A Transformational Approach to Parametric Accumulated-cost Static Profiling

R. Haemmerlé[1], P. López-García[1,2], U. Liqat[1],
M. Klemen[1], J.P. Gallagher[1,3], and M.V. Hermenegildo[1,4]

[1] IMDEA Software Institute
[2] Spanish Council for Scientific Research (CSIC)
[3] Roskilde University
[4] Technical University of Madrid (UPM)

Abstract. Traditional static resource analyses estimate the total resource usage of a program, without executing it. In this paper we present a novel resource analysis whose aim is instead the *static profiling of accumulated cost*, i.e., to discover, for selected parts of the program, an estimate or bound of the resource usage accumulated in each of those parts. Traditional resource analyses are parametric in the sense that the results can be functions on input data sizes. Our static profiling is also parametric, i.e., our accumulated cost estimates are also parameterized by input data sizes. Our proposal is based on the concept of cost centers and a program transformation that allows the static inference of functions that return bounds on these accumulated costs depending on input data sizes, for each cost center of interest. Such information is much more useful to the software developer than the traditional resource usage functions, as it allows identifying the parts of a program that should be optimized, because of their greater impact on the total cost of program executions. We also report on our implementation of the proposed technique using the CiaoPP program analysis framework, and provide some experimental results.

1 Introduction and Motivation

The execution of software consumes resources such as time, energy, and memory. The goal of automatic program resource analysis is to infer the resources that a program uses as a function of the size of the input data or other environmental parameters of the program, without actually executing the program. Previous work on this topic, mainly for inferring asymptotic time complexity bounds, goes back to the 1970s. Recent research has adapted and extended these techniques for inferring other resources, including for example energy [15, 14].

In this paper we investigate an extension of this problem which, although based on the same essential techniques, has a different range of applications. Rather than estimating the total resource usage of a program, we wish to perform *static profiling* of its resource usage. This means that we intend to discover, for selected parts of the program, an estimate of the resources used by those parts. As before, the estimates will be parameterised by input sizes. However, these

input sizes will be of the entry predicate/function, unlike the input sizes of the selected parts, as in the standard resource analysis.

There are several motivations for this research. Firstly, a profile of the resource usage of the program can show the developer which parts of the program are the most resource critical. For example, it can expose the cost of functions that are perhaps not particularly resource hungry by themselves but which are called many times. Such parts are natural targets for optimization, since there a small improvement can yield important savings. Secondly, there are cases where the overall resource functions of a program might not be obtainable. This can be for instance because some program parts are too complex for analysis or because the code for some parts is not available and the cost cannot even be reasonably estimated. In this case useful information may still be obtained by excluding such parts from the analysis, obtaining information about the resource usage for the rest of the program. Thirdly, resource usage models (for example Tiwari's energy consumption model [29]) are sometimes based on summing the individual resource usage of basic components of the program. The analysis presented here fits naturally with such models. Finally, in cases where a program has mutually recursive functions/predicates, the standard cost analysis infers similar resource functions for each recursive function. In such cases, a static profile finds precisely the resource functions for each mutually recursive part of the program, and helps identify the parts that are responsible for most of the cost.

The traditional profiling techniques are dynamic (i.e., require executing the program on some particular input) and are based either on code instrumentation, i.e., introducing additional pieces of code in the sections to be measured, or on running a process that performs the profiling together with the measured program. In both cases, the dynamic profiler introduces an overhead in the resource measured that needs to be properly discriminated, which is non trivial. For example, it may be the case that an instruction in the original program has a very different energy consumption in the presence of code added by the profiler just before it. In contrast, the static profiling approach we propose in this paper obtains safe upper and lower bounds on resource consumption, because it is based on the semantics of the program rather than particular executions of it. I.e., the results are valid for all possible program inputs.

Our starting point is the well-developed technique of extracting recurrence relations that express resource usage functions [32, 25, 6, 5, 7, 1]. These are then solved to get a closed-form function expressing the (bounds on) parameterised resource usage. In our work we will make use the CiaoPP program analysis framework, which includes a set of generic resource analyses based on these techniques. In particular, we will use the analysis described in [28]. CiaoPP operates on an intermediate semantic program representation based on Horn Clauses [16], that we will refer to as the "HC IR." By transforming the input language into this intermediate representation, the CiaoPP framework has been shown capable of analyzing imperative programs at the source, bytecode, or binary level with competitive precision and efficiency (see [16, 21, 20, 15, 14] for details).

Our approach to static profiling is based on a transformation that is performed at the level of the CiaoPP Horn Clause-based intermediate representa-

tion. The proposed transformation allows a standard cost analyzer (CiaoPP in our experiments) to statically infer functions that return bounds on *accumulated costs* depending on input data sizes, for a number of predefined program points of interest (predicates in our case), referred to as cost centers. Intuitively, given a program $\mathcal{P}$, the cost accumulated in a given predicate $\mathtt{p} \in \mathcal{P}$ is defined in the context of the execution of a single call to another predicate $\mathtt{q} \in \mathcal{P}$. It expresses the addition of (part of) the resource usages corresponding to the execution of all calls to predicate p generated by a single call to predicate $\mathtt{q} \in \mathcal{P}$.

In the rest of the paper, Section 2 presents informally a general model of (dynamic) profiling and how we turn it into a static version. Section 3 reviews established techniques for cost analysis based on extracting and solving cost relations. Section 4 formalizes our notion of *accumulated cost*. Section 5 describes the implementation of the technique, based on a source-to-source transformation. Section 6 reports some experimental results. In Section 7 we comment on some related work and finally Section 8 concludes, discussing future directions.

2 From Dynamic Profiling to Static Profiling

We start by presenting informally a general model of (dynamic) profiling and how we turn it into a static version. Our model is based on the notion of *cost centers*, inspired from the work of Sansom and Peyton Jones [26] and Morgan and Jarvis [18]. This approach was also applied to Logic Programs and extended to perform run-time checking of non-functional properties in [17]. Intuitively a cost center provides a dynamic scoping mechanism to uniquely attribute the execution costs of a part of the code to an identifier. The scope of the cost center is dynamic in the sense that execution costs of code that are not explicitly associated to a cost center are dynamically attributed to the same cost center as the caller. For a number of languages it is convenient to identify the cost centers with (a subset of) functions, procedures, or predicates. In this paper we follow this path. Alternatively, cost centers can be defined by special scoping constructs [26].

As an example,[5] assume that a programmer wishes to profile a program which uses the following `variance()` function (`variance()` naively computes the variance of an array of integers):

```
1 int  variance(int * arr, int size){
2   int tmp[size], i = size;
3   while(i > 0) {
4     i--;
5     tmp[i] = (arr[i] - mean(arr, size));
6     tmp[i] = tmp[i] * tmp[i]; }
7   return mean(tmp, size);
8 }
```

[5] As mentioned in the introduction, CiaoPP's analyses deal with programs written in such C-like languages (among others) by analyzing corresponding Horn Clause representations.

Assume that `mean()` is a given function that computes the mean of an integer array. First consider that both `mean()` and `variance()` are cost centers. In this case the actual execution costs of the code that appears textually within the `variance()` function will be aggregated at each call to such function and will be attributed to the `variance()` cost center. However the cost of calls to `mean()` –including those made from `variance()`– will not be attributed to `variance()`. Now consider the case where `variance()` is declared a cost center, but `mean()` is not. In this case the execution costs of calls to `mean()` made from the `variance()` function will be also aggregated to those of `variance()` (but not those made from other points in the program).

Returning to the case where both `variance()` and `mean()` are declared as cost centers, assume that the programmer profiles the energy consumption (measured as nano joules, nJ) of a call to the `variance()` function over the array $\{1, 2, 3, 4\}$, on some particular architecture. Assume that the result of the profiler is that 74.7 units of energy are accumulated in the `variance()` cost center and 464.4 units in the `mean()` cost center. Since `mean()` is called 4 times, the cost of a single call to it (with the array above) would be 116.1 nJ (464.4/4). If only `variance()` were declared a cost center, the profiler would have accumulated all the cost in it, i.e., $464.4 + 74.7$ nJ. In such a case, the cost measured by the profiler would be the same as what we call the standard cost of a (single) call to `variance()` with the given array (i.e., 539 nJ).

Since the accumulated value in the `mean()` cost center is much larger than that accumulated in the `variance()` cost center, this indicates that for this particular call most of the energy is consumed inside the `mean()` function, i.e., that this function is responsible for most of the standard cost of the call to `variance()`. This can be a strong indicator that it may be worthwhile to either optimize the body of `mean()` or try to reduce the number of times it is called. Note, however, that with just this data, which come from a run with a particular input, the programmer does not really have any guarantees that the results are representative of the general behavior of the program for all inputs. This problem is usually tackled by repeating the process on a large set of different inputs. This can lead to more indicative results, but still has several drawbacks. First, this process can be very long, because profiling usually introduces additional execution costs, which get multiplied by the number of inputs. Second, and more importantly, even if a large number of inputs is used, this still does not provide a strong guarantee, i.e., there may be some corner case inputs for which the call behaves in a very different way. Finally, the approach does not allow the comparison of the asymptotic cost accumulated in the different cost centers.

To overcome the problems outlined above, we propose to *statically* infer (lower and upper) bounds on the cost accumulated in the cost centers as functions of the sizes of the input data to the profiled call (the call `variance()` in our example). In the example above, the system we have implemented infers (for the resource "energy"[6]) that for a call to `variance()` with a list of size $size$, the costs

[6] Using as back-end analysis the energy analysis of [15, 14] on an XCore XS1 processor with the program compiled by the XMOS xcc compiler without optimization.

accumulated in the `variance` and `mean()` functions are $24.32 + \texttt{size} \times 12.59$ and $23.03 + 17.46 \times \texttt{size}^2 + 40.49 \times \texttt{size}$ energy units (nano joules) respectively. In this case the system infers these expressions for both the upper and lower bounds, which means that they are exact costs. Hence, the programmer does have the guarantee that for all non-trivial calls (i.e., for all calls with non-empty lists) *and for any input data*, the code of `mean()` consumes most of the energy. In this case an obvious improvement can be made, since the call to `mean(arr, size)` can be safely moved outside the loop:

```
1  int  variance(int * arr, int size){
2    int tmp[size];
3    int i = size;
4    int m = mean(arr, size);
5    while(i > 0) {
6      i--;
7      tmp[i] = (arr[i] - m);
8      tmp[i] = tmp[i] * tmp[i];
9    }
10   return mean(tmp, size);
11 }
```

For this version of the program, the system infers that the costs accumulated in the `variance()` and `mean()` functions are $28.18 + \texttt{size} \times 8.73$ and $46.06 + 34.92 \times \texttt{size}$ energy units (nano joules) respectively. For brevity and simplicity we chose a program that is rather naive and where the optimization is obvious (and would in fact be done by some compilers automatically), but the same reasoning applies to more complex cases that are not easy to spot without profiling information. Furthermore, the static profiling functions can also be used for guiding automatic optimization by the compiler.

3 The Classical Cost Relations-based Parametric Static Analysis

The approach to cost analysis based on setting up and solving recurrence equations was proposed in [32] and has been developed significantly in subsequent work. For example, in [25] an automatic upper-bound analysis was presented based on an abstract interpretation of a step-counting version of a functional program, in order to infer both execution time and execution steps. However, size measures could not automatically be inferred and the experimental section showed few details about the practicality of the analysis. In the context of Logic Programming, a semi-automatic analysis was presented in [6, 5] that inferred upper-bounds on the number of execution steps, given as functions on the input data sizes. This work also proposed techniques to address the additional challenges posed by the Logic Programming paradigm, as, for example, dealing with the generation of multiple solutions via backtracking. However, a shortcoming of the approach was its loss in precision in the presence of divide-and-conquer programs in which the sizes of the output arguments of the "divide" predicates

are dependent. This approach was later fully automated (by integrating it into the CiaoPP system and automatically providing *modes and size measures*) and extended to inferring both upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [7, 10]. In addition, [7] introduced the setting up of non-deterministic recurrence relations for the class of divide-and-conquer programs mentioned above, and proposed a technique for computing approximated closed form bound functions for some of them. Such a technique was based on bounding the number of terminal and non-terminal nodes in the set of computation trees corresponding to the evaluation of the non-deterministic recurrence relations, and bounding the cost of such nodes. Non-deterministic recurrence relations were also used and further developed in [1] (named Cost Relations). The approach in [6, 5, 7] was generalized in [22] to infer *user-defined resources* (by using an extension of the Ciao assertion language [11]), and was further improved in [28] by defining the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework [19, 24] of CiaoPP, obtaining features such as *multivariance*, efficient fixpoints, and assertion-based verification and user interaction. A significant additional improvement brought about by [28] is that it is combined with a *sized types* abstract domain, which allows the inference of non-trivial cost bounds when they depend on the sizes of input terms and their subterms at any position and depth. Recently, many other approaches have been proposed for resource analysis [30, 12, 9, 13, 23, 8, 1, 2]. While based on different techniques, all these analyses infer, for all predicates `p` of a given program $\mathcal{P}$, an approximation of the notion of cost that we call the *standard cost* or *single call cost*. Most of them infer an upper bound, while others infer both upper and lower bounds. The following example shows this (for the case of CiaoPP) and also illustrates that this concept of cost may not be directly useful for locating performance bottlenecks.

Example 1. Consider the following implementation of an `eval(E,M,R)` predicate that evaluates modulo $2^{\mathtt{M}}$ a given expression `E` built from additions and multiplications. This implementation assumes that two predicates are given: `add(A,B,M,R)` and `mult(A,B,M,R)`, that respectively add and multiply two infinite precision numbers `A` and `B` modulo $2^{\mathtt{M}}$, and unify the result with `R`.

```
1 eval(const(A),M,R) :- eval_const(A,M,R).
2 eval(A+B,      M,R) :- eval_add(A,B,M,R).
3 eval(A*B,      M,R) :- eval_mult(A,B,M,R).
4
5 eval_const(A,_,R) :- R=A.
6 eval_add(A,B,M,R) :- eval(A,M,RA), eval(B,M,RB), add(RA,RB,M,R).
7 eval_mult(A,B,M,R):- eval(A,M,RA), eval(B,M,RB),mult(RA,RB,M,R).
```

For the sake of simplicity, assume that all the costs are null except those related to the evaluation of `add/4` and `mult/4`. Assume that the cost of the evaluation of `add(A,B,M,R)` is `M` and the cost of the evaluation of `mult(A,B,M,R)` is $\mathtt{M}^2$. Under these assumptions, the standard CiaoPP cost analysis infers that the cost of the evaluation of `eval(E,M,R)` is bounded by $(2^{\mathrm{depth}(\mathtt{E})} - 1) \times (\mathtt{M} + \mathtt{M}^2)$

where depth(E) stands for the depth of the expression E – note that the exact bound is $(2^{\mathrm{depth(E)}} - 1) \times \mathtt{M}^2$. However, such an analysis does not help finding precisely which part of the code is responsible for most of the cost. Indeed since all the predicates (`eval/3`, `eval_add/4`, and `eval_mult/4`) are mutually recursive, the system will infer a similar cost for `eval_add/4` and `eval_mult/4`. Furthermore, those costs will be expressed in terms of different input variables making the actual comparison difficult.

4 Parametric Accumulated-cost Static Profiling

We now formalize the new notion of cost that we propose, the *accumulated cost*, which has been intuitively described in Section 1. As mentioned before, our approach is based on the notion of cost centers: user-defined program points (predicates, in our case) to which execution costs are assigned during the execution of a program. Data about computational events is accumulated by the cost center each time the corresponding program point is reached by the program execution control flow.

We start by presenting a formal *profiled semantics* for Logic Programming. For this purpose we assume given a program $\mathcal{P}$. We also assume that each predicate $\mathtt{p}$ is associated with a cost $\mathrm{cost}_{\mathtt{p}} \in \mathbb{R}$ and that the cost centers are defined as a set $\Diamond$ of predicate symbols. In the following we will use overlined symbols such as $\bar{\mathtt{t}}$, $\bar{\mathtt{x}}$, or $\bar{\mathtt{e}}$ to denote a sequence of terms, variables, or arithmetic expressions.

We define a *predicate call with context* as a tuple of the form $\mathtt{r} : \mathtt{p}(\bar{\mathtt{t}})$, where $\mathtt{r}$, the *context*, is a cost center (i.e., a predicate from $\Diamond$) and $\mathtt{p}(\bar{\mathtt{t}})$ is a predicate call. Then, we define *profiled states* as tuples of the form $\langle \alpha \,;\, \theta \,;\, \kappa \rangle$ where α is a sequence of predicate calls with context, θ is a substitution that maps variables to calling data, and κ, the *cost assignment*, is a family of real numbers indexed by the cost centers $\Diamond$. The *profiled resource semantics* is defined as the smallest relation $\rightarrow_{\mathcal{P}}$ over profiled states satisfying:

$$\frac{\mathtt{q} = \mathrm{update}_{\Diamond}(\mathtt{p}, \mathtt{r}) \quad (\mathtt{p}(\bar{\mathtt{s}}) \ \mathtt{:-}\ \beta) \in \mathcal{P}\rho \qquad \sigma \text{ is an m.g.u. of } \bar{\mathtt{s}} \text{ and } \bar{\mathtt{t}}\theta}{\langle \mathtt{r} : \mathtt{p}(\bar{\mathtt{t}}), \alpha \,;\, \theta \,;\, \kappa \rangle \rightarrow_{\mathcal{P}} \langle \mathtt{q} : \beta, \alpha \,;\, \theta \circ \sigma \,;\, \kappa[\mathtt{q} \mapsto \kappa_{\mathtt{q}} + \mathrm{cost}_{\mathtt{p}}] \rangle}$$

$$\frac{\sigma \text{ is an m.g.u. of } \mathtt{t} \text{ and } [\mathtt{s}\theta]}{\langle \mathtt{r} : (\mathtt{t} \ \mathtt{is}\ \mathtt{s}), \alpha \,;\, \theta \,;\, \kappa \rangle \rightarrow_{\mathcal{P}} \langle \alpha \,;\, \theta \circ \sigma \,;\, \kappa \rangle}$$

where:

- $\mathtt{q} : \beta, \alpha$ is a notation for the sequence $\mathtt{q} : \mathtt{p}_1(\bar{\mathtt{s}}_1), ..., \mathtt{q} : \mathtt{p}_n(\bar{\mathtt{s}}_n), \alpha$, assuming β is the sequence $\mathtt{p}_1(\mathtt{s}_1), ..., \mathtt{p}_n(\mathtt{s}_n)$.
- $[\mathtt{s}]$ stands for the arithmetic evaluation of $\mathtt{s}$ (if $\mathtt{s}$ is not a ground arithmetic expression, then $[\mathtt{s}]$ is not defined, as well as the rule using it),
- ρ stands for a renaming with fresh variables,
- $\kappa[\mathtt{q} \mapsto c]$ is the assignment that maps $\mathtt{p}$ to c if $\mathtt{p} = \mathtt{q}$ or to $\kappa_{\mathtt{p}}$ otherwise, and
- $\mathrm{update}_{\Diamond}(\mathtt{p}, \mathtt{r})$ equals either $\mathtt{p}$ if $\mathtt{p} \in \Diamond$, or $\mathtt{r}$ otherwise.

The first rule can be understood as an extension of SLD resolution with cost. Concretely, the cost $\mathrm{cost}_{\mathtt{p}}$ of the called predicate $\mathtt{p}$ is added to the value of the

current cost center, the cost center being updated beforehand to the current predicate if the latter is in fact a cost center, and left unchanged otherwise. The latter rule characterizes the semantics of the built-in `is/2`, where we assume w.l.o.g. that the operation has no cost. Standard left-to-right evaluation is simply recovered by ignoring the cost assignment together with the calling contexts. In the following section, we will use the notation $(\alpha\,;\theta)$, where α is a sequence of predicate calls and θ a substitution, to denote a standard (non-profiled) LP state.

In the following, we use Π as the set of tuples of terms, and $\mathbb{R}$ to denote the set of real numbers. For any cost center $\mathtt{p} \in \Diamond$, the *profiled resource usage function* is the function $\mathcal{C}^{\mathtt{p}}_{\Diamond} : 2^{\Pi} \to 2^{\mathbb{R}^n}$ defined as:

$$\mathcal{C}^{\mathtt{p}}_{\Diamond}(\bar{T}) = \begin{cases} \left\{\kappa \mid \bar{\mathtt{t}} \in \bar{T} \;\&\; \langle \mathtt{p} : \mathtt{p}(\bar{\mathtt{t}})\,;\epsilon\,;\bar{0}\rangle \to^*_{\mathcal{P}} \langle \Box\,;\theta\,;\kappa\rangle\right\} & \text{if } \mathtt{p}(\bar{\mathtt{t}}) \text{ terminates} \\ & \text{universally } \forall \bar{\mathtt{t}} \in \bar{T} \\ \mathbb{R}^n & \text{otherwise} \end{cases}$$

where $\bar{0}$ stands for the trivial cost assignment that maps any cost center to 0, $\to^*_{\mathcal{P}}$ is the reflexive and transitive closure of $\to_{\mathcal{P}}$, $\Box$ denotes the empty sequence of predicate calls, ϵ is the identity substitution, and n is the number of cost centers. We use the "top" element in $2^{\mathbb{R}^n}$ (i.e., $\mathbb{R}^n$) to denote a "don't know" cost for non-terminating programs, which, for simplicity, are currently not defined in our framework. Note that the cost $\kappa_{\mathtt{p}}$ in an infinite derivation can be (asymptotically) different from $+\infty$ as (1) $\mathtt{p}$ can be the context of only a finite number of the steps involved in an infinite derivation, and (2) because costs of predicates can be zero or negative. The profiled semantics is a natural generalization of the standard resource usage semantics which is able to handle several costs which are accumulated in the cost centers. Indeed the resource usage function inferred by the standard analysis can be understood as the function $\mathcal{C}^{\mathtt{p}} = \mathcal{C}^{\mathtt{p}}_{\{\mathtt{p}\}}$ defined over a unique cost center.

$\mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{T})$ denotes the cost accumulated in $\mathtt{q}$ from the calls $\mathtt{p}(\bar{\mathtt{t}})$ ($\bar{\mathtt{t}} \in \bar{T}$), that is, the union of the i^{th} component of all tuples in $\mathcal{C}^{\mathtt{p}}_{\Diamond}(\bar{T})$ if $\mathtt{q}$ is the i^{th} cost center in $\Diamond$ (formally $\mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{T}) = \left\{\kappa_{\mathtt{q}} \mid \kappa \in \mathcal{C}^{\mathtt{p}}_{\Diamond}(\bar{T})\right\}$). In particular, if $\mathtt{p}(\bar{\mathtt{t}})$ deterministically succeeds (e.g., when it is obtained by translation of some imperative program) the cost accumulated in $\mathtt{q}$ from $\mathtt{p}(\bar{\mathtt{t}})$ is unique, i.e., $\mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\{\bar{\mathtt{t}}\}) = \{c\}$ for some $c \in \mathbb{R}$. In such a case, by a slight abuse of notation, we denote the unique value by $\mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{\mathtt{t}})$.

Example 2. Consider the deterministic program given in example 1. If we profile the program, defining all the predicates of the program as cost centers except `add/4` and `mult/4`, the costs accumulated in `eval_const/3`, `eval_add/4` and `eval_mult/4` for a call of the form `eval(E,M,R)` are respectively bounded by 0, $(0.5 \times 2^{\mathtt{depth(E)}} \times \mathtt{M})$, and $(0.5 \times 2^{\mathtt{depth(E)}} \times \mathtt{M}^2)$. This makes it easier to spot the source of most of the cost, i.e., `eval_mult/4`. Therefore, to improve the efficiency of the whole program, it can be useful to concentrate on this predicate, either by optimizing its implementation or by reducing the number of times it is called.

We write $\mathtt{p} \rightsquigarrow \mathtt{q}$ if $\mathtt{q}$ is reachable from $\mathtt{q}$, that is, if $\mathtt{q}(\bar{\mathtt{t}}) \rightarrow^*_{\mathcal{P}} (\mathtt{p}(\bar{\mathtt{s}}), \alpha)$ for some calling data $\bar{\mathtt{t}}$ and $\bar{\mathtt{s}}$, and some sequence of calls α. Given a set $\Diamond$ of cost centers assigned to a program $\mathcal{P}$ and some predicate $\mathtt{p}$, we define the set of *reachable cost centers* from $\mathtt{p}$ as the sequence $\Diamond_{\mathtt{p}} = \{\mathtt{q} \mid \mathtt{q} \in \Diamond \wedge \mathtt{p} \rightsquigarrow^\star \mathtt{q}\}$.

Theorem 1. *Let $\mathcal{P}$ be a program and $\Diamond \subseteq pred(\mathcal{P})$ a set of cost centers for it. Then, for all $\mathtt{p} \in \Diamond$: for all $\bar{T} \subset \Pi$ it holds that: $\mathcal{C}_{\mathtt{p}}(\bar{T}) = \left\{\sum_{\mathtt{q} \in \Diamond_{\mathtt{p}}} \mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{T})\right\}$. In particular, if $\mathtt{p}(\bar{\mathtt{t}})$ deterministically succeeds $\mathcal{C}_{\mathtt{p}}(\bar{\mathtt{t}}) = \sum_{\mathtt{q} \in \Diamond_{\mathtt{p}}} \mathcal{C}^{\mathtt{p}}_{\mathtt{q}}(\bar{\mathtt{t}})$.*

Note that theorem 1 provides the basis for a compositional and modular definition of the standard (i.e., single call) cost analysis, from the results of the accumulated cost analysis. Note also that (by definition of reachable cost center) $\mathtt{p}$ is always reachable from itself, even though $\mathtt{p}$ does not call itself.

5 Inferring Accumulated Cost via Transformation

As mentioned before, our implementation of the static profiler is based on a source-to-source transformation. In this section we show such a transformation that allows obtaining accumulated cost information for cost centers by performing a sized type analysis in CiaoPP. Basically, the transformation consists of adding *shadow arguments* to each predicate of the Horn clauses that represent the accumulated cost for each cost center.

5.1 The Transformation

In this section we assume there is exactly n cost centers and $\Diamond$ is defined as the family $\{\mathtt{p}_i\}_{i \in 0..n-1}$. The transformation proposed consists of adding $n+1$ shadow arguments to each predicate, such that on success those variables will be assigned to the costs accumulated in the program. There are n shadow arguments for the cost accumulated in the cost centers called by the predicate, and an additional one for the cost associated with the calling context, which is not known statically.

Formally, the transformation is defined by the functions $[\![\cdot]\!]_\Diamond$ and $[\![\cdot]\!]_n$ that respectively translate clauses and goals. The function $[\![\cdot]\!]_n : \mathcal{A}^* \rightarrow (\mathcal{A}^* \times E^{n+1})$ (E is the set of possibly non-ground arithmetic expressions) that translates sequences of atoms is defined recursively on the length of the goal as:

- $[\![\mathtt{q}(\bar{\mathtt{t}}), \alpha]\!]_n = ((\mathtt{q}(\bar{\mathtt{t}}, \bar{\mathtt{x}}), \beta), \bar{\mathtt{x}} + \bar{\mathtt{e}})$ where $(\beta, \bar{\mathtt{e}}) = [\![\alpha]\!]_n$
- $[\![\Box]\!]_n = (\Box, \bar{0})$

where $\bar{\mathtt{x}}$ (resp. $\bar{0}$) stands for a sequence of $(n+1)$ fresh variables (a sequence of $(n+1)$ zeros). On the other hand the function $[\![\cdot]\!]_\Diamond : \mathcal{C} \rightarrow \mathcal{C}$ is defined by cases as follows:

$$[\![\mathtt{q}(\bar{\mathtt{t}}) \texttt{ :- } \alpha]\!]_\Diamond = \begin{cases} (\mathtt{q}(\bar{\mathtt{t}}, \bar{\mathtt{x}}) \texttt{ :- } \beta, \\ \quad \bar{\mathtt{x}} \texttt{ is } \bar{\mathtt{e}}[\bar{\mathtt{e}}_n \leftarrow 0][\bar{\mathtt{e}}_i \leftarrow (\mathrm{cost}_{\mathtt{q}} + \mathtt{e}_i + \mathtt{e}_n)] & \text{if } \mathtt{q} = \mathtt{p}_i \in \Diamond \\ (\mathtt{q}(\bar{\mathtt{t}}, \bar{\mathtt{x}}) \texttt{ :- } \beta, \\ \quad \bar{\mathtt{x}} \texttt{ is } \bar{\mathtt{e}}[\bar{\mathtt{e}}_n \leftarrow (\mathrm{cost}_{\mathtt{q}} + \mathtt{e}_n)] & \text{otherwise} \end{cases}$$

where $(\beta, \bar{e}) = [\![\alpha]\!]_n$, $\bar{x}$ is a sequence of $n+1$ fresh variables, and $\bar{x}$ `is` $\bar{e}$ is a notation for x_0 `is` e_0, ..., x_n `is` e_n (assuming $\bar{x} = (x_0, \ldots, x_n)$ and $\bar{e} = (e_0, \ldots, e_n)$).

The translation of a clause is defined by case on the predicate `q` it defines. Suppose `q` is some cost center $p_i \in \Diamond$. In this case the costs associated with `q` itself (i.e., cost_q) are assigned to the argument corresponding to `q`, namely e_i. Furthermore the costs in evaluating `q` that are not associated to any other cost center (i.e., e_n) are also assigned to e_i. Thus we have $\bar{e}[\bar{e}_n \leftarrow 0][\bar{e}_i \leftarrow (\text{cost}_q + e_i + e_n)]$. On the other hand, if `q` is not a cost center, then the costs associated with `q` are associated to its context, namely e_n, and thus we have $\bar{e}[\bar{e}_n \leftarrow (\text{cost}_q + e_n)]$.

Example 3. We show now the translation of the code corresponding to our running example, given in Example 1, assuming that the cost centers are `eval/3`, `eval_const/4`, `eval_add/4`, and `eval_mult/4`. In the translation the output arguments `Ce`, `Cc`, `Ca`, and `Cm` correspond to the cost accumulated in the respective cost centers. On the other hand, the output `C` is the cost that has not been accumulated in any of the cost centers. Within the translation we leave the actual implementations of `add/4` and `mult/4` unspecified and marked by `(...)`.

```
1  eval(const(A),M,R,Ce,Cc,Ca,Cm,C) :-
2    eval_const(A,M,R,De,Dc,Da,Dm,D),
3    Ce is De+D, Cc is Dc, Ca is Da, Cm is Dm, C is 0.
4  eval(A+B,M,R,Ce,Cc,Ca,Cm,C) :-
5    eval_add(A,B,M,R,De,Dc,Da,Dm,D),
6    Ce is De+D,  Cc is Dc, Ca is Da, Cm is Dm, C is 0.
7  eval(A*B,M,R,Ce,Cc,Ca,Cm,C) :-
8    eval_mult(A,B,M,R,De,Dc,Da,Dm,D),
9    Ce is De+D,  Cc is Dc, Ca is Da, Cm is Dm, C is 0.
10 eval_const(A,_M,R,Ce,Cc,Ca,Cm,C) :- R=A,
11   Ce is 0, Cc is 0, Ca is 0, Cm is 0, C is 0.
12 eval_add(A,B,M,R,Ce,Cc,Ca,Cm,C) :-
13   eval(A,M,RA,De,Dc,Da,Dm,D), eval(B,M,RB,Ee,Ec,Ea,Em,E),
14   add(RA,RB,M,R,Fe,Fc,Fa,Fm,F),
15   Ce is De+Ee+Fe, Cc is Dc+Ec+Fc, Ca is Da+Ea+Fa+D+E+F,
16   Cm is Dm+Em+Fm, C is 0.
17 eval_mult(A,B,M,R,Ce,Cc,Ca,Cm,C) :-
18   eval(A,M,RA,De,Dc,Da,Dm,D), eval(B,M,RB,Ee,Ec,Ea,Em,E),
19   mult(RA,RB,M,R,Fe,Fc,Fa,Fm,F),
20   Ce is De+Ee+Fe, Cc is Dc+Ec+Fc, Ca is Da+Ea+Fa,
21   Cm is Dm+Em+Fm+D+E+F, C is 0.
22 add(RA,RB,M,R,Ce,Cc,Ca,Cm,C) :-
23   (...)
24   Ce is 0, Cc is 0, Ca is M, Cm is 0, C is 0.
25 mult(RA,RB,M,R,Ce,Cc,Ca,Cm,C) :-
26   (...)
27   Ce is 0, Cc is 0, Ca is 0, Cm is M*M, C is 0.
```

The following theorem states that the translation of a given program simulates the original one, while reifying the cost assignment as a first-order argument.

Theorem 2. *Assume a given program $\mathcal{P}$ profiled according n cost centers $\Diamond = \{\mathtt{p}_0, \ldots, \mathtt{p}_{n-1}\}$ and a predicate* `p` *different from* `is`*.*

(Soundness) *If $(\mathtt{p}(\bar{\mathtt{t}}, \bar{\mathtt{x}}) ; \theta) \rightarrow^*_{[\![\mathcal{P}]\!]_\Diamond} (\square ; \sigma)$ (for some sequence of pairwise $\bar{x}$ distinct variables free in $\bar{t}$ and θ) then there exists a derivation of the form $\langle \mathtt{p}_i : \mathtt{p}(\bar{\mathtt{t}}) ; \theta ; \bar{0} \rangle \rightarrow^*_{\mathcal{P}} \langle \square ; \sigma' ; \kappa \rangle$, with $\bar{\mathtt{t}}\sigma' = \bar{\mathtt{t}}\sigma$, $\kappa_{\mathtt{p}_j} = \mathtt{x}_j\sigma$ (for $j \in 1, ..., n-1$ and $j \neq i$), and $\kappa_i = \mathtt{x}_i\sigma + \mathtt{x}_n\sigma$.*

(Completeness) *If $\langle \mathtt{p}_i : \mathtt{p}(\bar{\mathtt{t}}) ; \epsilon ; \bar{0} \rangle \rightarrow^*_{\mathcal{P}} \langle \square ; \theta ; \kappa \rangle$, then there exists a derivation of the form $(\mathtt{p}(\bar{\mathtt{t}}, \bar{\mathtt{x}}) ; \epsilon) \rightarrow^*_{[\![\mathcal{P}]\!]_\Diamond} (\square ; \sigma)$, with $\bar{\mathtt{t}}\theta = \bar{\mathtt{t}}\sigma$, $\kappa_{\mathtt{p}_k} = \mathtt{x}_j\sigma$ (for $j \in 1, ..., n-1$ and $j \neq i$), and $\kappa_i = \mathtt{x}_i\sigma + \mathtt{x}_n\sigma$.*

5.2 Performing the Resource Usage Analysis

The Horn Clause program resulting from the transformation described above, whose predicates are augmented with shadow output arguments representing the accumulated cost for each cost center, is analyzed in order to infer lower and upper bounds on the sizes of such arguments, which actually represent bounds on the respective accumulated costs.

In order to obtain such bounds, we use the size analysis presented in [27, 28], integrated in the CiaoPP system. The goal of this analysis is to infer lower and upper bounds on the sizes of output arguments as a function on the sizes of input arguments. This analysis is based on the abstract interpretation framework present in CiaoPP, and basically infers *sized types* for output arguments. Sized types are representations that incorporate structural (shape) information and allow expressing both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. For a more detailed explanation of this process, we refer the reader to [27].

Continuing with our running example, consider the output argument `Ca`, which represents the accumulated cost of the cost center `eval_add/4` when it is called from `eval/4`. In a preprocessing step, the program is unfolded in order to avoid mutual recursion, which makes the analysis harder. After the unfolding step, the analysis infers types for the predicate arguments by using an existing analysis for regular types [31]. This analysis infers that for a call to a transformed version of `eval/4` (with shadow variables) of the form:

```
eval(Exp,M,R,Ce,Cc,Ca,Cm,C)
```

with `Exp` and `R` bound and the rest of arguments as free variables, then `Ca` gets bound to a number upon success, i.e., a term of type `num`. From the inferred regular type, the analysis derives a *sized type schema*, which is just a sized type with variables in bound positions, along with a set of constraints over those variables.

In this case, the corresponding sized type for `num` is $\mathtt{num}^{(\alpha,\beta)}$, where α and β are variables representing lower and upper bounds on the size of the elements

that belong to such type. The metric we use for the size of a number is its actual value, since `num` is a basic type. For compound types, e.g., lists, trees or arithmetic expressions, we can use several metrics for the size of any term belonging to them, such as the depth of such term (as in our example), or the number of type rule applications needed for the type definition to succeed for such term.

The next step involves setting up recurrence relations between size variables. Thus, for β, that represents the upper bound of the size of `Ca`, we obtain the following equation (where $Size_{arg}^{pred}$ is the size of the argument arg corresponding to predicate $pred$):

$$\beta = Size_{Ca}^{eval}(Size_{exp}, M) = \begin{cases} 2 * Size_{Ca}^{eval}(Size_{exp} - 1, M) + M & \text{if } Size_{exp} > 1 \\ 0 & \text{otherwise} \end{cases}$$

At this point, we have obtained a recurrence relation that represents the size of the output argument. However, such expression is not useful for some applications. One disadvantage of using recurrence relations is that the evaluation of them given concrete input values usually takes longer than the evaluation of an equivalent non-recursive expression. In addition, it is not easy to see the complexity order of a given procedure just by looking at its recurrence relation, and the comparison with other functions is also more difficult. For this reason, the analysis uses a solver for obtaining closed-form representations for recurrence relations. Such closed forms can be either exact solutions or safe overapproximations. In our example, the closed-form version for the recurrence is:

$$\beta = Size_{Ca}^{eval}(Size_{exp}, M) = (2^{Size_{exp}} - 1) * M$$

Assuming that the metric for the size of arithmetic expressions is the depth of the term representing them, we have that $Size_{exp} = depth(exp)$. Thus, we can finally conclude that the accumulated cost of `eval_add/4` when called from `eval/3` (i.e., the size of `Ca` in the transformed version of the program), is given by

$$(2^{depth(exp)} - 1) * M$$

6 Experimental Results

We have performed an experimental evaluation of our techniques with the prototype implementation described in Section 5 over a number of selected benchmarks from [28]. The benchmarks are written directly as Horn Clause programs (in Ciao). In each benchmark, a number of predicates are marked as cost centers. The results are shown in Table 1. Static profiling was performed for each cost center, capturing the accumulated cost with respect to an entry predicate (marked with a *star*, e.g., *appendAll*2*). While in the experiments both upper and lower bounds were inferred, for the sake of brevity we only show upper bound functions. Also, each clause body is assumed to have unitary cost.

Table 1. Experimental results.

Program	Cost-Center Predicate	Accumulated Cost UB	Static vs. Dyn	Standard Cost UB	#Calls
appendAll2	*appendAll2**	b_1	0%	$2b_1b_2b_3+b_1b_2+b_1$	1
	appendAll	b_1b_2	33%	b_1b_2	b_1
	append	$2b_1b_2b_3$	61%	β	$b_1b_2+b_1$
hanoi	*hanoi**	2^v-1	0%	$2^{v+1}-2$	1
	processMove	2^v-1	0%	1	2^v-1
coupled	*coupled**	1	0%	$v+1$	1
	f	$\frac{v}{2}+\frac{(-1)^v}{4}+\frac{3}{4}$	1.2%	v	$\frac{v}{2}-\frac{(-1)^v}{4}+\frac{1}{4}$
	g	$\frac{v}{2}+\frac{(-1)^v}{4}-\frac{1}{4}$	0%	v	$\frac{v}{2}+\frac{(-1)^v}{4}-\frac{1}{4}$
minsort	*minsort**	$\beta+1$	0%	$\frac{(\beta+1)^2}{2}+\frac{\beta+1}{2}$	1
	findmin	$\frac{(\beta+1)^2}{2}+\frac{\beta-1}{2}$	7%	β	$\beta+1$
dyade	*dyade**	β_1	0%	$\beta_1(\beta_2+1)$	1
	mult	$\beta_1\beta_2$	0%	β	β_1
variance_ naive	*variance**	1	0%	$2\beta^2$	1
	sq_diff	$\beta-1$	0%	$2\beta_2\beta_1-2\beta_2$	$\beta-1$
	mean	$2\beta^2-\beta$	0%	$\beta-1$	β
variance	*variance**	1	0%	$5\beta+3$	1
	sq_diff	β	0%	β	β
	mean	$4\beta+2$	0%	$2\beta+1$	2
listfact	*listfact**	β	0%	$\beta(\delta+2)$	1
	fact	$\beta\delta+\beta$	47%	$\delta+1$	β

– $ln^{(\alpha_i,\beta_i)}(n^{(\gamma_i,\delta_i)})$ represents the size of the list of numbers L_i, where β_i and δ_i (resp. α_i and γ_i) denote the upper (resp. lower) bounds on the length of the list and the size of its numbers respectively.
– $lln^{(a_1,b_1)}(lln^{(a_2,b_2)}(ln^{(a_3,b_3)}(n^{(a_4,b_4)})))$ represents the size of the list of lists of lists of numbers similarly.
– $n^{(\mu,v)}$ denotes the size of a number with lower- and upper-bounds μ and v respectively.

Column 1 of Table 1 shows the list of benchmarks while column 2 provides the list of cost centers for each benchmark. Column 3 shows the parametric accumulated cost inferred for each cost center, as a resource usage upper bound function on input data sizes of the entry predicate. Column 4 compares the parametric accumulated cost function of each cost center from column 3 with the results from a dynamic profiling tool [17]. Although the analysis infers upper bounds on the accumulated cost, for some benchmarks these are exact upper bounds (in fact, exact costs) and for others these are correct but relatively imprecise. The imprecision introduced in the benchmarks *listfact* and *appendAll2* is due to the fact that the cost not only depends on the input data sizes but also on the sizes of the sub-terms in the input data, since the analysis statically assumes an upper bound on the sizes of the sub-terms. Note that CiaoPP is the only analysis tool that infers concrete upper bound functions over sized types (costs that depend on the sizes of subterms) [28].

Column 5 shows for comparison the cost inferred by the standard (i.e., non-accumulated) cost analysis [28] for each program and its auxiliary predicates (also marked as cost centers). The comparison of the accumulated and standard cost functions (columns 3 vs. 5) shows the usefulness of our approach: the upper bounds on cost centers display accumulated costs for program parts that were not visible with the standard analysis. For instance, similarly to Example 1, the *coupled* benchmark has two auxiliary mutually recursive predicates f and g that are processing elements of a list alternatively until the list becomes empty. The standard analysis infers almost the same upper bound for both functions due to the mutual recursion, whereas the accumulated cost precisely points out the source of cost in the mutually recursive parts. Similarly, in *hanoi*, although the cost of *processMove* (processing a single *hanoi* move) is unitary, we can see that it is called an exponential number of times. The analysis is providing hints to the programmer about the parts of the program that are most profitable candidates for optimization. Note that the upper bound cost functions inferred by static profiling for each cost center predicate are on the input data sizes of the program (entry predicate), in contrast to the standard analysis where the cost functions are on the input data sizes of the predicate that the cost function corresponds to.

Finally, in column 6 an additional *#Calls* cost is presented, indicating the number of times each predicate is called, as a function of input data sizes of the entry predicate. These cost functions are inferred using the standard analysis by defining explicitly a *#Calls resource* for each cost center predicate. A big complexity order in the number of calls to a predicate (in relation to that of a single call) might give hints to reduce the number of calls to such predicate in order to effectively reduce its impact on the overall cost of the program (i.e., the cost of a call to the entry point). More interestingly, since both the *Accumulated* and *#Calls* costs of a predicate `q` are expressed as functions of input data sizes of the entry predicate, their quotient (Column 3 / Column 6) is meaningful and will give an approximation of the cost of a single call to `q` as a function of the input data sizes *of the entry predicate*. Note that the standard analysis (Column 5) also provides an upper-bound approximation of this cost but as a function of the input data sizes of predicate `q`.

7 Related Work

Static profiling in the context of Worst Case Execution Time (WCET) Analysis of real-time programs is presented in [4]. It proposes an approach to computing worst-case timing information for all code parts of a program using a complementary metric, called *criticality*. Every statement of a real-time program is assigned with a criticality value, expressing how critical the respective code is for the global WCET. Our approach is not limited to WCET, since it is able to obtain results for a general class of *user-defined* resources. Furthermore, our inferred metrics are parametric on the input data sizes of the main program, in contrast to the *criticality* metric, which is a numeric value in the range $[0, 1]$. In

addition, our approach is modular and compositional, able to compute accumulated costs w.r.t. calls originating from different procedures of the program, and not only the main program entry point. In [3] the authors present static profiling techniques to estimate the execution *likelihood* and *frequency* of program points in order to assess whether the cost of certain compile-time optimizations would pay off. To this end, they explore the use of some static analysis techniques for predicting the result of conditional branches, such as assuming uniform distribution over all branches, making heuristic based predictions, and performing value range propagation. In this context, our approach can be used to infer bounds on the number of times a certain program point will be called from a given entry point, as functions on input data sizes, in contrast with a single value representing the execution likelihood or frequency. Besides, since our techniques are supported mainly by the theory of abstract interpretation, the approximations inferred are *correct* by design.

8 Conclusions

In this paper we have presented a novel approach of *static profiling of accumulated cost* that infers upper- and lower-bounds of the resource usage accumulated in particular parts of a program as a functions on the input data sizes of the program. We have constructed a prototype implementation of the proposed approach using the CiaoPP program analysis framework. Preliminary experimental results with the tool support the usefulness of our approach where precise accumulated upper bound cost functions were inferred for parts of the program for which the standard analysis was not able to infer precise information. The upper bound functions inferred by the static profiling were also evaluated against a dynamic profiling tool [17], and showed promising accuracy for the static analysis. However in cases where the cost depended on the sizes of the sub-terms of the input, the upper bound accumulated cost loses precision.

Acknowledgements: This research has received funding from the European Union 7th Framework Program agreement no 318337, ENTRA, Spanish MINECO TIN'12-39391 *StrongSoft* project, and the Madrid M141047003 *N-GREENS* program.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.
2. E. Albert, S. Genaim, and A. N. Masud. More Precise yet Widely Applicable Cost Analysis. In *12th Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *Lecture Notes in Computer Science*, pages 38–53. Springer Verlag, January 2011.
3. C. Boogerd and L. Moonen. On the use of data flow analysis in static profiling. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 79–88, Sept 2008.

4. F. Brandner, S. Hepp, and A. Jordan. Static profiling of the worst-case in real-time programs. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS 2012, pages 101–110, New York, NY, USA, 2012. ACM.
5. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
6. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
7. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.
8. J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *PPDP*, pages 1–12. ACM, 2012.
9. B. Grobauer. Cost recurrences for DML programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 253–264, New York, NY, USA, 2001. ACM.
10. M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
11. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.
12. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62, 2012.
13. A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 331–342, New York, NY, USA, 2002. ACM.
14. U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR. In *Proc. of the Foundational and Practical Aspects of Resource Analysis*, Lecture Notes in Computer Science. Springer, 2015. To Appear.
15. U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers*, volume 8901 of *Lecture Notes in Computer Science*, pages 72–90. Springer, 2014.
16. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in Lecture Notes in Computer Science, pages 154–168. Springer-Verlag, August 2007.
17. E. Mera, T. Trigo, P. López-García, and M. Hermenegildo. Profiling for Run-Time Checking of Computational Properties and Performance Debugging. In *Practical Aspects of Declarative Languages (PADL'11)*, volume 6539 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, January 2011.
18. R. G. Morgan and S. A. Jarvis. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programing*, 8(3):201–237, 1998.

19. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
20. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32, April 2008. Extended Abstract.
21. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 65–82. Elsevier - North Holland, March 2009.
22. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.
23. F. Nielson, H. Nielson, and H. Seidl. Automatic complexity analysis. In *Programming Languages and Systems*, volume 2305 of *Lecture Notes in Computer Science*, pages 243–261. Springer Berlin Heidelberg, 2002.
24. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in Lecture Notes in Computer Science, pages 270–284. Springer-Verlag, September 1996.
25. M. Rosendahl. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156. ACM Press, 1989.
26. Patrick M. Sansom and Simon L. Peyton Jones. Time and Space Profiling for Non-Strict, Higher-Order Functional Languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, pages 355–366, New York, NY, USA, 1995. ACM.
27. A. Serrano, P. Lopez-Garcia, F. Bueno, and M. Hermenegildo. Sized Type Analysis for Logic Programs (technical communication). In T. Swift and E. Lamma, editors, *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, volume 13, pages 1–14. Cambridge U. Press, August 2013.
28. A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754, 2014.
29. V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: a First Step Towards Software Power Minimization. *IEEE Trans. VLSI Syst.*, 2(4):437–445, 1994.
30. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, September 2003.
31. C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.
32. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, September 1975.