mickey.spouse.age* the object referred to by the path expression *Mickey.spouse* is accessed. As we were interested in an estimation of cost, decapsulation returns those path expressions, e.g., *Mickey.spouse*, that are dereferenced. Due to the strong typing the type of the object every path expression refers to is known at compile time, e.g., *Mickey.friends.spouse* refers to a set of *Persons*.

2.3.2 Infinite buffer model

For the cost estimation, we assume that the size of the system's main memory buffer is potentially infinite. As a consequence, we postulate that no object has to be brought into the buffer twice when running the application and therefore, every object induces to the maximum one unit of cost. Obviously, this assumption is a source of error, as, realistically, the size of the buffer is limited. However, we are mostly concerned with "non-standard" applications, e.g., CAD/CAM, that have a very high locality in their access behavior. To our experience, for this kind of applications, the infinite buffer model applies very well.

2.4 Related Work

For relational database systems, the problem of determining the cost of a *declarative* query (e.g., in SQL) is understood very well [SAC+79, Yao79, Ull88]. Wedekind and Zörntlein [WZ86] already used a method that analyzes application programs to support *prefetching*. However, their work was restricted to the relational model, too.

Basically, decapsulation carries out a data flow analysis [ASU89]. Larus and Hilfinger [LH88] and Weihl [Wei80] also used data flow analysis for the related problem of detecting aliases.

Recently, a number of algorithms to *sample* the state of a database have been published. Lipton and Naughton developed an adaptable sampling algorithm to estimate the size of generalized transitive closures [LN89]. They modified their algorithm in order to determine the selectivity of a Boolean expression [LNS90]. Haas and Swami concentrated on the estimation of the size of a query by the means of sampling [HS92]. However, we know of no work that applies these sampling algorithms to the estimation of the execution cost of an arbitrary object base application program.

3 Decapsulation

3.1 Analysis of Isolated Functions

As we suggested in the preceding section, analyzing a program by means of decapsulation denotes modeling its behavior by way of a finite automaton. In this section, we provide some insight into the techniques utilized.

Decapsulation is grounded firmly on data flow analysis, a well-known approach borrowed from the field of compiler construction (cf., e.g., [ASU89]). Thus, the program is dissected statement by statement while pursuing the flow of information into and out of each one. As we are only concerned about I/O cost, path expressions are the only piece of information we are interested in. For the same reason, only statements where path expressions are involved need to be considered.

3.1.1 Extracting path expressions

Generally, a program's expressions (arithmetic etc.) may, as a special case, contain path expressions. To extract the path expressions from a program, every statement, specifically the statements' expressions, must be checked for path expressions; the set of all the statements' path expressions is used to estimate the execution cost. If a program does not contain any assignments, such a simplish scheme yields indeed the desired result. However, usually programs do contain assignments, so a technique has to be developed that keeps track of the variables' values.

3.1.2 Tracing values

Assignments establish new or modify existing relationships between the target variable and "basic variables," namely function arguments or global variables that constitute entry points into the object base. To get path expressions that originate in basic variables we must keep track of the values, i.e., path expressions assigned.

As a consequence of the assumption of side-effect freeness, only assignments to local variables are possible—everything else would constitute a modification of the object base's state. Thus, an assignment is generally of the shape

$$var_1 := var_2.attr_1.attr_2.\cdots.attr_{n-1}.attr_n$$

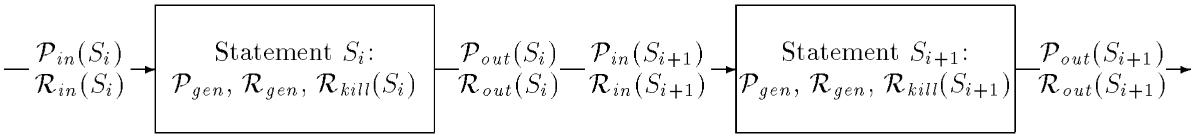


Figure 2: Principle of data flow analysis

where var_1 is a local variable. The information derived from this kind of statement is twofold: first, decapsulation registers that the path expression $var_2.attr_1.attr_2.\cdots.attr_{n-1}$ is dereferenced. Second, from this point on, the variable var_1 can be replaced by its new value, namely $var_2.attr_1.attr_2.\cdots.attr_n$.

Because we want to represent each path expression as an attribute chain originating in basic variables, local variables have to be replaced by their actual values. The tools for accomplishing this task are so-called *rewrite rules*. In the above statement, the resulting rewrite rule, the second piece of information extracted, is $var_1 \rightarrow var_2.attr_1.\cdots.attr_n$

3.1.3 Analyzing the data flow

Every statement of the program yields these two pieces of information: the set of path expressions, denoted as $\mathcal{P}_{gen}$, and the set of rewrite rules, denoted as $\mathcal{R}_{gen}$. This information does not depend on the context and is characteristic for the statement.

However, as it does not suffice to perform an isolated analysis of each statement, a connection to statements executed earlier, if any, must be established. Already existing rewrite rules (denoted by $\mathcal{R}_{in}$) are applied to the newly generated rules and path expressions, yielding the sets $\mathcal{P}_{out}$ and $\mathcal{R}_{out}$, that become, in turn, $\mathcal{P}_{in}$ and $\mathcal{R}_{in}$ for the next statement; this process is illustrated in Figure 2.

Apart from assignments, there are more types of statements in a program that affect the course of execution. Control structures, although contributing nothing by themselves, play a decisive rôle. *if*-statements lead to forking of the control flow, which is handled by considering each branch separately, whereas loops (*while*, *foreach*) are dealt with by viewing the rewrite rules of the body as productions of a left-linear grammar; variables constitute the non-terminal alphabet of the grammar, and variable and attribute *names* the terminal alphabet. The repeated application of productions in this grammar models the iterative characteristic of loops. Because a linear grammar is always regular [Sal73], we can represent it conveniently as the accepting finite automaton [Sal69]. The advantage lies in the canonical specification (as opposed to regular expressions) of path expression sets that facilitates further processing, especially cost estimation. An example that illustrates this approach is provided in Section 3.3.

In Table 1, the dataflow equations for the rewrite rules $\mathcal{R}$ are listed for each statement type. The operator "$\circ$" in the expression $\mathcal{R}_1 \circ \mathcal{R}_2$ denotes the application of rewrite rules in $\mathcal{R}_2$ to rules in $\mathcal{R}_1$ by substituting the variable on the right-hand side of a rule in $\mathcal{R}_1$ by the entire right-hand side of the matching rule in $\mathcal{R}_2$. For instance, the rule $var_1 \rightarrow var_2.attr_1.\cdots.attr_n$ from Section 3.1.2 can be rewritten by a rule with var_2 on its left-hand

Statement S	*Dataflow Equations*
$v := p$	$\mathcal{R}_{gen}(S) = \{(v \to p)\}$ $\mathcal{R}_{kill}(S) = \{\, (v' \to p') \in \mathcal{R}_{in}(S) \mid v' = v \,\}$ $\mathcal{R}_{out}(S) = (\mathcal{R}_{gen}(S) \circ \mathcal{R}_{in}(S)) \cup (\mathcal{R}_{in}(S) \setminus \mathcal{R}_{kill}(S))$
if e *then* S_1 *else* S_2	$\mathcal{R}_{gen}(S) = (\mathcal{R}_{gen}(S_1), \mathcal{R}_{gen}(S_2))$ $\mathcal{R}_{kill}(S) = (\mathcal{R}_{kill}(S_1), \mathcal{R}_{kill}(S_2))$ $\mathcal{R}_{in}(S_1) = \mathcal{R}_{in}(S)$ $\mathcal{R}_{in}(S_2) = \mathcal{R}_{in}(S)$ $\mathcal{R}_{out}(S) = (\mathcal{R}_{out}(S_1), \mathcal{R}_{out}(S_2))$
$S_1; S_2$	$\mathcal{R}_{gen}(S) = (\mathcal{R}_{gen}(S_2) \circ \mathcal{R}_{gen}(S_1)) \cup (\mathcal{R}_{gen}(S_1) \setminus \mathcal{R}_{kill}(S_2))$ $\mathcal{R}_{kill}(S) = \mathcal{R}_{kill}(S_2) \cup \mathcal{R}_{kill}(S_1)$ $\mathcal{R}_{in}(S_1) = \mathcal{R}_{in}(S)$ $\mathcal{R}_{in}(S_2) = \mathcal{R}_{out}(S_1)$ $\mathcal{R}_{out}(S) = \mathcal{R}_{out}(S_2)$
while e *do* S_1	$\mathcal{R}_{gen}(S) = \mathcal{R}_{gen}(S_1^*)$ $\mathcal{R}_{kill}(S) = \{\, (v \to p) \in \mathcal{R}_{in}(S) \mid \exists p' \colon (v \to p') \in \mathcal{R}_{gen}(S) \,\}$ $\mathcal{R}_{in}(S_1^*) = \mathcal{R}_{in}(S)$ $\mathcal{R}_{out}(S) = (\mathcal{R}_{gen}(S) \circ \mathcal{R}_{in}(S)) \cup (\mathcal{R}_{in}(S) \setminus \mathcal{R}_{kill}(S))$
foreach v *in* p *do* S_1	$\mathcal{R}_{gen}(S) = \mathcal{R}_{gen}(S_1^*) \circ \{v \to p@\}$ $\mathcal{R}_{kill}(S) = \{\, (v' \to p') \in \mathcal{R}_{in}(S) \mid \exists p'' \colon (v' \to p'') \in \mathcal{R}_{gen}(S) \,\}$ $\mathcal{R}_{in}(S_1^*) = \mathcal{R}_{in}(S)$ $\mathcal{R}_{out}(S) = (\mathcal{R}_{gen}(S) \circ \mathcal{R}_{in}(S)) \cup (\mathcal{R}_{in}(S) \setminus \mathcal{R}_{kill}(S))$

Table 1: Data flow equations for rewrite rules

side:

$$\{var_1 \to var_2.attr_1.\cdots.attr_n\} \circ \{var_2 \to var_3.attr_a.\cdots.attr_k\} =$$
$$\{var_1 \to var_3.attr_a.\cdots.attr_k.attr_1.\cdots.attr_n\}$$

The set $\mathcal{R}_{kill}$ appearing in Table 1 has not been mentioned so far. It simply contains all rewrite rules that are invalidated ("killed") by the current statement. For instance, an assignment with v_1 on its left-hand side kills all rewrite rules that also have v_1 on their left-hand side.

The data flow equations for extracted path expressions look quite similar to Table 1; however, lack of space does not permit a comprehensive treatment of each aspect of decapsulation. The main purpose of Table 1 is the illustration of the working principles. Section 3.3 contains the complete analysis of a sample program; a more formally oriented description of the underlying concepts can be found in [KSK93].

3.1.4 Assigning weights

Apart from the letter of the terminal alphabet, a *weight* is assigned to every transition of the automaton derived this way. It denotes the probability that the object attribute represented by the terminal letter is dereferenced upon execution of the program. Only the branching probabilities of *if*-statements are estimated; conditions in *while*-statements are not considered, because no attempt is made by the decapsulation module to predict the expected number of iterations of a *while*-loop—due to the fact that probabilities of this kind are likely to change in the course of performing the iterations, this would in most cases be impossible anyway. In our implementation, we used a simple estimation similar to [SAC+79]; in particular, the probability of an equality comparison ($expr_1 = expr_2$) is estimated as 1/10, and ($expr_1 > expr_2$) as 1/3. Checks for NULL references are treated differently: ($expr \neq$ NULL) is assumed to be always true, i.e., the probability is 1, because such checks are usually terminating conditions for loops or recursions, therefore similar considerations as for *while*-loops can be made. In addition, all conditions are presumed to be stochastically independent, so the probability of a conjunction is the product of the individual probabilities.

It does not need to be mentioned that this rather elementary scheme leaves plenty to be desired. However, as it does not affect the principles of decapsulation, we bear with this strategy for its simplicity while keeping in mind that enhanced precision can be accomplished by introducing a more sophisticated method, e.g., one of [Dem80, Chr83, PSC84, MD88, LNS90]. At any rate, the precision of the probability estimation is of little importance if, for instance, the execution costs of the two branches of an *if*-statement are approximately the same. Therefore, the results of the cost estimation are, in spite of the simple probability estimation, quite promising.

3.2 Inter-Function Decapsulation

In the previous section, the principles of decapsulation for a single function have been introduced. Now we shall take a higher point of view for the analysis and investigate the relationships between several functions that invoke each other, including recursive invocations. Because the object base constitutes, in general, a directed graph, recursive functions are frequently encountered, e.g., in implementations of transitive closures. It is therefore important that an analysis tool is capable of handling this situation.

However, it is not necessary to introduce completely new concepts: the basic idea of expressing each traversed path in terms of basic variables does not need to be abandoned.

First of all, the prerequisites for inter-function decapsulation must be stated:

1. Parameter passing is always "by value."

2. The function called must be identifiable by static analysis. This rules out "pointer to function" variables and late binding.

3. There is a single *return*-statement at the end of the function body.

The analysis is carried out in a manner similar to loop analysis. In the first step, the functions involved are treated separately, i.e., an isolated analysis is performed as described in the previous section—this is analogous to the treatment of a loop body. The "*return expr*" statement of a function *op* is considered an assignment $op := expr$ (similar to PASCAL). Furthermore, parameter passing is realized by assigning every actual parameter p_i to the corresponding formal parameter v_i. The rewrite rules that can thus be derived are viewed as the productions of a left-linear grammar, and the further analysis is the same as for loops. The resulting sets of rewrite rules and path expressions, $\mathcal{R}_{global}$ and $\mathcal{P}_{global}$, can be viewed as the "inter-function" analogue for the sets $\mathcal{R}_{out}$ and $\mathcal{P}_{out}$ of the isolated analysis.

```
declare search: Node, int → Node
  code search_code;

define search_code(tree, searchkey) is
begin
  while (tree ≠ NULL and tree.key ≠ searchkey)
  begin
    if (searchkey > tree.key)
          tree := tree.right;
        else
          tree := tree.left;
  end;
  return (tree);
end search;
```

(a)

```
declare sum: Node → int
  code sum_code;

define sum_code(tree) is
  var retval: int;
begin
  if (tree ≠ NULL)
      begin
        retval := tree.entry;
        retval := retval + sum(tree.left);
        retval := retval + sum(tree.right);
      end;
    else
        retval := 0;
  return (retval);
end sum;
```

(b)

Figure 3: Sample programs

3.3 Sample Analysis

To illustrate how decapsulation works, we shall take a closer look at two sample programs. The first program (Figure 3a) searches a key in a tree and returns (a pointer to) the associated node (NULL in case of failure), the second (Figure 3b) computes the sum of all entries in a tree. A tree that may serve as input for both programs is depicted in Figure 6b.

3.3.1 Analysis of "search"

We shall start with the function *search* in Figure 3a. It traverses exactly one path from the root of the argument tree to one of its leaves. Because the cost for non-object operations is presumed negligible, statements of this kind will be omitted in the following discussion. At the beginning, the set of rewrite rules $\mathcal{R}_{in}$ and path expressions $\mathcal{P}_{in}$ is initialized as:

$$\begin{aligned} \mathcal{R}_{in} &= \{\text{tree} \rightarrow \text{tree}\} \\ \mathcal{P}_{in} &= \emptyset \end{aligned}$$

The first statement in *search* is the *while*-loop. According to Section 3.1, the body of the loop has to be treated first. It consists of a single *if*-statement. The rewrite rule for the *then*-branch is $tree \rightarrow tree.right$, for the *else*-branch $tree \rightarrow tree.left$.

Thus, the loop body yields the following two rules (according to Section 3.1.4, the probability of $searchkey > tree.key$ is 1/3):

Rewrite Rule	Probability
tree → tree.right	1/3
tree → tree.left	2/3

The complete grammar $G = (N, \Sigma, \pi, Z)$ that we obtain from these rules is:

$$N = \{N_{tree}, Z\}$$

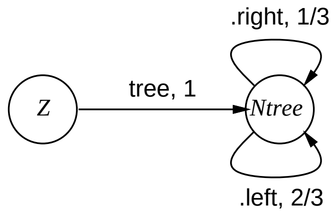


Figure 4: Automaton modeling the *while*-loop of *search*

$$\begin{aligned}\Sigma &= \{\text{tree}, .\text{left}, .\text{right}\}\\ \pi &= \{Z ::= N_{tree}, N_{tree} ::= N_{tree}.\text{left}, N_{tree} ::= N_{tree}.\text{right}\}\end{aligned}$$

The finite automaton that accepts the language generated by G is depicted in Figure 4, together with the probabilities; Z is the initial state, N_{tree} the (sole) final state. The set of path expressions accepted by this automaton is

$$\{\text{tree}, \text{tree.left}, \text{tree.right}, \text{tree.left.left}, \ldots\} = \{\text{tree}(.\text{left} \mid .\text{right})^*\}$$

The question is, how the sets $\mathcal{R}_{out}$ and $\mathcal{P}_{out}$ can be derived from this automaton. For the rewrite rules, the answer is simple: the set consists of rules that have the variables corresponding to the final states as their left-hand sides, and all possible words that lead to this particular final state as their right-hand sides. In our example, there is but one final state, so the rules in $\mathcal{R}_{gen}$ for the *while*-loop are

$$\mathcal{R}_{gen} = \{\text{tree} \rightarrow \text{tree}(.\text{left} \mid .\text{right})^*\}.$$

$\mathcal{R}_{in}$ for the *while*-loop consists of only one rule, namely *tree* $\rightarrow$ *tree*, that does not modify the rules of $\mathcal{R}_{gen}$. Thus, we obtain $\mathcal{R}_{out} = \mathcal{R}_{gen}$.

However, our chief interest focuses on the set of path expressions potentially traversed by *search*; this is the information we ultimately want to extract. Looking closely at the *while*-loop, we notice that some paths have not been considered yet. These paths are found in expressions that do not yield a reference to an object, but a Boolean or integer value. Therefore, they do not influence the computation of the sets of rewrite rules. However, they do influence the computation of the sets of path expressions.

First, the set $\mathcal{P}_{gen}$ for one pass through the body of the *while*-loop has to be determined. In contrast to the computation of $\mathcal{R}_{gen}$, the conditions of the *if*-statement must be taken into account. However, the only "cost-relevant" path expression $\mathcal{P}_{gen}$ of the loop's body is

$$\mathcal{P}_{gen} = \{\text{tree}\}$$

with weight 1. In order to obtain $\mathcal{P}_{out}$, $\mathcal{R}_{gen}$ has to be applied to $\mathcal{P}_{gen}$ and the path expressions extracted from the *while*-condition. The union of the two results constitute $\mathcal{P}_{gen}$ of the entire *while*-loop. Exactly as for the computation of $\mathcal{R}_{out}$, $\mathcal{R}_{in}$ has to be applied to $\mathcal{P}_{gen}$; furthermore, $\mathcal{P}_{in}$ has to be added. In our example, however, this does not make a difference, so we obtain:

$$\mathcal{P}_{out} = \{\text{tree}(.\text{left} \mid .\text{right})^*\}$$

as the result of the analysis of the function *search*.

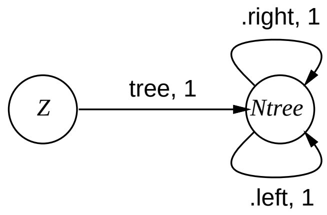


Figure 5: Automaton modeling the recursive implementation of *sum*

3.3.2 Analysis of "sum"

The recursive implementation of the function *sum* (Figure 3b) cannot be resolved by the means used in the previous section. Instead, the concepts of inter-function decapsulation as sketched in Section 3.2 must be applied.

The first step consists of the isolated analysis of the function in exactly the same way as in the previous section. However, as the body of *sum* does not contain any assignment of object references, no rewrite rules are generated. In other words, $\mathcal{R}_{out}$ is the same as the initial value of $\mathcal{R}_{in}$, i.e., $\{tree \rightarrow tree\}$. Additional rewrite rules are generated by modeling the parameter passing mechanism of the recursive invocation. *sum* is called with actual parameter *tree.left* and *tree.right*, generating the rewrite rules

$$\begin{array}{lcl} \text{tree} & \rightarrow & \text{tree.left} \\ \text{tree} & \rightarrow & \text{tree.right} \end{array}$$

that are propagated into $\{tree \rightarrow tree\}$, i.e.,

$$\{\text{tree} \rightarrow \text{tree}\} \circ \{\text{tree} \rightarrow \text{tree.left}, \text{tree} \rightarrow \text{tree.right}\} = \{\text{tree} \rightarrow \text{tree.left}, \text{tree} \rightarrow \text{tree.right}\}$$

yielding the set $\mathcal{R}_{global}$,

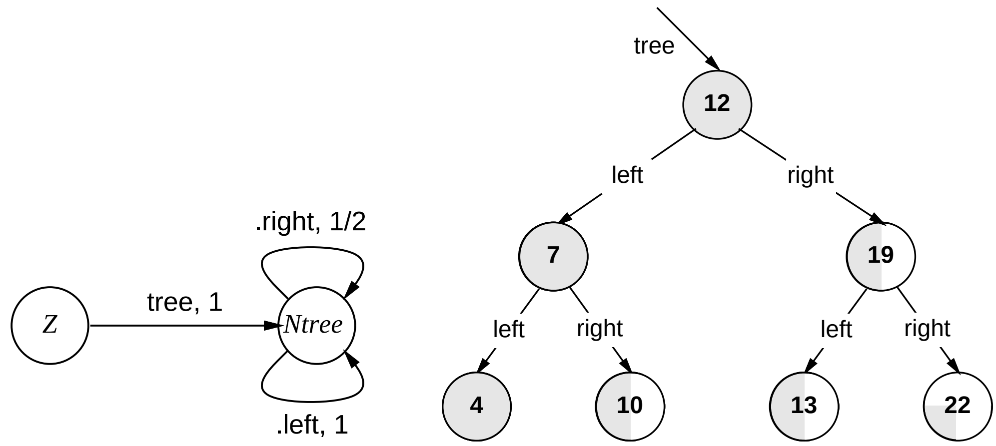
Rewrite Rule	Weight
tree → tree.left	1
tree → tree.right	1

which leads to the automaton in Figure 5. It is quite similar to Figure 4, but the different weights are crucial for the cost estimation.

With this information, the computation of the set of traversed paths $\mathcal{P}_{global}$ for *sum* is straightforward: The rules of $\mathcal{R}_{global}$ must be applied to the set of path expressions that are extracted from *sum*'s body. The set of extracted path expressions for the function body is $\mathcal{P}_{out} = \{tree\}$ with weight 1. Propagating the rules of $\mathcal{R}_{global}$ into $\mathcal{P}_{out}$, the set of paths traversed by an invocation of *sum* can be described by the following regular expression:

$$\mathcal{P}_{global} = \{\text{tree}(.\text{left} \mid .\text{right})^*\}$$

Although the set of traversed paths is described by the same regular expression for both functions, the actual costs are completely different. Therefore, the cost estimator, which is presented in the next section, does not take the regular expression as its input,



(a) (b)

Figure 6: Matching an automaton onto a tree

but the automaton in Figure 4 resp. Figure 5. The different weights of the corresponding transitions, which reflect the different behavior of *search* and *sum*, make the difference in the cost estimation.

4 Estimation of Execution Costs

As stated in Section 2, the information derived by decapsulation can be used to support a great variety of applications. In this section, the estimation of the expected number of objects accessed by an application on the basis of decapsulation is demonstrated. Considering Yao's formula [Yao77] and/or the clustering of objects on pages [GKKM92b], the expected number of objects accessed is a measure for the I/O activity and hence, for the cost (running time) of executing an application.

For example, Figure 6a illustrates the automaton and the weights returned by decapsulation for a modification of function *sum* (cf. Fig. 3b). Once a node is visited, its *left* field is dereferenced with probability 1, whereas its *right* field is only dereferenced with probability 1/2. On an average 4.75 objects are read, when applying this program on the tree illustrated in Figure 6b. In Figure 6b, the probability that a node is read is indicated by the portion of the node that is shaded, e.g., Node 22 is accessed with probability 1/4.

There are several possibilities to estimate the cost of an application given the automaton with weights and the object base. In [GKKM92a], an algorithm is devised that traverses the object base on the basis of the automaton. Starting at every possible entry point of the object base and the initial state of the automaton the algorithm navigates through the object base according to the transitions of the automaton until a NULL reference is detected or up to a certain depth. The algorithm proposed in [GKKM92a] was designed to obtain information for clustering; however, it could easily be adapted to our problem.

A more efficient, albeit less precise, way to estimate the execution cost is to *sample* the object base; i.e., rather than considering every possible entry point, the object base is traversed iteratively beginning at a randomly selected entry point until a terminating condition holds. Then, the mean is computed as an estimation of the running time of the application. Lipton and Naughton suggested a sampling algorithm that would suit our problem quite well [LN89].

Unfortunately, even a sophisticated sampling algorithm will run for hours, when applied to a realistically large database. Since this delay is not tolerable for many applications, we designed a very fast estimation method; thereby allowing an even greater imprecision. The estimation is obtained by *simulating* the execution of the application on Petri nets [Pet77] from the point of view of the main memory buffer. Within this simulation, no I/O is necessary to investigate the state of the object base. No matter how large the object base is and how many states the automaton has, the running time of the simulation will always be less than a second.

4.1 The Parameters of the Simulation

The state of the object base is characterized by the following parameters:

e: number of entry points of the object base.

$card(T)$: the cardinality of the extension of type T.

$fi(T)$: average fan-in of an object of type T; i.e., the number of references that refer to objects of type T divided by $card(T)$.

$fo(T)$: average fan-out of an object of type T; i.e., the number of references emanating form objects of type T.

$card(T.x)$: the average cardinality of all sets referred to by an x field of an object of type T, if x contains a reference to a set-structured object.

In most systems, all entry points to the object base are known; e.g., in our model GOM the entry points are declared as persistent variables [KMWZ91]. Commonly, $card(T)$ is a known quantity, too. On the other hand, the computation of $fi(T)$ and $card(T.x)$ can be expensive. However, they only need to be computed once, or each time the object base has changed significantly, and henceforth, can be stored in the schema. The average fan-out of all instances of type T is determined considering the attributes objects of type T contain.

Obviously, these parameters alone cannot reflect completely the complex structure of a large object base. Exact results will only be achieved, if additional assumptions apply. In particular, the structure of the object base has to be regular, e.g., all instances of the same type T have approximately the same fan-in $fi(T)$. In the current version, anomalies, like objects that are not referred to by any other object, are not dealt with, because we were interested to keep the system as "slim" as possible. However, the simulation system that is based on Petri nets can easily be extended to incorporate additional information. For example, the degree of sharing (the number of objects that are shared) can be taken into account by adjusting the weights of the transitions of the Petri net correspondingly.

4.2 Constructing the Petri Nets

The idea of the simulation is to sort out for every type the number of objects that will be read into the buffer and the number of objects that will not be accessed. Accordingly,

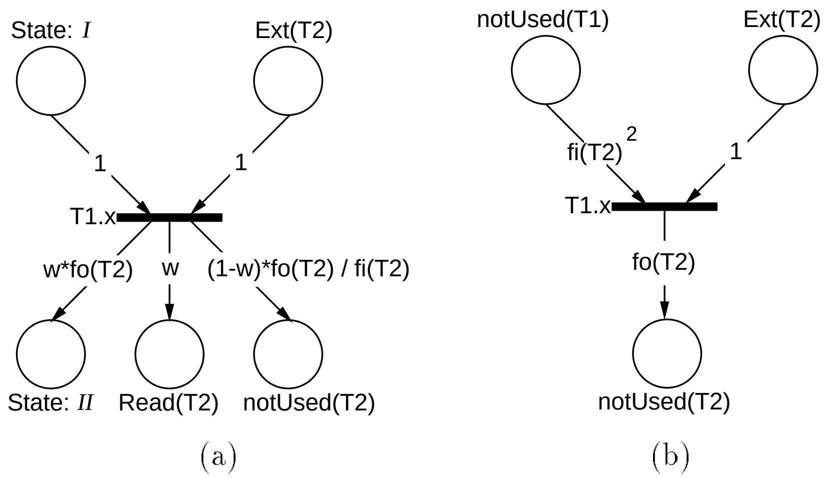


Figure 7: (a) Transition derived from a transition of an automaton, and (b) a *notUsed* transition

three *places* in the Petri net are established for every type T: $Ext(T)$ keeps at every point of the simulation the number of objects that still need to be sorted out, $Read(T)$ represents the number of objects that have been brought into the buffer, and $notUsed(T)$ collects references referring to objects that have not been accessed. Initially, $Ext(T)$ holds $card(T)$ tokens, whereas $Read(T)$ and $notUsed(T)$ are empty.

The behavior of the application is considered by mapping the automaton returned by decapsulation into the Petri net. For every state of the automaton a place is established, and each transition of the automaton is reflected by a transition of the Petri net. The places for the states of the automaton represent the number of references that need to be processed when the execution "reaches" this state. Initially, the place for the initial state of the automaton holds e tokens (representing the e entry points of the object base) and all other places are empty.

Figure 7a illustrates how an arbitrary transition (from State *I* to State *II* with weight w via attribute x of an object of type *T1* referring to an object of type *T2*) of the automaton is reflected in the Petri net. Dereferencing the pointer in the x field is simulated by the *firing* of the transition. A token is removed from the places for State *I* and $Ext(T2)$ denoting that another object is considered. On the other hand, the transition fires w tokens into the $Read(T2)$ place denoting that w objects are read on an average, as the application will dereference the x field with probability w. If a new object of type *T2* is read, $fo(T2)$ *new* references that are stored in the object need to be considered. Consequently, $w * fo(T2)$ tokens are fired into the place representing State *II*. Analogously, if x is set-valued, $card(T1.x)$ tokens are removed from the $Ext(T2)$ place, $w * card(T1.x)$ tokens are fired into the $Read(T2)$ place, and $w * fo(T2) * card(T1.x)$ tokens are fired into the place representing State *II*.

With probability $(1 - w)$ the object that is referred to by the x field is not accessed. The probability that this object may be sorted out into the $notUsed(T2)$ place depends largely on the fan-in of the object. The higher the fan-in (*fi*), the higher the probability that the object can be accessed via another path. We obtained good results by adjusting the weight of the incoming edge of a $notUsed(T)$ place by $1/fi(T)$.

In addition, some objects of the transitive closure of an object that is not accessed will not be read. This is modeled by establishing a *notUsed* transition for every attribute identifier. Figure 7b depicts the structure of the *notUsed* transition for the x field of objects of type *T1*. Again, the probability that an object within the transitive closure is not accessed depends on its fan-in. For this reason, the activity of a *notUsed* transitions is stemmed by $fi(T2)^2$; again, this weighting was obtained heuristically by extensive benchmarking.

To summarize, the Petri net is constructed as follows: the places and transitions are derived from the automaton and the schema of the object base. The weights of the edges are derived from the parameters $fi(T)$, $fo(T)$, and $card(T.x)$, and from the weights returned by decapsulation. The initial marking is determined by the parameters e and $card(T)$.

4.3 Heuristics of the Simulation

The simulation is carried out iteratively obeying the following rules:

1. In each iteration of the simulation every transition will fire that is able to fire. If possible, a transition will also fire several times simultaneously; e.g., if all weights are 1 and all affected places contain two tokens, the transition will fire twice. On the other hand, transitions will also transport fractions of tokens, e.g., if a place only holds half a token. No transition will be preferred. Therefore, it is possible that two transitions *share* a token, when they fire simultaneously.

2. The simulation terminates as soon as no transition is able to fire anymore. Termination is guaranteed, because the number of tokens in the *Ext* places decreases with every iteration.

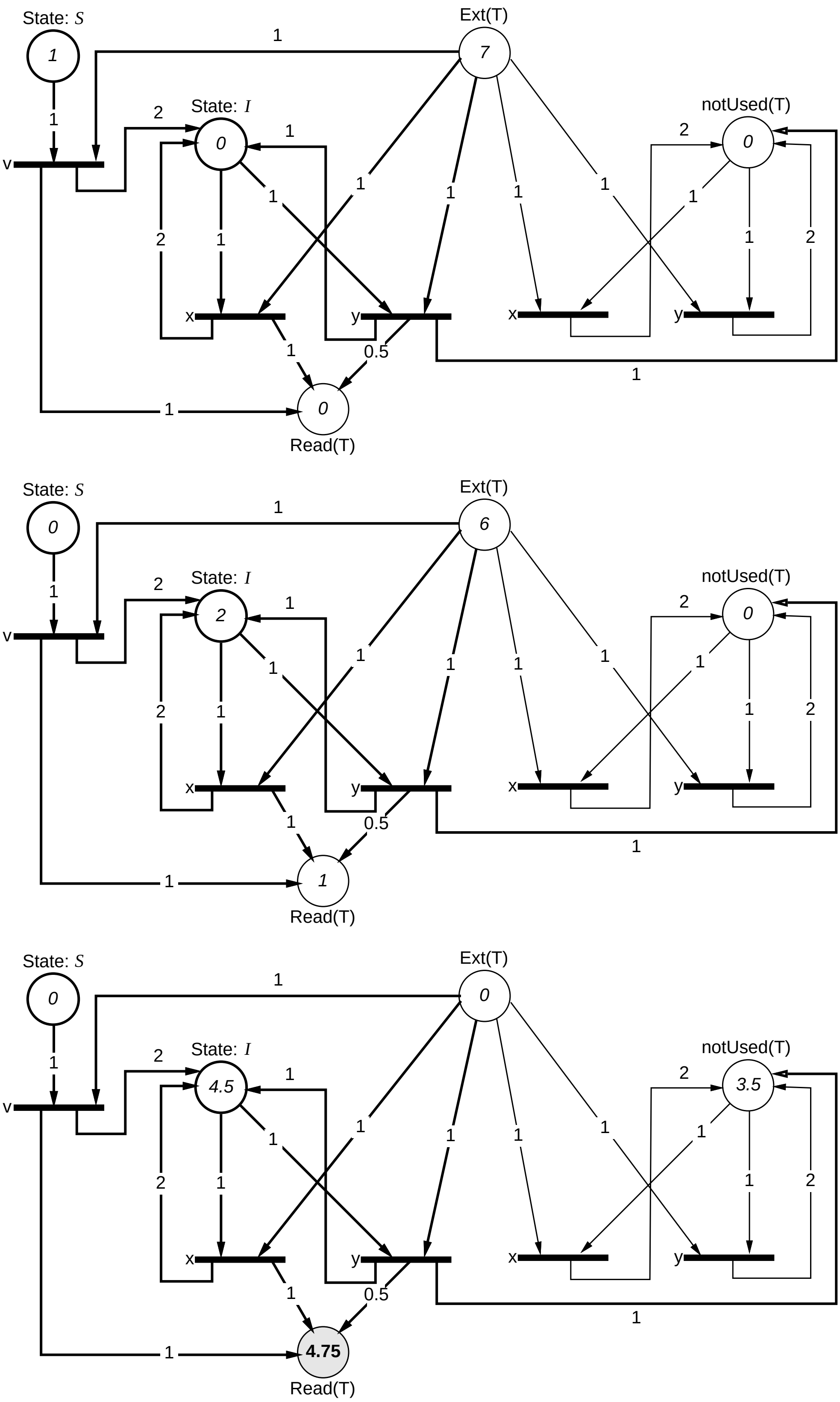
The estimation is obtained by counting the number of tokens in the $Read(T)$ places at the end of the simulation.

Figure 8 illustrates some steps carried out in the simulation of the example of Figure 6. The tree has 7 nodes ($card(T) = 7$). There is only one entry point ($e = 1$), the root of the tree. In trees, no objects are shared, consequently $fi(T) = 1$ holds. Furthermore, the tree is binary inducing $fo(T) = 2$. As expected, the simulation returns 4.75 as the cost for the execution of the application.

5 Qualitative and Quantitative Analysis

The simulation on the basis of decapsulation will estimate the execution profile for some applications precisely; e.g., an application that rotates every *Cube* of a set of *Cubes* whose cardinality is known. Decapsulation will detect that every *Vertex* of every *Cube* is accessed and the simulation will estimate the cost precisely. However, in general, it is not possible to predict the cost of an application exactly. Unfortunately, there even is no upper bound for the error of an estimation based on program analysis that holds for arbitrary object bases, arbitrary applications, and arbitrary user's input.

In the estimation procedure devised in this work there are five sources of error:



(a)

(b)

(c)

Figure 8: Determining the execution cost with a Petri net: (a) initial marking, (b) after the first iteration, and (c) after termination.

1. Decapsulation returns no information about the number of iterations a loop may perform. For example, a loop that navigates through a very large list may halt after a few iterations when an appropriate element is found, or it may scan through the entire list to compute an aggregation. Decapsulation has no means to detect this difference; it is always assumed that the entire list is scanned through.

2. The selectivity of a condition is not always determined precisely.

3. For recursive operations, the set of all path expressions that are possibly dereferenced is context-free and not regular as assumed. The set of path expressions accepted by the automaton returned by decapsulation is a super-set of all the path expressions that are possibly dereferenced, which may be the cause of overestimating the cost of an application.

4. The parameters of the simulation do not characterize the state of an object base precisely.

5. The infinite buffer model can be a source of imprecision.

These flaws become particularly apparent in applications that browse recursively through the object base, e.g., the computation of transitive closures or searching for a particular key. Consequently, the remainder of this section concentrates on presenting experiments for applications of this kind, because programs with straightforward behavior do not pose a problem for decapsulation.

5.1 Searching in Trees

In the first experiment, we measured the cost to search for a key in trees that are organized in four different ways. We chose four different variants to investigate how decapsulation reacts on varying traversal patterns. Figure 9 illustrates how Node 13 is retrieved for each of the four variants; every node that has to be visited to find Node 13 is shaded. If the nodes are unsorted, as in Figure 9a, the tree is traversed depth-first until the key is found. In the worst case, if the key is not stored in the tree, the entire tree needs to be read.

In Figure 9b the tree is organized so that the search may be carried out by *diving* through the tree. First the root is compared with the key to search for. In case of failure, every son of the root is inspected and the subtree is chosen in which the key can possibly be stored. The search terminates as soon as the key is found or a leaf is reached. If *signposts* are kept within every node that indicate in which subtree to continue the search, not every child needs to be visited. This technique is illustrated in Figure 9c. Figure 9d illustrates an index tree. Keys are only stored in the leaves of a tree, and every inner node contains signposts that indicate to which subtree to branch.

For each of the four variants, we estimated the cost of the search by analyzing the corresponding program with the help of the decapsulation tool, and then, carrying out the simulation on the corresponding Petri net. Here, we considered three different instances: trees with degree 2, 3 and 5.[2] The higher the degree of a tree, the lower is the tree's

[2]The degree of a tree corresponds to the fan-out, parameter *fo* of the simulation.

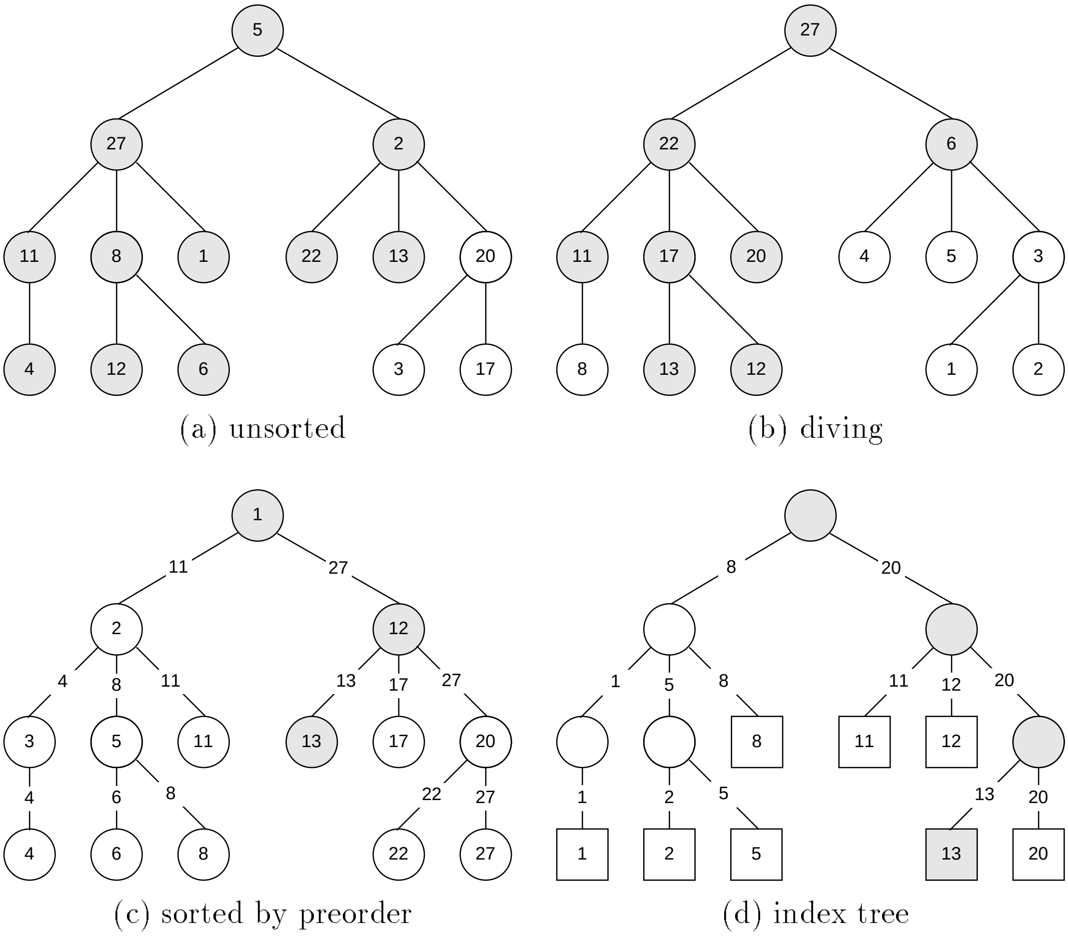


Figure 9: Search trees

height. Except for the search in *unsorted* trees the height is the dominating factor. In all cases, the trees consisted of 10000 nodes and were balanced. On the other hand, we monitored the execution of the corresponding programs; i.e., we carried out the search and counted the number of nodes that were brought into the buffer. We repeated the search with a sample of 10000 random search keys. Beforehand, we found out experimentally that 10000 samples are usually enough to obtain precise and stable results for these experiments. However, no matter how many samples are considered, error can never be precluded when using a random sample.

Figure 10 summarizes the experimental results (a, c, e, f) and the relative error (b, d, f, h) of the simulation under the assumption that the results obtained by (extensive) monitoring are representative. The hit probability is plotted against the x-axis. The hit probability is defined as the fraction of the search keys that are stored in the tree, when building the sample. With increasing hit probability the average cost to search for a key decreases, because the algorithms terminate as soon as the key is found. Only for index trees, the average cost remains invariant with increasing hit probability, as here, execution stops at leaves in any case. The graphs labeled by SIM refer to the estimations obtained by the simulation, whereas those labeled by MON refer to the costs measured by monitoring.

For *diving, preorder search,* and *index tree* the error is quite small (less than 10%).

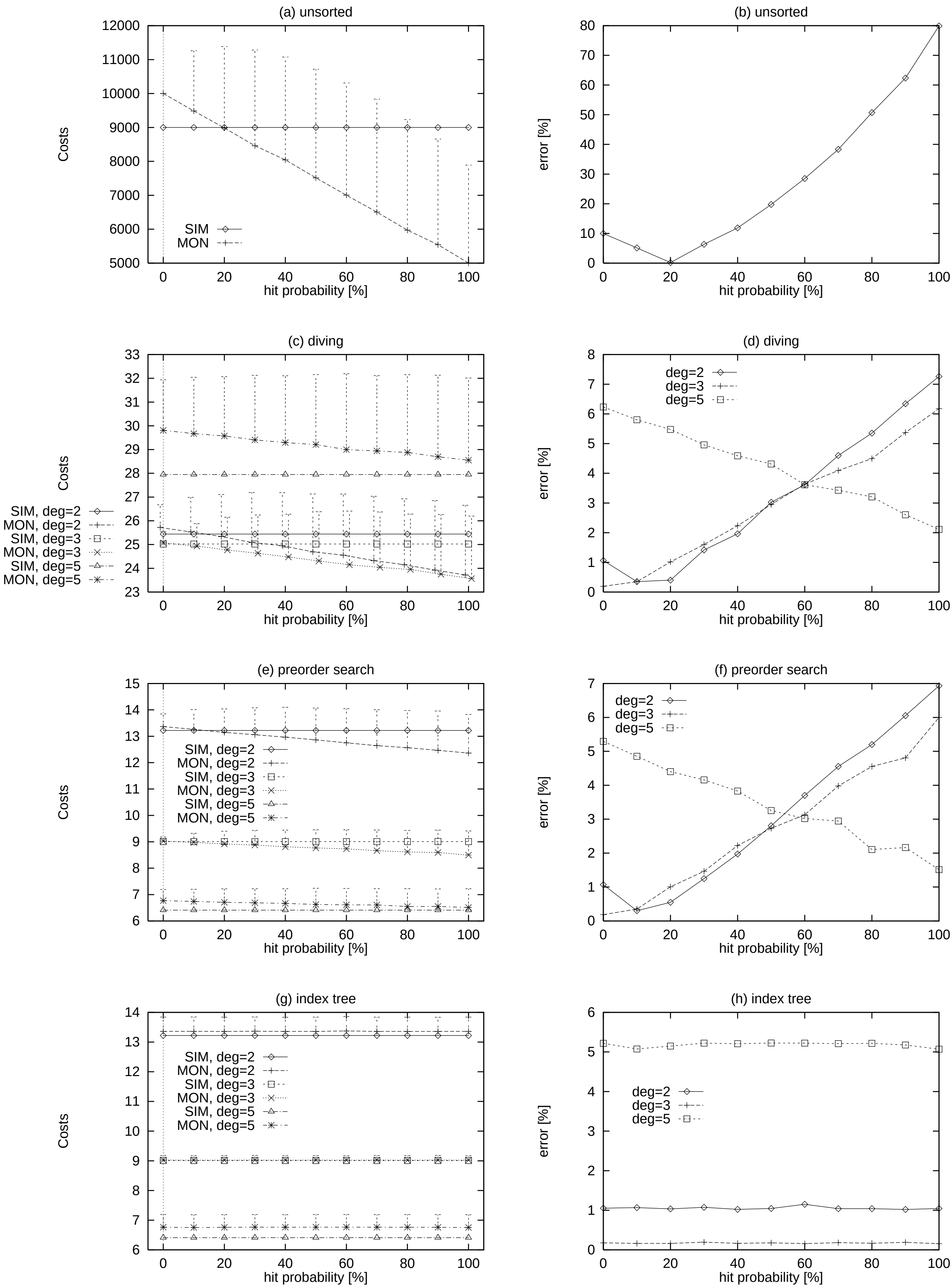


Figure 10: Costs for searching a random key

These strategies turned out to have a very low deviation and the hit probability only has a rather small effect. Consequently, the uncertainty at compile-time is low, and decapsulation returned precise results. On the other hand, the cost of searching in an *unsorted* tree has a very high deviation and is very sensitive with respect to the hit probability. If the key is stored in the tree, half the tree must be read in the average case until the key is found. However, decapsulation assumes that the entire tree is traversed with probability 0.9; thus, it is estimated that on an average 9000 nodes are read.[3]

5.2 Transitive Closure in General Digraphs

In general, the object base is not tree-structured. Consequently, we considered general digraphs in the subsequent experiments. We generated randomly e connected subgraphs with one entry point each. In total, 10000 nodes were created. Then, we added edges—possibly connecting two subgraphs—by selecting two nodes randomly; thereby, cycles in the resulting digraph were not precluded.

As in the preceding experiments, the edges were implemented by inter-object references. Every node contained an attribute *to*, a multi-set of references representing the *outgoing* edges of a node. As no other references were defined, $fi(Node) = card(Node.to) = \#edges/10000$ applied.

We implemented a naive algorithm that computes the transitive closure of a given entry point. The algorithm dereferences all the pointers stored in the *to* fields recursively and marks all the visited nodes. Figures 11a and 11c illustrate the experimental results for varying fi and e, and two different probability distributions that control the generation of the subgraphs. Again, the estimation obtained by carrying out the simulation for the various configurations (SIM) is contrasted to monitoring the execution of the algorithm (MON). Here, we executed and monitored the algorithm for every possible entry point and computed the mean of the resulting costs.

The estimation obtained by the simulation can be seen as a good measure of cost that detects what influence the parameters have on the execution profile. On the other hand, the error can become very high; we observe peaks for the relative error (cf. Fig. 11b), when the simulation overestimates drastically the cost of computing the transitive closure. However, considering the relative error only is not always fair: for a constant absolute error the relative error increases with decreasing cost. This becomes particularly apparent in Figure 11d for $fi = 2$.

6 Conclusion

In this paper, we developed a cost estimation for object-oriented database applications which is based on static program analysis and subsequent simulation. This approach facilitates the very efficient derivation of cost estimates—orders of magnitude faster than monitoring the execution of the application programs on sample input parameters and a representative database configuration.

The quantitative evaluation of our method indicates that the obtained results are quite precise in most cases. However, there are cases—especially when considering ir-

[3] 0.9 is the estimated selectivity of the condition ($tree.key \neq searchkey$).

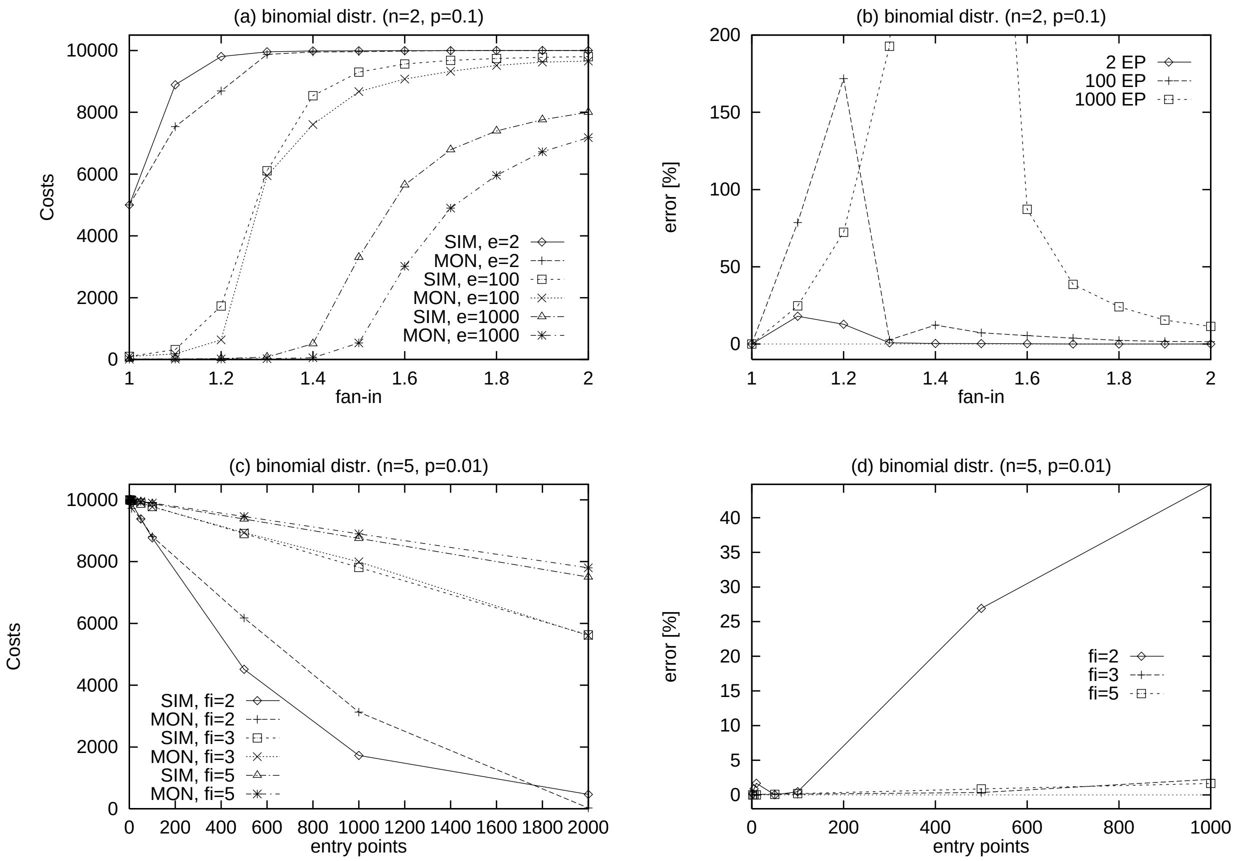


Figure 11: Transitive closure of digraphs

regular (contrived) program and/or object base structures—in which the cost estimate obtained by decapsulation deviates substantially from the actual cost. In future work, we will analyze a large set of realistic programs—obtained from engineering cooperation partners—to evaluate the quality of decapsulation on real-life object base applications.

Acknowledgements

This work was partially carried out within the graduate college "Computer Science and Engineering" at the Technical University (RWTH) Aachen and supported by the German Research Council under contract number Ke 401/6-1. Franz Detro implemented decapsulation and helped carrying out the experiments. We thank Carsten Gerlhof for letting us use his graph generator that is part of the simulation workbench TEXAS. The authors also thank Christoph Kilger and Guido Moerkotte for their contribution in the design of decapsulation.

References

[ASU89] A. V. Aho, R. Sethi, and J. D. Ullman. *Compiler Construction*, volume II. Addison Wesley, Reading, MA, 1989.

[Chr83] S. Christodoulakis. Estimating record selectivities. *Information Systems*, 8(2):105–115, 1983.

[Dem80] R. Demolombe. Estimation of the number of tuples satisfying a query expressed in predicate calculus language. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 55–63, 1980.

[GKKM92a] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Clustering in object bases. Technical Report 6/92, Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Jun 1992.

[GKKM92b] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-based clustering in object bases: From theory to practice. Technical Report 92-34, RWTH Aachen, D-5100 Aachen, Dec 1992.

[HS92] P.J. Haas and A.N. Swami. Sequential sampling procedures for query size estimation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 341–350, Jun 1992.

[KKM90] A. Kemper, C. Kilger, and G. Moerkotte. Materialization of Functions in Object Bases: Design, Realization, and Evaluation. Technical Report 28/90, Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Sept. 1990. To appear in: IEEE Trans. Knowledge and Data Engineering, Apr 1994.

[KM92] A. Kemper and G. Moerkotte. Access Support Relations: an indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

[KMS92] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing Boolean expressions in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 79–90, Vancouver, B.C., Canada, Aug 1992.

[KMWZ91] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. der GI-Fachtagung Datenbanken in Büro, Technik und Wissenschaft (BTW)*, pages 198–217, Kaiserslautern, Mar 1991. Springer-Verlag, Informatik-Fachberichte Nr. 270.

[KSK93] D. Kossmann, M. Steinbrunn, and A. Kemper. Decapsulation: Estimating execution profiles in object bases. Technical report, RWTH Aachen, D-5100 Aachen, 1993.

[LH88] J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 21–34, Atlanta, June 1988.

[LN89] R. Lipton and J. Naughton. Estimating the size of generalized transitive closures. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 165–171, Amsterdam, 89.

[LNS90] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1–11, 1990.

[MD88] M. Muralikrishna and D.J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 28–36, 1988.

[Pet77] J. Peterson. Petri nets. *ACM Computing Surveys*, 9:223–252, 1977.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 256–276, Boston, USA, Jun 84.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, Jun 1979.

[Sal69] A. Salomaa. *Theory of Automata*, volume 100 of *Int. Series of Monographs in Pure and Applied Mathematics*. Pergamom Press, Oxford, 1969.

[Sal73] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[Str90] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1990.

[Ull88] J. D. Ullman. *Principles of Data and Knowledge Bases*, volume II. Computer Science Press, Woodland Hills, CA, 1988.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proc. 7th ACM Sypm. on Principles of Programming Languages*, pages 83–94, San Diego, August 1980.

[WZ86] H. Wedeking and G. Zörntlein. Prefetching in realtime database applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 215–226, Washington, D.C., 1986.

[Yao77] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, April 1977.

[Yao79] S.B. Yao. Optimization of query evaluation algorithms. *ACM Trans. on Database Systems*, 4(2):133–155, 1979.