Combining Source and Target Level Cost Analyses for OCaml Programs

Stefan K. Muller
Carnegie Mellon University

Jan Hoffmann
Carnegie Mellon University

Abstract

Deriving resource bounds for programs is a well-studied problem in the programming languages community. For compiled languages, there is a tradeoff in deciding when during compilation to perform such a resource analysis. Analyses at the level of machine code can precisely bound the wall-clock execution time of programs, but often require manual annotations describing loop bounds and memory access patterns. Analyses that run on source code can more effectively derive such information from types and other source-level features, but produce abstract bounds that do not directly reflect the execution time of the compiled machine code.

This paper proposes and compares different techniques for combining source-level and target-level resource analyses in the context of the functional programming language OCaml. The source level analysis is performed by Resource Aware ML (RaML), which automatically derives bounds on the costs of source-level operations. By relating these high-level operations to the low-level code, these bounds can be composed with results from target-level tools. We first apply this idea to the OCaml bytecode compiler and derive bounds on the number of virtual machine instructions executed by the compiled code. Next, we target OCaml's native compiler for ARM and combine the analysis with an off-the-shelf worst-case execution time (WCET) tool that provides clock-cycle bounds for basic blocks. In this way, we derive clock-cycle bounds for a specific ARM processor. An experimental evaluation analyzes and compares the performance of the different approaches and shows that combined resource analyses can provide developers with useful performance information.

1 Introduction

The programming languages community has extensively studied the problem of statically analyzing the resource consumption of programs. The developed techniques range from fully automatic techniques based on static analysis and automated recurrence solving [2, 11, 25, 38, 51, 54], to semi-automatic techniques that check user annotated bounds [19, 53], to manual reasoning systems that are integrated with type systems and program logics [15, 20, 21, 40]. Static resource analysis has interesting applications that include prevention of side channels [46], finding performance bugs and algorithmic complexity vulnerabilities [49] and bounding gas usage in smart contracts [24]. More generally, it is an appealing idea to provide programmers with immediate feedback about the efficiency of their code at design time.

When designing a resource analysis for a compiled higher-level language, there is a tension between working on the source code, the target code, or an intermediate representation. Advantages of analyzing the source code include more effective interaction with the programmer and more opportunities for automation since the control flow and type information are readily available. The advantage of analyzing the target code is that analysis results apply to the code that is eventually executed. Many of the tools developed in the programming languages community operate on the source level and derive upper bounds on a high-level notion of cost like number of loop iterations or user-defined cost metrics [25, 40, 53]. In the embedded systems community, the focus is on tools that operate on machine code and derive bounds that apply to concrete hardware [9, 55].

In this paper, we study the integration of source and target level resource analyses for OCaml programs. We build on Resource Aware ML (RaML) [29, 30], a source level resource analysis tool for OCaml programs that is based on *automatic amortized resource analysis (AARA)* [33, 36]. AARA systematically annotates types with potential functions that map values of the type to a non-negative number. A type derivation can be seen as a proof that the initial potential is sufficient to cover the cost of an execution. Advantages of AARA include compositionality and efficient inference of potential functions, and thus resource bounds, using linear programming, even if potential functions are polynomial [28]. RaML can derive bounds for user-defined metrics that assign a constant cost to an evaluation step in the dynamic semantics.

Our approaches to integrating source and target level analyses broadly follow the idea of using RaML to derive resource usage bounds that are parametric in the resource usages of basic blocks, and then composing these results with a lower-level analysis that operates on each basic block. Implementing these approaches in practice requires a technical extension of RaML: we extend RaML to enable bound inference for cost metrics that contain symbolic expressions. Instead of specifying cost 8128 at a certain spot in the program, it is now possible to specify a cost expression such $8128a + 9b$ where a and b are symbolic constants. RaML will then derive a bound that is a function of both the arguments and the constants a and b. In the context of this paper,

PL'19, January 01–03, 2018, New York, NY, USA
2019.

symbolic resource analysis can be used to devise resource metrics that are parametrized by the costs of basic blocks. To this end, we automatically annotate the source program with cost annotations that correspond to beginnings of basic blocks in the compiled code. Each cost annotation is labeled with a fresh symbol that corresponds to the, yet unknown, cost of the corresponding basic block. A simple translation validation procedure ensures that every block has been labeled with at least one cost annotation. At the target level, we can now analyze the cost of individual basic blocks and substitute the results for the corresponding symbol in the high-level bound.

Our third contribution is the implementation of the described technique for the OCaml bytecode and native-code compilers. For the OCaml bytecode compiler, we associate the symbolic constants with the number of bytecode instructions in their respective basic block. In this way, we derive symbolic bounds on the number of bytecode instructions that are executed by a function. For the OCaml native code compiler, we use AbsInt's Worst-case execution time (WCET) analysis tool aiT to derive clock-cycle bounds for each basic block for the ARM Cortex-R5 platform. Together with the source-level bounds, this yields symbolic clock-cycle bounds for the compiled machine code. In many cases, aiT cannot automatically derive loop and recursion bounds. So a final combination of source and target level analysis that we explore is to use the basic block analysis performed by RaML to derive aiT control-flow annotations for specific input sizes.

Our technique for connecting an high-level cost model with compiled code is similar to existing techniques that have been implemented in the context of verified C compilers [5, 15] (see Section 6). The novelty of our work is that we implemented the technique for a functional language and an existing optimizing compiler, support higher-order functions, and combine compilation with AARA, and support OCaml-specific features such as an argument stack for avoiding the creation of function closures.

We have evaluated our techniques on several OCaml programs and found them to be both practical and reasonably precise. For example, our bytecode analysis generates asymptotically tight bounds on instruction counts for all of the example programs, and exact bounds for several of them. In addition, for several of our example programs, the control-flow annotations derived by our analysis result in WCET cycle counts that are identical to results from hand-written annotations. Hand annotations require manual reasoning about the recursive structure of the program (which is labor-intensive and error-prone) in addition to the effort of manually inserting the annotations.

The remainder of the paper is organized as follows. Section 2 gives an overview and description of the symbolic resource analysis technique in RaML. In Section 3, we apply the symbolic analysis to connect source and target level analyses. Section 4 describes the application of the technique to the OCaml bytecode compiler and evaluates its effectiveness with experiments. In Section 5, we study the combination with WCET analysis and the OCaml native compiler and report the findings from the respective experiments. Finally, we discuss related work (Section 6) and conclude.

```
1  let rec fold f b l =
2    match l with
3    | [] → b
4    | x::xs → f (fold f b xs) x
5
6  let countsum1 l =
7    let count = fold (fun c _ →c + 1) 0 in
8    let sum = fold (fun s n →s + n) 0 in
9    (count l, sum l)
10
11 let countsum2 l =
12   fold (fun (count, sum) n →
13         (count + 1, sum + n))
14        (0, 0)
15        l
```

Figure 1. Two implementations of the `countsum` function

2 Symbolic Resource Analysis

The first ingredient for connecting the source-level resource analysis with compiled code is an extension of RaML we call symbolic resource analysis. Before describing the technique, we present an overview of symbolic resource analysis and its applications through an example. Consider the two OCaml functions in Figure 1, defined using the auxiliary function `fold`. Both take as an argument an integer list and return a pair of the count and the sum of the elements. The first function, `countsum1`, makes two passes over the list, counting the elements, then summing them, and finally returns a pair. The second, `countsum2`, computes both results in one pass. RaML allows us to compare the two implementations based on how many list operations they perform by instrumenting `fold` with a "tick" annotation indicating that it performs one list operation (a pattern match).

```
1 let rec fold f b l =
2   (Raml.tick (1.0);
3   match l with
4   | [] → b
5   | x::xs → f (fold f b xs) x)
```

When the code is analyzed with this version of `fold`, RaML correctly reports that `countsum1` has a cost of `2.00 + 2.00*M` where `M` is the length of the list, while `countsum2` has a cost of `1.00 + 1.00*M`, reflecting the fact that the former processes the list twice. This analysis doesn't tell the whole story, though. While `countsum2` performs fewer list operations, it performs more operations on tuples because it

```
let listmatch = Raml.fresh_symb ()
let tuplematch = Raml.fresh_symb ()
let tuplecons = Raml.fresh_symb ()

let rec fold f b l =
  (Raml.sytick(listmatch);
  match l with
  | [] → b
  | x::xs → f (fold f b xs) x)

let countsum1 l =
  let count = fold (fun c _ →c + 1) 0 in
  let sum = fold (fun s n →s + n) 0 in
  (Raml.sytick(tuplecons);
  (count l, sum l))

let countsum2 l =
  fold (fun (count, sum) n →
        Raml.sytick(tuplematch);
        Raml.sytick(tuplecons);
        (count + 1, sum + n))
       (Raml.sytick(tuplecons); (0, 0))
       l
```

Figure 2. The countsum implementations with symbolic resource annotations.

destructs and constructs a pair each time the inner function is called. We could add tick annotations to countsum2 to reflect the cost of the tuple operations, but this would require us to know *a priori* the relative costs of tuple and list operations. In particular, constructing a tuple, which performs an allocation, might be more expensive in practice than pattern matching on the head of a list.

Symbolic resource analysis allows us to analyze both implementations in terms of the three types of operations of interest (list matches, tuple matches, tuple allocations) without specifying concrete costs for them. In the code in Figure 2, we use the new built-in function Raml.fresh_symb to generate three symbols, one for each operation, and Raml.sytick to annotate costs in terms of these symbols. Our extended version of RaML is able to analyze both versions of the code and report that the cost of countsum1 is

```
(2.0listmatch + 1.0tuplecons) + 2.0listmatch*M
```

and the cost of countsum2 is

```
(1.0listmatch + 1.0tuplecons) +
(1.0listmatch + 1.0tuplecons + 1.0tuplematch)*M
```

These cost bounds allow us to profile both implementations at a glance and determine the relevant tradeoffs: countsum1 would be preferred if all three operations are equally expensive, but countsum2 would be preferred if we are primarily concerned about the cost of list operations.

In addition to allowing symbolic resource annotations using Raml.sytick, the symbolic version of RaML allows us to define resource metrics that themselves use symbolic costs. In particular, we define a resource metric "symb" which analyzes unannotated code to produce upper bounds on how many times each type of operation (e.g., pattern match, constructor application) is performed. Note that this analysis is done entirely at the source level and is not to be confused with our analysis for instructions executed at the target level, which we discuss in Section 5. When we run RaML using the "symb" metric on the *unannotated* code of Figure 1, it reports the cost

```
(2Mbase + 4Mapp + 10Mvar + 4Mclos
 + 4Mlet + 2Mmat + Mtupl)
+ (Mbase + 8Mapp + 2Mopld + 2Mopev
   + 17Mvar + 2Mclos + 2Mlet + 2Mmat)*M
```

for countsum1 and the cost

```
(2Mbase + Mapp + 4Mvar + Mclos
 + Mlet + Mmat + Mtupl)
+ (Mbase + 4Mapp + 2Mopld + 2Mopev + 11Mvar
   + Mlet + Mmat + Mtupl + Mtupm)*M
```

for countsum2. In addition to the $M + 1$ additional pattern matches and M fewer tuple allocations and matches, this analysis also alerts us to the $4M + 3$ additional function applications, $6M + 6$ additional variable accesses, $2M + 3$ additional closure allocations and $M + 3$ additional let bindings performed by countsum1.

This overview has given a relatively contrived example of using symbolic analysis for profiling, in the interest of simplicity of presentation. Apart from connecting source and target-level code, we anticipate that symbolic resource analysis can have multiple applications, from profiling the number of times functions are executed to determine bottlenecks, to analyzing how the cost of a program is affected by the load times of certain variables (which could, for instance, be used by a compiler to optimize the layout of data in memory).

2.1 Implementation

We have implemented symbolic analysis in the RaML cost analysis tool for OCaml. We begin with a brief overview of the design of RaML, and then describe two implementations of symbolic analysis.

Existing Design of Resource Aware ML RaML relies on INRIA's OCaml parser and type checker to generate a typed OCaml syntax tree. Since the intermediate languages in INRIA's compiler are untyped and types are crucial to automatically derive bounds that are functions of data structures, RaML directly interfaces with the OCaml source-level syntax tree. The first step in the analysis is a compilation of the OCaml syntax tree to a typed RaML syntax tree in "share-let normal form". In addition to the guarantees of "let normal form", which requires that all intermediate sub-computations

be explicitly sequenced using "let" bindings, share-let normal form requires that each variable be used at most once. Variables that are used twice have to be explicitly duplicated using a "share" construct that is similar to a let binding. This conversion step also includes a number of simplifications, such as removal of nested patterns, as well as the introduction of a refined type derivation that corresponds to the argument stack in RaML's dynamic semantics.

The typed RaML expressions in share-let normal form are subject to the actual resource bound analysis using AARA. First, the analysis is provided with a maximal degree of the potential functions and a resource metric that assigns a constant cost to each step of the source-level operational semantics. Metrics can, for example, count the number of steps taken, the number of allocations, or *ticks* that can be defined by a user using expressions of the form `Raml.tick(n)` for an integer n. The intended meaning of a negative n is that n resources become available at this point.

The type derivation is then annotated with multivariate resource polynomials [30] that define potential functions in the sense of amortized analysis. The multivariate resource polynomials are non-negative linear combinations of base polynomials that are inductively defined over types. Base polynomials for lists include for instance binomial coefficients $\binom{n}{k}$. They play a role that is similar to a basis in linear algebra. The coefficients in the non-negative linear combination of the base polynomials are a priori unknown and type (and bound) inference amounts to choosing these coefficients so that the local constraints in the syntax-directed type rules are satisfied. They ensure that potential is soundly used to cover the resource cost and distributed correctly during construction and destruction of data structures. A main advantage of AARA is that these constraints are linear inequalities even if bounds are non-linear.

To derive the non-negative rational values of the coefficients that correspond to a valid type derivation, the inequalities are solved by Coin-Or's off-the-shelf linear program (LP) solver CLP. If the set of inequalities is infeasible then RaML produces a message that indicates that no bound could be found. If CLP produces a solution, RaML extracts the values of the coefficients that correspond to the initial resource polynomial. This polynomial is then simplified and presented to the user as the worst-case bound for the program.

Symbolic Resource Analysis Interface Adapting Resource Aware ML to allow symbolic resource analysis required extending the RaML parser and runtime with facilities for manipulating symbolic annotations `Raml.sytick(...)`, developing a runtime representation of symbolic costs and extending the analysis engine itself to actually perform the symbolic analysis.

We added the following signatures to the interface for the `Raml` module, against which all RaML code is compiled:

```
type sycost
val fresh_symb : unit →sycost
val (++) : sycost →sycost → sycost
val ( ** ) : float →sycost → sycost
val sytick : sycost →unit
```

At runtime, all of these operations are effectively no-ops, as tick annotations have no runtime behavior. The function signatures involving symbolic costs are designed to allow symbolic RaML programs to parse and typecheck as valid OCaml programs using the unmodified OCaml parser and type-checker. When the typed OCaml syntax tree is being converted into a RaML syntax tree, symbolic cost expressions are handled specially by a function that converts them into an internal representation of symbolic costs used by the analysis engine.

Internally, our implementation represents symbolic costs as sparse lists of coefficient-symbol pairs, sorted by the symbol's unique identifier, plus an additional constant term. In practice, most costs manipulated by the system have few non-zero coefficients, so this representation allows for fast operations on costs, such as addition.

Approach #1: Orthogonal Cost Constraints One viable approach to performing symbolic analysis is to conceptually perform one instance of RaML's standard analysis algorithm for each symbol and then combine the results. This approach is valid as long as all costs are positive (with negative costs, costs of one type, such as `tuplematch` in the example above, could be used to "pay for" costs of another type)[1]. Advantages of this approach are that it is conceptually simple and theoretically sound: the theorems that show the soundness of the RaML analysis [31] can be directly applied to each of the orthogonal analyses, although a formalization of symbolic analysis and this soundness proof is outside the scope of this paper.

A direct realization of the above technique, performing a complete analysis for each symbol, would be relatively simple to implement, but it would perform a great deal of repeated work and scale linearly with the number of symbols. Instead, our implementation performs a number of optimizations that drastically improve scalability to larger numbers of symbols. We run the RaML analysis once on the unmodified program. The costs being manipulated are thus symbolic, rather than concrete. The design of the type annotation and resource analysis components of the existing RaML codebase required relatively few changes in order to use our representation of symbolic costs instead of floating-point numbers. This is because those parts of the code rarely manipulate costs directly; most of this manipulation is done by the constraint generation and solving portions of the system, which we modified much more heavily in ways that

[1] In this paper, we are primarily concerned with execution time; as time is only ever consumed, all costs are positive.

will be described below. In places where costs were manipulated directly, we modified the code to use the appropriate operations on symbolic costs.

The new code for our symbolic analysis sits between the analysis engine and the LP solver. Our code stores constraints generated by the analysis engine, rather than immediately passing them to the solver. Each constraint involving symbolic costs expands to a set of numerical constraints, one for each symbol. In practice, however, most of these constraints are trivial. When the analysis code wishes to invoke the solver, we split each stored constraint into its nontrivial components. We then invoke one copy of the Coin-Or LP solver for each symbol that has nontrivial constraints.

Approach #2: Pure LP Solving In the approach used for symbolic resource analysis in this paper, we extend the linear program (LP) used in the resource analysis with an additional variable for each symbol. When the analysis generates a constraint on a variable in terms of a symbolic cost, e.g., $x_0 + x_1 \leq a$ where x_0 and x_1 are normal variables of the analysis and a is a symbol, we rewrite this constraint as $x_0 + x_1 - x_a \leq 0$ where x_a is the LP variable added for symbol a. It is a pleasant consequence of the constraints generated by RaML's analysis that the constraints always remain linear under this transformation (e.g., RaML does not generate symbolic constraints of the form $x_0 \leq a \cdot x_1$, which could not be rewritten into a linear constraint in terms of x_0, x_1 and x_a).

The resulting LP appropriately reflects the constraints on each analysis variable in terms of the symbols. However, this format is not conducive to optimizing for a particular analysis variable and deriving the resulting objective value in terms of the symbols (which is how we would, for example, derive the symbolic bounds given in the `countsum` example above). There are two ways to derive such a result. If concrete values for the symbols are available (for example, if we have in mind particular costs of each operation and would like to substitute these in), we add additional constraints to the LP of the form $x_a = v_a$ where v_a is the concrete value for symbol a [2]. If values for the symbols are unavailable and we need an abstract result in terms of the symbols, we can derive one as follows. First, we add constraints $x_s = 0$ for all symbols and solve to get the constant component of the cost. Next, we remove these constraints and add the constraints $x_a = 1$ and $x_b = 0$ for all symbols $b \neq a$. Subtracting the constant component from the resulting solution gives the coefficient of symbol a in the cost. We then repeat this step for each symbol to derive all of the coefficients.

[2]The astute reader may notice that, in this case, we could also simply substitute the concrete values for the symbols before performing the analysis. However, in many cases, we wish to consider different possible values for the symbols. Substituting before performing the analysis would require us to redo the analysis, which can be quite costly, for each set of values. Substituting after the analysis only requires re-solving the LP with the new constraints, which can be fast with modern LP solvers.

3 Basic Block Execution Bounds

In the remainder of the paper, we focus on a particular application of symbolic resource analysis: deriving the number of times each basic block of an OCaml program is executed. In this section, we describe how we perform this analysis and ensure, in a compiler-independent fashion, that the basic blocks of the compiled code can be matched up to the bounds in the analysis. The following two sections combine these bounds with low-level information about each basic block to derive resource usage bounds on the compiled code using what we refer to as *low-level* resource metrics.

The essence of the basic block analysis is examining an OCaml program to determine which subexpressions will correspond to basic blocks of the compiled code, creating a unique symbol for each basic block, and inserting calls to `Raml.sytick` with the appropriate symbols such that each basic block performs at least one "tick" of its symbol. The essence of matching the basic blocks used in the analysis to the basic blocks of the compiled code is inserting markers before compilation which are preserved through compilation and can be easily matched to the symbols used by the analysis for each basic block. In the remainder of this section, we detail the pipeline we have implemented to perform both of these functions.

When RaML is invoked on an input program, we run a program transformation to annotate each basic block of the input program with a unique annotation, in our case a call, using OCaml's foreign function interface (FFI), to a C primitive `startbb` using as an argument a unique integer representing the basic block. For this transformation, we made use of the OCaml distribution's convenient functionality for implementing abstract syntax tree rewriters [42, Ch. 27.1]. We insert annotations at the starts of function and loop bodies, conditional and `match` branches, and so forth. In addition, we insert annotations after returns from function calls and at the join points of conditionals. In order to ensure that the value of an expression remains unchanged, we must occasionally introduce additional bindings to temporarily store the result of a computation and then return it after the annotation. For example, we would transform a function application as follows:

```
f x1 ... xn
```
becomes
```
let y = f x1 ... xn in startbb k; y
```

where `y` is a new compiler-generated variable and `k` is a fresh ID for the basic block. This transformation introduces additional overhead for the binding and also hinders some optimizations performed by the compiler, so whenever possible, we attempt to avoid performing it and instead add the call to `startbb` later in the basic block.

After this transformation, RaML compiles the resulting code using the OCaml compiler appropriate to the analysis we are performing (in Section 4, we consider the bytecode

compiler `ocamlc` and in Section 5, we consider the native compiler `ocamlopt`). The resulting code (bytecode or binary) is then analyzed separately to obtain appropriate costs for each basic block; we discuss these analyses in the coming sections. Because C primitives are opaque to the OCaml compiler, these annotations are preserved through compilation; the compiled code will include an annotation of each basic block with a number that corresponds to the symbol used for analysis. We then take the transformed source program and convert the FFI calls to symbolic ticks with a symbol corresponding to the number of the basic block, e.g., `startbb 42` becomes `Raml.sytick bb42`. We then run the symbolic resource analysis pipeline on this program to produce a bound on the resource usage of the program in terms of the resource usage of each basic block or, equivalently, an upper bound on the number of times each basic block is executed. Figure 3 illustrates the entire pipeline, from parsing the source program, through the basic block annotations, to compilation and basic block analysis. Dashed lines demarcate the components that are part of the OCaml compiler and RaML, as well as the end-to-end automated bytecode analysis tool we discuss in the next section.

In the remainder of the paper, we explore various ways to combine the resulting analysis with information about each basic block (derived from a lower-level resource analysis performed on the compiled, annotated code) to predict resource usage properties of the whole program. The soundness of these analyses makes some assumptions about the compiler, for example that it does not introduce additional basic blocks not present in the annotated source code. In the remainder of this section, we formalize the assumptions and discuss how reasonable they are in the context of real-world compilers.

3.1 Soundness Assumptions

Formalization of Assumptions We formalize several assumptions about a compiler by assuming evaluation models for both source and target code, and modeling a compiler as an abstract transformation between source and target programs. We relate the source- and target-level evaluations using *traces*, which record the operations performed using a set of *labels*. Assume we have a set of labels *Labels* containing, at a minimum, $\{\text{jump}\} \cup \{\text{tick}(s) \mid s \in \mathcal{S}\}$, where jump records a jump instruction and tick(s) records a tick annotation of a symbol s drawn from a global set $\mathcal{S}$ of symbols. A trace is a sequence of labels $l_1 :: \ldots :: l_{n-1}$. We use metavariables of the form l for labels and T for traces.

We write $e \Downarrow T$ to indicate that the source expression e evaluates, under an unspecified semantics, with a trace T. We model the evaluation of compiled code using an abstract machine model. A *machine* consists of a set of machine states $\mathcal{D}$ together with a labeled transition relation $\mapsto$: $\mathcal{D} \times \mathcal{D} \times \mathit{Labels}$. An execution of a program on a machine consists of a sequence of transitions $D_1 \overset{l_1}{\mapsto} \ldots \overset{l_{n-1}}{\mapsto} D_n$ where $D_1, \ldots, D_n \in \mathcal{D}$ and $l_1, \ldots, l_{n-1} \in \mathit{Labels}$. For such an execution, we can take the sequence of labels $T = l_1 :: \ldots :: l_{n-1}$ to be a trace. We will write a machine definition as a triple $(\mathcal{D}, \mathit{Labels}, \mapsto)$ of its state set, its label set and its transition relation. Given a machine $(\mathcal{D}, \mathit{Labels}, \mapsto)$, we model a compiler as a function $[\![e]\!] : \mathcal{D}$ from source expressions to machine states.

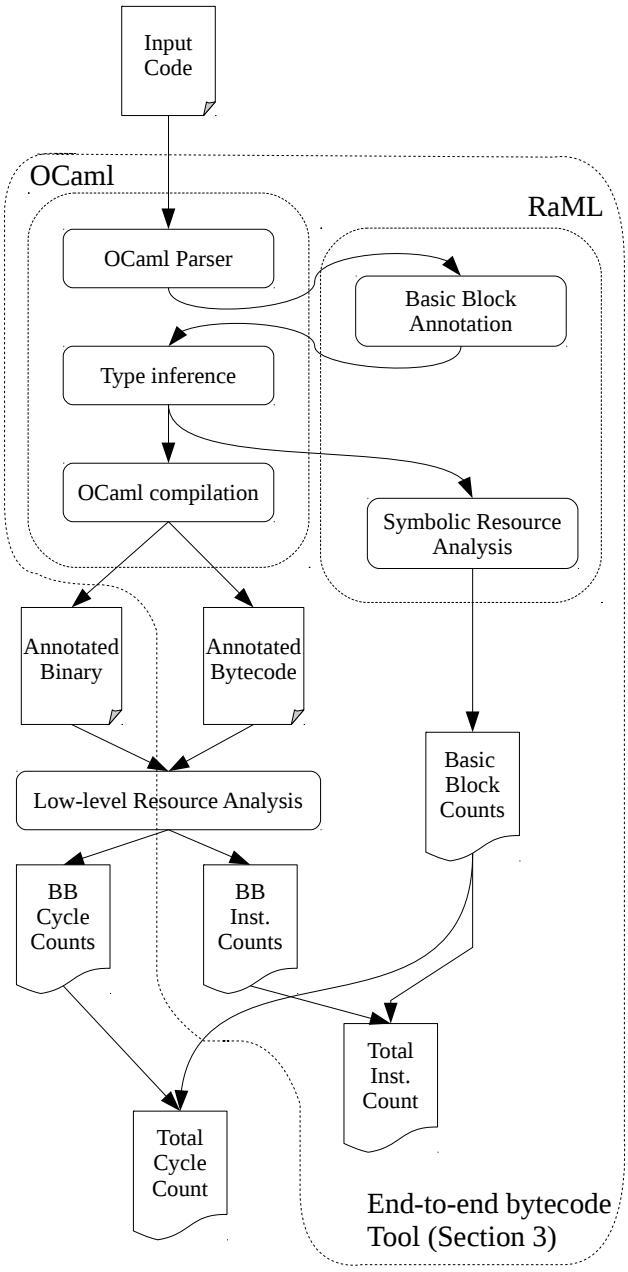

Figure 3. A diagram of the analysis pipeline showing the basic block analysis as well as the low-level analyses of Section 4 (inside our tool, on OCaml bytecode) and Section 5 (external to our tool, on native ARM code).

Let $e \rightsquigarrow e' \mid \mathcal{S}$ denote that e' is the source program obtained by annotating the basic blocks of e using the procedure described in this section, where $\mathcal{S}$ is the set of basic block symbols used in the process. The correctness of our annotation procedure implies that if $e \rightsquigarrow e' \mid \mathcal{S}$ and $e' \Downarrow T$, then $T = T_0 :: \mathsf{jump} :: \ldots :: T_{n-1} :: \mathsf{jump} :: T_n$ where $T_0, \ldots, T_n$ contain no jump instructions and each T_i contains at least one tick instruction. In other words, every basic block of the program contains an annotation. Note that we do not assume here that the basic blocks of source-level program evaluation correspond to basic blocks of the compiled program; this will be explicitly formalized as a compiler assumption below.

Finally, we assume a *resource metric* $M : \mathit{Labels} \to \mathit{Costs}$ that records the cost of an operation. Resource metrics extend naturally to traces: $K(l_1 :: \ldots :: l_n) = \sum_{i=1}^{n} K(l_i)$.

Using the above definitions, we formalize the compiler assumptions made by our analyses. We first assume that, for any expression e, the trace of an execution of $[\![e]\!]$ can be matched up to the execution trace of e so that tick annotations instructions in the source-level execution correspond one-or-more-to-one to tick annotations in the low-level execution. This implies that each basic block is executed at most as many times in the compiled program as in the source program.

Definition 1. A compiler $[\![\cdot]\!]$ is *annotation-preserving* if for any expression e, if

$$e \Downarrow T_1 :: \mathsf{tick}(s_1) :: T_2 :: \ldots :: T_{n-1} :: \mathsf{tick}(s_n) :: T_n$$

then

$$[\![e]\!] \overset{l_1}{\mapsto} D_1 \overset{l_2}{\mapsto} D_2 \mapsto^* D_{m-1} \overset{l_m}{\mapsto} D_m$$

and

$$\begin{aligned} l_1 :: \ldots :: l_m \quad = \quad & T'_1 :: \mathsf{tick}(s_1) :: T'_2 \\ & :: \ldots :: T'_{n-1} :: \mathsf{tick}(s_n) :: T'_n \end{aligned}$$

where $T'_1, \ldots, T'_n$ contain no ticks of symbols in $\{s_1, \ldots, s_n\}$.

A compiler is *block-preserving* if, for *any* expression e the trace of an execution of $[\![e]\!]$ can be matched up to the execution trace of e so that jump instructions *and* tick annotations instructions in the source-level execution correspond one-to-one to jump instructions and tick annotations in the low-level execution. Basic blocks are defined by the locations of jumps and annotated by ticks, so this implies that the basic block structure is the same between the two executions and that the tick annotations are preserved by the compiler.

Definition 2. A compiler $[\![\cdot]\!]$ is *block-preserving* if for any expression e, if

$$\begin{aligned} e \Downarrow \quad & T_1 :: \mathsf{tick}(s_1) :: T_2 :: \mathsf{jump} :: \\ & \ldots :: \mathsf{jump} :: T_{2n-1} :: \mathsf{tick}(s_n) :: T_{2n} \end{aligned}$$

where $T_1, \ldots, T_{2n}$ contain no jump instructions then

$$[\![e]\!] \overset{l_1}{\mapsto} D_1 \overset{l_2}{\mapsto} D_2 \mapsto^* D_{m-1} \overset{l_m}{\mapsto} D_m$$

and

$$\begin{aligned} l_1 :: \ldots :: l_m \quad = \quad & T'_1 :: \mathsf{tick}(s_1) :: T'_2 :: \mathsf{jump} \\ & :: \ldots :: \mathsf{jump} :: T'_{2n-1} :: \mathsf{tick}(s_n) :: T'_{2n} \end{aligned}$$

where $T'_1, \ldots, T'_{2n}$ contain no jump instructions.

We also require that the compiler is *bounded*, meaning that there is a finite bound on how many steps the execution of a single basic block is allowed to take. This does *not* require that a compiled program terminates—indeed, if an expression e is non-terminating then under any block-preserving compiler, $[\![e]\!]$ is required to not terminate. The boundedness requirement simply means that any unbounded execution is executing basic blocks an unbounded number of times, not a single basic block of unbounded length.

Definition 3. A block-preserving compiler $[\![\cdot]\!]$ is *bounded* if for any expressions e and e' such that $e \rightsquigarrow e' \mid \mathcal{S}$ and any low-level cost metric K, there exist costs C_s for each $s \in \mathcal{S}$ such that if

$$[\![e']\!] \overset{l_1}{\mapsto} D_1 \overset{l_2}{\mapsto} D_2 \mapsto^* D_{m-1} \overset{l_m}{\mapsto} D_m$$

and

$$\begin{aligned} l_1 :: \ldots :: l_m \quad = \quad & L_1 :: \mathsf{tick}(s_1) :: L_2 :: \mathsf{jump}_1 \\ & :: \ldots :: \mathsf{jump}_{n-1} :: L_{2n-1} :: \mathsf{tick}(s_n) :: L_{2n} \end{aligned}$$

where $L_1, \ldots, L_{2n}$ contain no jump instructions, then $K(L_{2n-1}) + K(L_{2n}) \leq C_{s_n}$ and for all $i \in [1, n-1]$, we have $K(L_{2i-1}) + K(L_{2i}) + K(\mathsf{jump}_i) \leq C_{s_i}$

Finally, we require that the annotation algorithm can only increase the cost of the code.

Definition 4. A compiler $[\![\cdot]\!]$ is *expanding* if, for any e and e' such that $e \rightsquigarrow e' \mid \mathcal{S}$, if

$$[\![e']\!] \overset{l_1}{\mapsto} D_1 \overset{l_2}{\mapsto} D_2 \mapsto^* D_{n-1} \overset{l_n}{\mapsto} D_n$$

and

$$l_1 :: \ldots :: l_n = L_1 :: \mathsf{tick}(s_1) :: L_2 :: \ldots :: L_{m-1} :: \mathsf{tick}(s_{m-1}) :: L_m$$

where $L_1, \ldots, L_m$ contain no tick annotations then

$$[\![e]\!] \overset{l'_1}{\mapsto} D'_1 \overset{l'_2}{\mapsto} D'_2 \mapsto^* D_{k-1'} \overset{l'_k}{\mapsto} D'_k$$

where for any K,

$$K(l'_1 :: \ldots :: l'_k) \leq \sum_{i=1}^{m} K(L_i)$$

Discussion of Assumptions We have found the assumptions describe in the previous subsection to hold in practice on the two compilers we have examined, and believe them to hold for many other real-world compilers as well. The soundness of the analysis is surprisingly robust to compiler optimizations, including those that would appear to disrupt the basic block structure of the code. For example, if the body of a function is annotated as `bb25` and this function is inlined into several other basic blocks by the compiler, the low-level analysis will correctly find the cost of those basic

blocks including the inlined function body. The annotation for `bb25` will appear within each of these basic blocks, but having additional annotations in a basic block is not harmful. Furthermore, the low-level analysis will find `bb25` itself to have no cost, so the cost of the function will not be double-counted. Some compiler transformations may, however, introduce some imprecision into the analysis. For example, if `bb38` corresponds to the body of a loop with M instructions that is unrolled by the compiler, the low-level analysis might find `bb38` to have $2M$ instructions. The basic block analysis, however, will still believe that the loop runs for N iterations rather than $N/2$. For the compilers we examine in this paper, we have found such imprecisions to be small, as we demonstrate in the evaluations of the individual analyses. We additionally assume that the annotation process does not result in code with a shorter execution time than the unannotated code. This is a broadly reasonable assumption for most compilers, as we only add code and we do so in such a way that should not enable new optimizations (indeed, as discussed above, some of our annotations inhibit certain optimizations, but through careful engineering of the annotation process, we have kept to a minimum the overhead that results from these lost optimizations).

4 Case Study: OCaml Bytecode Compiler

This section presents the first of two case studies in combining the basic block execution counts from the previous section with low-level information about each basic block. In this first study, we use the analysis to estimate the number of instructions executed by the OCaml virtual machine on a program compiled with the OCaml bytecode compiler `ocamlc`. We focus on `ocamlc` in this section because it generates easily machine-readable code for which we could easily develop tooling to analyze the basic blocks. A similar analysis would be possible for the native compiler (e.g., for counting x86 instructions), but would require tools to parse and analyze the binaries for the desired target platform.

The soundness of the analysis in this section depends on the assumptions described in Section 3.1. Although we have not formally proven that these assumptions are valid for `ocamlc`, we believe this to be the case based on extensive experience with the compiler and hand-examination of the compiler source code.

As part of this case study, we implemented a tool that accepts a compiled OCaml file, parses the bytecode instructions, splits the instructions into basic blocks and applies a "low-level resource metric" to each basic block, outputting the resulting cost for each basic block. A low-level resource metric for OCaml bytecode is a function from bytecode instructions to (possibly symbolic) costs. We have implemented two such resource metrics. The first, `vmsteps`, counts the instructions; the resulting bound is an upper bound on the number of steps taken by the virtual machine during execution. In case instructions do not take equal time to execute, it may be helpful to break this count down by type of instruction; as an example, we provide a second resource metric, `vmap`, which returns the number of function applications.

Combining the above analysis with the basic block analysis of the previous section results in a full end-to-end system that accepts an OCaml program and reports the resource usage based on the chosen low-level resource metric. RaML, and thus our extended analysis tool, is able to report costs both for a whole program and for individual functions; the latter are parametrized by the size and structure of the function arguments. After basic block instrumentation, the instrumented program is passed through the RaML compiler pipeline, followed by the symbolic resource analysis. The result of this analysis is an upper bound on the number of executions of each basic block. Separately, our pipeline compiles the instrumented program using `ocamlc` and interprets the compiled program using the tool described above and analyzes the basic blocks, which are annotated in the compiled program, using the chosen low-level resource metric.

We evaluated our end-to-end resource analysis for OCaml bytecode programs on a number of RaML benchmarks, mostly adapted from the benchmark suite available on the RaML website [27]. The benchmarks used, with brief descriptions, are listed in Table 1. The table also shows the number of basic blocks annotated by our program transformation for the example program. For each benchmark, we analyze the cost of the "main" function of the benchmark, parametrized by the program inputs (e.g., the cost of the Quicksort function in terms of the size of the list).

Tables 2 and 3 give the analysis results for the example functions, in terms of the input size parameters. The tables also give the analysis time for each analysis. We also compiled each example file (instrumented with the basic block annotations) with the unmodified `ocamlc` compiler, and ran the compiled files using INRIA's standard `ocamlrun` bytecode interpreter. When run in debug mode with the proper options, `ocamlrun` produces a trace of the instructions executed. We then processed this trace using standard GNU utilities to determine the actual values of each of the low-level resource metrics for the execution. The results in the two tables show that the analysis is quite precise; in many cases it predicts the cost exactly.

5 Case Study: Combining with WCET for Basic Blocks

In this case study, we use worst-case execution time (WCET) analysis for the low-level analysis of the compiled code. We discuss WCET in more detail in Section 6. Briefly, WCET tools analyze programs, generally at the level of machine code, to derive sound upper bounds on execution time. These tools use detailed hardware models to simulate the behavior

Benchmark	Input Size Parameters	Blocks	Description
append	$\|L_1\|$, $\|L_2\|$	9	Append lists L_1 and L_2 (with lengths $\|L_1\|$ and $\|L_2\|$)
calculator	X, L, N, K	21	Evaluate a symbolic arithmetic expression with X subtractions, L additions and N unary numbers $\leq K$.
isort	M	27	Sort M integers using insertion sort
quicksort	M	34	Sort M integers using Quicksort

Table 1. Example programs used for evaluation, along with the number of basic blocks in the instrumented code.

File	Analysis Result (instructions)	Actual Cost (instructions)	Analysis Time (s)
append	$8 + 20\|L_1\|$	$8 + 20\|L_1\|$	0.06
quicksort	$7 + 33.5M + 32.5M^2$	$7 + 32.5M + 32.5M^2$	1.04
isort	$7 + 13M + 18M^2$	$7 + 13M + 18M^2$	0.07
calculator	$8 + 46KLN + 30.5KN + 33L + 33X$	$8 + 33L + 33X + 30.5KN + 6KLN$	12.18

Table 2. Analysis results and times for low-level metric `vmsteps` in terms of input size.

File	Analysis Result (applications)	Actual Cost (applications)	Analysis Time (s)
append	$\|L_1\|$	$\|L_1\|$	0.06
quicksort	$3M + M^2$	$3M + M^2$	1.01
isort	$1.5M + 0.5M^2$	$1.5M + 0.5M^2$	0.08
calculator	$2KLN + 1.5KN + 2L + 2X$		11.94

Table 3. Analysis results and times for low-level metric `vmap` in terms of input size.

of memory, caches and instruction pipelines to predict precise wall-clock times for execution on real-time hardware. In general, these tools are quite good at deriving timing bounds for straight-line code but struggle with features such as indirect jumps, loops and recursion, which are present in many real-world programs but especially prevalent in functional languages such as OCaml. These tools often require users to annotate targets of indirect jumps as well as bounds on the number of iterations of a loop or recursive instances of a function. Conversely, the extension of RaML we developed (Section 3) derives upper bounds on the number of executions of each basic block, which implies a great deal of information about loop and recursion bounds.

The above discussion suggests two ways to combine WCET information with our basic block analysis. First, we can use a WCET tool to derive upper bounds on the execution time of each basic block and substitute these times into the basic block analysis results, much as we did with the instruction counts in the previous case study. Second, we can use the basic block execution counts derived by our analysis to automatically generate annotations that can be used to guide a WCET tool, allowing it to generate more precise results. In the remainder of this section, we explore both approaches, and compare the two quantitatively and qualitatively.

For the WCET analyses in this section, we use the aiT static timing analysis tool developed by AbsInt Angewandte Informatik GmbH. This is a commercial-grade, state-of-the-art tool used widely in industry, including for the analysis of real-time safety-critical software. The aiT tool targets many platforms; we use the version for the 32-bit ARM architecture. To apply this tool to our example programs, we compile the (basic-block-annotated) programs for a 32-bit ARM target using OCaml's native code compiler `ocamlopt` [3].

In our timing analyses, we assume that all memory allocations succeed and that garbage collection is never invoked. Timing analysis for garbage collection is an interesting research question in itself and is outside the scope of this paper. We also do not compute the cost of the FFI calls that are used to annotate basic blocks, instead replacing this cost with a small, fixed number of cycles. As a result, the WCET analysis we perform is sound for a version of the annotated binary that has been altered (using standard binary rewriting techniques) to not perform the FFI calls. We also note that we are unable to actually run the example programs on the hardware targeted by aiT's analysis, as the timing analysis tool

[3]The aiT software officially only supports a fixed set of C compilers. However, `ocamlopt` follows relatively standard conventions and emits code that largely resembles the output of standard compilers such as `gcc`, so we are able to apply the aiT analysis after a handful of manual annotations.

targets only very basic microcontrollers; porting the OCaml ecosystem to such hardware poses substantial challenges.

5.1 Worst-case Execution Time for Basic Blocks

In our first set of experiments, we used aiT to derive WCET bounds for each basic block of our example programs. For each example, we manually examined the disassembled ARM binaries and compiled a list of basic blocks. For each basic block, we annotated the memory addresses of the start and end of the block, as well as any additional user annotations that would be necessary to help aiT analyze the code for the basic block (e.g., indicating that allocations succeed). Compiling this data took approximately 5-15 minutes per example and likely could be largely or entirely automated. From this basic block data, we used a custom-made script to automatically set up an aiT analysis for each basic block. We ran the resulting analyses (the analysis of each basic block took under 1 second) and parsed the results into worst-case cycle counts for each basic block.

We ran the basic block analysis of Section 3 on each example, this time instructing it to output the execution counts for each basic block (rather than preemptively substitute instruction costs, as in the previous case study) and substituted the cycle count for each basic block into the result. Table 4 shows the results of the analysis, in terms of the input sizes, as well as the time taken by the basic block analysis. These times are greater than those of Table 2 because of the additional cost of determining the coefficients for each symbol.

To put these results into context, we paired each benchmark program with a concrete input and ran the aiT tool on the whole program (aiT is not able to reason about costs symbolically in terms of input size), with manual control-flow annotations to inform aiT of branch information and recursion bounds (information which our basic block analysis derives automatically). The results, including the size of the input and the instantiation of the bounds of Table 4 to these input sizes, are shown in Table 5. Figure 4 shows a more detailed plot for the function quicksort that compares the symbolic bound with the aiT result for different input sizes. The cycle counts predicted by our analysis come within 30% of the whole-program aiT results. We find this accuracy acceptable, given that the whole-program aiT analysis is able to make use of information such as what data remains in cache between basic blocks. Our analysis, by contrast, must assume the worst-case time for each instance of each basic block. Still, the analysis provides a similarly precise and much less effortful alternative to manual annotations, which can be time-consuming and error-prone to provide.

The soundness of the above analysis depends on the same assumptions as the analysis from the previous section. Again, we believe that these assumptions hold for `ocamlopt`.

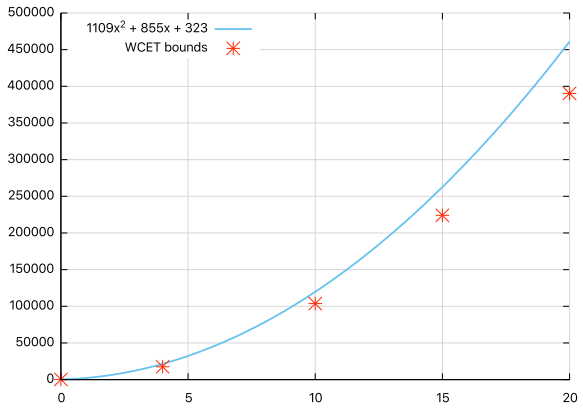

Figure 4. Concrete aiT clock-cycle bounds (red points) for different input sizesand the symbolic bound derived by the combined resource analysis (blue line) for quicksort.

5.2 Deriving Flow Constraints from Basic Block Analysis

As we discussed at the end of the previous subsection, aiT (and WCET tools in general) work better on a whole program, assuming jump targets and loop bounds are predictable, than on individual basic blocks because they can make use of additional cache and pipeline information. This motivates our next set of experiments, in which we use the information returned by the basic block analysis to generate aiT annotations (of the form we manually inserted for the comparison in Table 5) automatically. Recall that our basic block analysis generates an upper bound on the resource usage of the program in terms of the resource usage of each basic block. In this bound, the coefficient of each basic block is an upper bound on the number of times that basic block executes. We use these bounds, together with the code addresses of each basic block, to generate, for each benchmark, a set of "flow constraints" that indicate to aiT how many times execution passes a particular program point. We then ran the aiT tool on each benchmark with the generated constraints in order to obtain upper bounds on the number of cycles for each program execution.

Table 6 shows the results of these analyses for the problem sizes given in Table 5, together with the aiT results with the manual annotations derived for the previous set of experiments. In each case except calculator (for which we were not even able to obtain tight bounds manually), our tool automatically generated constraints that resulted in identical WCET bounds. Thus, in these benchmarks, by using the results of the basic block analysis, we were able to match the results obtained with hand calculations and manual annotations, which are tedious and error-prone.

[4]Using the default analysis parameters, the WCET calculation for the calculator example runs out of memory, so we ran it using a limited call

Benchmark	Analysis Result (cycles)	BB Analysis Time (s)
append	$357 + 687\|L_1\|$	0.19
quicksort	$1109M^2 + 855M + 323$	1.95
isort	$468.5M^2 + 734.5M + 327$	0.21
calculator	$1472KLN + 1018KN + 1533L + 1024.5X + 402$	89.69

Table 4. Analysis results and times for substituting basic-block cycle counts.

File	Input Size	Instantiation (cycles)	WCET w/ manual annotations (cycles)
append	$\|L_1\| = 5$	3792	3112
quicksort	$M = 3$	12869	10065
isort	$M = 3$	6747	5456
calculator	$(X, L, N, K) = (1, 3, 5, 2)$	60365.5	29950

Table 5. Instantiations of the cycle count results to concrete inputs, compared to aiT results with manual annotations.

	WCET with... (cycles)	
Benchmark	Automatic Constraints	Manual Constraints
append	3112	3112
quicksort	10065	10065
isort	5456	5456
calculator	55454[4]	29950

Table 6. WCET using basic block results as flow constraints.

5.3 Comparison of the Two Approaches

In this section, we have presented two ways of combining our basic block analysis for OCaml programs with the WCET analysis provided by the aiT tool. In the first subsection, we gathered WCET bounds for each basic block and substituted these into the basic block analysis results to obtain upper bounds on cycle count for each example in terms of the input size. In the second subsection, we used the basic block analysis to bound the number of times each basic block is executed and used these results as annotations for aiT in computing a WCET bound for the cycle count of the whole program. For reference, the bounds resulting from the two approaches ("substitution" and "constraints", respectively) are listed in Table 7.

As discussed earlier, the constraint-generation approach results in tighter bounds. On the other hand, the substitution approach is able to produce bounds that are parametrized by the input size while WCET, at least with the tool used in this paper, is constrained to consider a particular input. In addition, the substitution approach required minimal human effort: we took under approximately 15 minutes per example to generate the input required for the WCET analysis of the basic blocks, and believe this effort could be largely automated. On the other hand, even using the automatically generated flow constraints, analyzing a whole program using a WCET tool requires some manual effort to provide targets of indirect jumps, sanity-check the results of the analysis and add more annotations as necessary to guide the analysis. This effort corresponded to approximately 15-30 additional minutes per benchmark.

An additional advantage of the constraint-generation approach is that it requires fewer assumptions on the compiler: the compiler need only be annotation-preserving, i.e., the predicted upper bound on the number of executions of each basic block must be sound. This assumption guarantees that the generated flow constraints are sound, and so we then simply need to trust the WCET analysis is sound (which is proven in the case of many WCET tools).

[4] string length and aiT's "local worst-case" option, which is faster and less memory-intensive but does not guarantee soundness.

6 Related Work

Automatic Amortized Resource Analysis Automatic amortized resource analysis (AARA) has been developed by Hofmann and Jost [33] to automatically derive linear heap-space bounds for first-order functional programs. It has subsequently been extended to linear bounds for higher-order functional programs [36], polynomial bounds for first-order programs [28, 32], and polynomial bounds for higher-order programs [30]. AARA has also been applied beyond purely functional programming. For instance, it has been integrated with separation logic [6] and type systems for object-oriented languages [34]. Moreover, AARA has been applied to derive resource bounds for programs with references [43], imperative integer programs [16, 17], term rewrite systems [35], and probabilistic programs [45]. These works focus on bounds on the source level that are not connected to the compiled code. Their cost models are also not symbolic.

File	WCET with... (cycles) Substitution	Automatic Constraints	Manual Constraints
append	3792	3112	3112
quicksort	12869	10065	10065
isort	6747	5456	5456
calculator	60365.5	55454	29950

Table 7. A comparison of the two approaches to combining WCET with the basic block analysis.

Previous work on the Hume language in the EmBounded project [26, 36, 37] shares some of the goals this paper. Hume is a functional programming language that is equipped with a linear AARA that can derive bounds on the compiled code. The key idea is to identify snippets of machine code that correspond to high-level language constructs and represent the cost of these snippets through constants in a source-level cost model. The approach presented in this paper is more flexible and can link entire basic blocks in the compiled code with source-level cost annotations. As a result, it can take into account compiler optimizations and derive bounds that apply to custom target-code resource metrics. Additionally, we implemented our technique for OCaml, a real-world optimizing compiler, instead of a research prototype like Hume.

Source-Level Resource Analysis Source level resource analysis has been extensively studied. Apart from AARA, there are techniques based on sized types [52], linear dependent types [40, 41], refinement types [18, 19, 53], other annotated type systems [20, 21], and defunctionalization [7]. Other techniques rely on extracting and solving recurrence relations [1, 2, 11, 22, 23, 38, 39, 54], abstract interpretation [12, 25, 51, 56], and techniques from term rewriting [8, 13, 48]. In contrast to our work, all of these techniques derive bounds that apply to abstract source-level cost models and do not consider compilation.

Worst-Case Execution Time (WCET) Analysis There has also been a large body of work on WCET analysis that operates directly on the machine code [55]. In contrast to source-level analyses, WCET analyses do usually not yield symbolic bounds but focus on modeling caches and instruction pipelines of specific hardware. However, there are techniques that can yield symbolic bounds using a polyhedral abstraction [14, 44]. Altmeyer et al. [3, 4] reported a similar approach. Recently, Ballabriga et al. [9] introduced a method based on symbolic computation over the control flow graph. Since these techniques operate on low-level code, the control flow is more difficult to analyze and type information that can guide the analysis is missing. As a result, these techniques cannot derive bounds for functions or loops that traverse data structures and are less scalable and compositional than source-level techniques.

Resource Preserving Compilation There has been little previous work combining source-level and target-level resource analysis during compilation. Bedin França et al. [10] have described and implemented a verified compiler technique for the sound transport of source-level annotations to compiled code. However, these annotations are limited to memory assertions and cannot directly include loop bounds or other quantitative properties.

Closer to our work is previous work on resource aware compilation for C programs. Carbonneaux et al. [15] have described and implemented a verified technique for compiling stack-space bounds from the C level to assembly code. Similarly, the Certified Complexity (CerCo) project [5] has developed a C compiler that can transport resource bounds from the source to the target level. Our work uses similar techniques to the aforementioned projects. However, we implemented the technique for a functional language and an existing optimizing compiler, support higher-order functions, and combine compilation with a sophisticated source-level cost analysis that can automatically derive symbolic bounds. Paraskevopoulou and Appel [50] showed that an implementation of closure conversion (in a compilation pipeline from a functional language to a subset of C) preserves space and time guarantees, but their approach does not extend to other phases of compilation nor other resource metrics.

7 Conclusion

In this paper, we described a technique for deriving resource bounds of compiled code by combining resource bounds inferred at the source level with bounds inferred at the target level. This combines the best of both worlds between using type information at the source level to make sound inferences about the control flow of the program and using precise timing information available at the target level to produce bounds that match real-world executions. Our experiments indicate that it is possible to derive useful symbolic and concrete performance information at compile time.

There are several promising avenues for future work in integrating compilation, source level resource analysis, and target level resource analysis. An area that we plan to study is compiler optimizations and their effect on high-level resource bounds. Moreover, we plan to take into account automatic memory management following previous results [47].

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. 2012. Automatic Inference of Resource Consumption Bounds. In *Logic for Programming, Artificial Intelligence, and Reasoning, 18th Conference (LPAR'12)*. 1–11.

[2] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. 2015. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*. 85–100.

[3] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. 2011. Precise and Efficient Parametric Path Analysis. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'11)*. 141–150.

[4] Sebastian Altmeyer, Christian Humbert, Björn Lisper, and Reinhard Wilhelm. 2008. Parametric Timing Analysis for Complex Architectures. In *4th IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*. 367–376.

[5] Roberto M. Amadio, Nicolas Ayache, Francois Bobot, Jaap P. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2014. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis*, Ugo Dal Lago and Ricardo Peña (Eds.).

[6] Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *19th European Symposium on Programming (ESOP'10)*. 85–103.

[7] Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2012. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th International Conference on Functional Programming (ICFP'15)*. 152–164.

[8] Martin Avanzini and Georg Moser. 2013. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*. 55–70.

[9] Clément Ballabriga, Julien Forget, and Giuseppe Lipari. 2017. Symbolic WCET Computation. *ACM Trans. Embed. Comput. Syst.* 17, 2 (Dec. 2017), 39:1–39:26.

[10] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. 2012. Formally Verified Optimizing Compilation in ACG-based Flight Control Software. In *Embedded Real Time Software and Systems (ERTS 2012)*.

[11] Ralph Benzinger. 2004. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science* 318, 1-2 (2004), 79–103.

[12] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference (LPAR'10)*. 103–118.

[13] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference (TACAS'14)*. 140–155.

[14] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. 2009. An Efficient Algorithm for Parametric WCET Calculation. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*. 13–21.

[15] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-End Verification of Stack-Space Bounds for C Programs. In *35th Conference on Programming Language Design and Implementation (PLDI'14)*. Artifact submitted and approved.

[16] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. 2017. Automated Resource Analysis with Coq Proof Objects. In *29th International Conference on Computer-Aided Verification (CAV'17)*.

[17] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds. In *36th Conference on Programming Language Design and Implementation (PLDI'15)*. Artifact submitted and approved.

[18] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17)*.

[19] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*. 406–431.

[20] Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *27th Symposium on Principles of Programming Languages (POPL'00)*. 184–198.

[21] Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th Symposium on Principles of Programming Languauges (POPL'08)*. 133–144.

[22] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2012. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th International Conference on Functional Programming (ICFP'15)*. 140–151.

[23] Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposium (APLAS'14)*. 275–295.

[24] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 116 (Oct. 2018), 27 pages.

[25] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th Symposium on Principles of Programming Languauges (POPL'09)*. 127–139.

[26] Kevin Hammond and Greg Michaelson. 2003. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In *Generative Programming and Component Engineering, 2nd Int. Conf. (GPCE'03)*. 37–56.

[27] Jan Hoffmann. 2018. Resource Aware ML. http://www.raml.co

[28] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*.

[29] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*.

[30] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*.

[31] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

[32] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *19th European Symposium on Programming (ESOP'10)*.

[33] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*. 185–197.

[34] Martin Hofmann and Steffen Jost. 2006. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming (ESOP'06)*. 22–37.

[35] Martin Hofmann and Georg Moser. 2015. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*. 241–256.

[36] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th Symposium on Principles of Programming Languages (POPL'10)*. 223–236.

[37] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. 2009. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*. 354–369.

[38] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 248–262.

[39] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. 2018. Non-linear reasoning for invariant synthesis. *PACMPL* 2, POPL (2018), 54:1–54:33.

[40] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *26th IEEE Symposium on Logic in Computer Science (LICS'11)*. 133–142.

[41] Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. In *40th Symposium on Principles of Programming Languages (POPL'13)*. 167–178.

[42] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. *The OCaml system release 4.07*. INRIA.

[43] Benjamin Lichtman and Jan Hoffmann. 2017. Arrays and References in Resource Aware ML. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD'17)*.

[44] Björn Lisper. 2003. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *3rd International Workshop on Worst-Case Execution Time Analysis (WCET'03)*. 99–102.

[45] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *39th Conference on Programming Language Design and Implementation (PLDI'18)*.

[46] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *38th IEEE Symposium on Security and Privacy (S&P '17)*.

[47] Yue Niu and Jan Hoffmann. 2018. Automatic Space Bound Analysis for Functional Programs with Garbage Collection. In *22nd International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'18)*.

[48] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. 2013. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *Journal of Automated Reasoning* 51, 1 (2013), 27–56.

[49] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Conference on Programming Language Design and Implementation (PLDI'15)*. 369–378.

[50] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure Conversion is Safe for Space. *Proc. ACM Program. Lang.* 3, ICFP, Article 83 (July 2019), 29 pages. https://doi.org/10.1145/3341687

[51] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th International Conference (CAV'14)*. 743–759.

[52] Pedro Vasconcelos. 2008. *Space Cost Analysis Using Sized Types*. Ph.D. Dissertation. School of Computer Science, University of St Andrews.

[53] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *OOPSLA*.

[54] Ben Wegbreit. 1975. Mechanical Program Analysis. *Commun. ACM* 18, 9 (1975), 528–539.

[55] Reinhard Wilhelm et al. 2008. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.* 7, 3 (2008).

[56] Florian Zuleger, Moritz Sinn, Sumit Gulwani, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th International Static Analysis Symposium (SAS'11)*. 280–297.