2012 ACM SIGPLAN Workshop on ML

Handling Overflow in MLton

Alexander Bjerremand Hansen
Mjølner Informatics
alx@cs.au.dk

Jan Midtgaard
Aarhus University
jmi@cs.au.dk

Abstract

The definition of Standard ML dictates that overflow in integer arithmetic raises an overflow exception. As a consequence standard-conforming implementations of the language have to check for overflow which can be costly. We examine the possibility of further optimizing the code generated by the industrial-strength compiler MLton. By studying the code generated for integer arithmetic we identify dead code where overflow cannot happen. We propose a flow-sensitive integer interval analysis that detects and removes such needless overflow checks on integer arithmetic. We have implemented an analysis prototype in Standard ML and integrated it into MLton as an additional optimization pass. It is able to remove 22% of the overflow checks in the MLton benchmark suite. The binary size of the programs decrease up to 10% and the run time of arithmetic heavy programs decrease significantly - in some cases up to 27%.

1. Introduction

Program optimization is an important aspect of a compiler. The Standard ML compiler MLton is a whole program optimizing compiler that focuses on generating fast and small binary executables. However MLton, the industrial-strength compiler developed over the last 14 years, produces sub-optimal code regarding overflow handlers. In this paper we identify and remove dead overflow handlers in real-world programs using standard static analysis tools and thereby reduce the size and run time of the generated binaries.

1.1 The MLton compiler

During the compilation of ML programs several intermediate languages are used internally by the MLton compiler. These languages primarily serve as translation steps from ML source code to machine code where each translation performs one or more transformations on the code from the previous level. Among these translations are *monomorphisation* that removes all uses of polymorphism and *closure conversion* (or rather: defunctionalization) that removes higher-order functions. The intermediate language reached after closure conversion is a simply-typed, first-order language called SSA (Static Single-Assignment). This intermediate form allows for easy optimization and the MLton compiler performs many of its optimizations at this level.

1.2 SSA form

SSA form is a common intermediate representation used for optimization in modern compilers. As the *single-assignment* part of the name suggests, each variable has only one definition in the program text. The *static* part emphasizes that the single-assignment part is a compile-time property rather than a dynamic one since an assignment can be placed in a function that is called several times.

In MLton's SSA form arithmetic occurs in a construct with an explicit overflow handler attached and overflow will implicitly be signaled by the primitive arithmetic operations. Since the construct that performs possibly overflowing arithmetic is a *control transfer* from one SSA block to the next, the control-flow on the primitive overflow is explicit. The overflow handler block will then explicitly raise SML's dedicated `Overflow` exception with the `raise` control transfer. Alternatively, arithmetic in SSA can be performed using primitive function definitions that take the operands as parameters and return the result of the calculation. These operations do not signal overflow.

Consider the following example taken from the intermediate output when compiling `hamlet.sml` from MLton's benchmark suite:

```
L_25120 (x_16013: word32)
 x_10771: bool = WordU32_lt (x_10769, global_33)
 case x_10771 of
   true => L_22171 | false => L_24271
L_22171 ()
 loop_355 (x_10769 + global_5) Overflow => L_24272()
```

It contains arithmetic with an explicit overflow handler attached in the last line. Note that there are no other call sites to the label `L_22171` than the one in the *case transfer* above, so we are guaranteed to have been through the preceding comparison when arriving at that label. The involved variables with the `global` prefix are defined as constants in the header of the SSA program as follows:

```
global_5:  word32 = 0x1
global_33: word32 = 0x3B7
```

2. Analysis and Transformation

Overflow of integers is essentially concerned with the number of bits used to express a certain value. If the number of bits required is bigger than what a resulting word type can hold an overflow happens. We therefore need to model the number of bits used by variables in the program. For our analysis we will use an interval lattice for each variable in the program. For n-bit words let $MAX = 2^{n-1} - 1$ and $MIN = -(2^{n-1})$ and define the finite interval lattice $\langle IL; \sqsubseteq \rangle$ as follows:

$$IL = \{\langle a, b\rangle \mid a, b \in \{MIN, \dots, MAX\} \wedge a \leq b\} \cup \{\bot\}$$
$$\bot \sqsubseteq \langle a, b\rangle \qquad \langle a, b\rangle \sqsubseteq \langle c, d\rangle \triangleq c \leq a \wedge b \leq d$$

Based on IL we now define a lattice I that relates variables to intervals under a pointwise ordering: $\langle\, Var \to IL; \dot{\sqsubseteq} \rangle$.

2.1 Value propagation

To gain information for each variable we must consider what constructs in the SSA program that give rise to information and how to update the interval lattice accordingly. The following rule captures the behavior of a constant definition of a variable on the lattice I.

$$\frac{(X \leftarrow v) \in P \qquad v : word}{I[\![X]\!] = \langle v, v\rangle}\ \textsc{Constant Definition}$$

This rule specifies that for all constant word values v assigned to a variable X in the input program P, we update I's entry for X with the interval containing only v. Several other rules propagate this information throughout the program. For example, for calls in the source code, we join the value of I for an actual parameter into the receiver's formal parameter.

There are rules for every construct in the SSA language that manipulate word values; arithmetic, primitive functions, calls, tail calls, and gotos. Arithmetic operations are defined on the interval lattice and performed on the abstract values to soundly model integer arithmetic on variables in the SSA source. Due to lack of space we only list two more rules. The first rule is used to refine the intervals for variables when we learn more about them. When a statement S compares two word variables and uses the result in a branch, we obtain valuable information about both variables on both branches.

$$\dfrac{\begin{array}{c}(X \leftarrow \texttt{Word\_lt}(x_1, x_2)) = S \qquad S \in P \\ X : \mathit{bool} \qquad \mathit{label}(S) = \texttt{l}\end{array}}{\begin{array}{c}A_{\texttt{T},x_1}[\![X]\!] = L(x_1, \texttt{l}) \prec L(x_2, \texttt{l}) \\ A_{\texttt{T},x_2}[\![X]\!] = L(x_2, \texttt{l}) \succeq L(x_1, \texttt{l}) \\ A_{\texttt{F},x_1}[\![X]\!] = L(x_1, \texttt{l}) \succeq L(x_2, \texttt{l}) \\ A_{\texttt{F},x_2}[\![X]\!] = L(x_2, \texttt{l}) \prec L(x_1, \texttt{l})\end{array}} \ \textsc{Prim. Def. LT}$$

The symbols $\prec$ and $\succeq$ are defined as interval bounding operations with signature $I \times I \rightarrow I$. The function $L(x, \texttt{l})$ is used to look up the interval for the variable x when we need the abstract value inside the SSA code block `l`. $A_{\texttt{T},y}[\![X]\!]$ and $A_{\texttt{F},y}[\![X]\!]$ represent the values of the variable y if the boolean variable X has evaluated to `true` and `false`, respectively. Both these lattices store temporary information that will be used when a case transfer dispatches on the boolean variable X. When this happens the following rule will propagate the stored information:

$$\dfrac{\begin{array}{c}(\texttt{case}\ x\ \texttt{of true=>l | false=>l}') \in P \qquad x : \mathit{bool} \\ A_{\texttt{T},y}[\![x]\!] = i \qquad A_{\texttt{T},z}[\![x]\!] = i' \\ A_{\texttt{F},y}[\![x]\!] = i'' \qquad A_{\texttt{F},z}[\![x]\!] = i'''\end{array}}{\begin{array}{c}i \sqsubseteq F[\![l]\!][\![y]\!] \qquad i' \sqsubseteq F[\![l]\!][\![z]\!] \\ i'' \sqsubseteq F[\![l']\!][\![y]\!] \qquad i''' \sqsubseteq F[\![l']\!][\![z]\!]\end{array}} \ \textsc{Case Trans.}$$

The information is moved from the temporary lattice A to a new lattice F that represent *facts*. Facts are maps from SSA blocks to variables to intervals. These facts are used in the lookup function L. When we need the interval for a variable we first check if a fact is present in the map for the variable's enclosing code block. If so, we return the fact interval. Otherwise we fall back to the interval value from the global I lattice. These rules and a number of additional ones express the static analysis over the lattices I, A, and F and enable us to transform a given program based on the analysis result.

2.2 Program transformation

For each arithmetic transfer let a and b be the arguments to the arithmetic operator `op` and let MIN and MAX be the smallest and largest number, respectively, that the resulting variable can hold. These two numbers depend on the number of bits in the word and whether or not the value is signed (using one bit for the sign). Let $I[\![a]\!] = \langle a_l, a_h\rangle$ and $I[\![b]\!] = \langle b_l, b_h\rangle$ and a `op` $b = c$ then we have $I[\![c]\!] = I[\![a]\!]\ \bar{\texttt{op}}\ I[\![b]\!] = \langle c_l, c_h\rangle$.

To detect if an arithmetic transfer is guaranteed to not overflow we inspect the bounds of the resulting interval. If $c_l > \mathit{MIN}$ and $c_h < \mathit{MAX}$ no overflow can happen and we are able to transform this arithmetic transfer by removing the overflow check.

To remove the overflow check we translate the arithmetic transfer into a primitive function call and a *goto transfer*. Hence we translate $\texttt{l}(a\ \texttt{op}\ b)\ \texttt{Overflow => l}'$ into $x \leftarrow p_{\texttt{op}}(a, b)$ followed by $\texttt{l}(x)$, where x is a fresh variable and where $p_{\texttt{op}}$ is the primitive function corresponding to the operator `op` in the arithmetic transfer; `Word_add`, `Word_sub`, or `Word_mul`. Following this transformation, MLton's ecosystem of SSA optimization passes may now remove the dead handler code at $\texttt{l}'$.

3. Results

If we examine the example SSA code from the introduction the arithmetic in the last line cannot possibly overflow. The variable `x_10769` is used in a less-than comparison against the constant `global_33` which in this case is `951` in decimal. Four facts are created for the boolean variable `x_10771` and when the case transfer branches on this boolean variable we are able to utilize those facts. In the `true` branch we now know that variable `x_10769` is less than `951` and when `global_5` (which is `1`) is added to it no overflow can happen in 32-bit arithmetic.

When compiling `hamlet.sml` with this optimization and searching for the specific instance from the example we find the following snippet:

```
loop_355 (x_10769: word32)
 x_10772: bool = WordU32_lt (x_10769, global_33)
  case x_10772 of
   true => L_22171 | false => L_24271
 L_22171 ()
  x_18279: word32 = Word32_add (x_10769, global_5)
  loop_355 (x_18279)
```

We have successfully removed an overflow transfer and therefore an explicit overflow check on arithmetic that can never overflow.

The first measure we will examine is the number of overflow handlers that are transformed in the entire MLton benchmark suite. We find that out of 4240 overflow transfers we are able to transform 913 (22%) into a statement and a goto transfer. This leads to reduced binary sizes of many of the programs ranging from 0-10% depending on the amount of removed arithmetic transfers and whether the removal of overflow checks will create dead code that is subsequently removed by MLton's other optimization passes.

The run time of the programs are also reduced depending on how many times the transformed arithmetic is run. In benchmarks with intensive integer arithmetic such as `imp-for` we are able to reduce the run time by 27%. Below we list a subset of benchmark programs representable of the entire benchmark suite and their corresponding compile-time (CT) and run-time (RT) results:

Benchmark	LOC	# hits	Δ CT	RT w/o	RT w	Δ RT
`hamlet`	22.9K	41	19.15	9.60	9.59	-0.01
`imp-for`	31	4	-0.05	8.88	6.55	-2.33
`matrix-multiply`	60	8	-0.03	6.25	4.56	-1.69
`mlyacc`	7.3K	79	8.37	5.61	5.61	0
`tailfib`	22	5	-0.04	10.54	8.45	-2.09

For more details we refer to the first author's MSc thesis [1].

4. Conclusion

We have illustrated how to detect and remove unused overflow handlers by using a flow-sensitive interval analysis to drive an additional MLton optimization pass. The result can successfully remove 22% of the emitted overflow checks for MLton's benchmark suite and results in speed ups of up to 27%. The optimization comes at the cost of increased compile times. The current prototype implementation is unoptimized in this respect and as a result does not scale well with the input size. This may be improved by a better choice of data structures and by adjusting the widening heuristics of the analysis; only in the context of a loop is widening necessary. Finally we see several possibilities for improving the analysis further, e.g., modeling vector lengths and intraprocedural return values more precisely can potentially enable even more optimization.

References

[1] Hansen, A. B. 2011. Exception Analysis in MLton, MSc thesis, Dept. Comp. Sci., Aarhus University. Available at `http://cs.au.dk/~alx/`.