Optimization Techniques for Abstract State Machines-Based Programs

Fabio Tirelo Roberto da Silva Bigonha

Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Av. Antônio Carlos, 6627, CEP 31270-010, Belo Horizonte, MG, Brazil
{ftirelo,bigonha}@dcc.ufmg.br

Abstract

Optimization techniques are applied on compiling code for language based on Abstract State Machines (ASM). The techniques are intended to be used on translating ASM code into imperative language such as C. This paper also shows quantitative analysis of the code produced from the selected benchmark. Experiments show that these optimization techniques may improve the performance of the code execution in up to 94%.

1 Introduction

Abstract State Machines (ASM) are a formal specification model [11]. A formal specification method based on ASM may be directly executed on von Neumann machines and its mathematical foundations provide the basis for formal system verification. The model is considered very simple and can be easily used by programmers. Different sorts of systems, such as Computer Architectures [3,5], Programming Languages Semantics [6,7], Distributed Systems [2,4], and Real-time systems [10,12], have been specified using ASM.

An ASM specification consists of a basic model which is successively refined until it is completely implemented, i.e., it is in its most detailed or concrete level. However, a naïve code generator may produce inefficient code because of some particular characteristics of the model. In fact, one can obtain optimized and efficient code by applying optimization techniques which directly attack the distinguishing traits that separate ASM model from imperative programming.

There are various ASM implementations including interpreters and compilers. The most important interpreters are those of Michigan[8] and ASM-Workbench[9]. In those implementations, the main focus is not efficiency,

so sophisticated optimization techniques are not applied. Compilers which apply optimization techniques are XASM [1] and EVADE [15]. In XASM, the only optimization reported is the efficient representation of functions by means of hash tables. The only optimization reported in EVADE is common sub-expressions elimination. No existing tool applies optimization techniques which directly address ASM inefficiency prone features, since they use only classical techniques from imperative code optimization.

2 Abstract State Machines

This section presents a brief description of ASM model. A complete definition can be found in [11].

An ASM specification consists of defining an algebra, or initial state, and a transition rule. The algebra consists of a vocabulary, i.e., the set of symbols used in the transition rule, and the interpretation of the vocabulary symbols. The transition rule defines the state transition by redefining the interpretation of the vocabulary symbols.

In ASM, systems are operationally defined by means of a state sequence caused by the transition rule. In this model, there exists the notion of current state, denoted by $\mathcal{S}$, and a new state is created by executing the transition rule at $\mathcal{S}$. A state is a mapping from *locations* to values.

A *location* is a pair $(f, \bar{x})$, where f is a n-ary function name and $\bar{x}$ is a tuple of n elements. An *update* is a pair (l, y), where l is a location and y is an element of the co-domain of f.

A transition rule can be of the following types:

- *update* – when it has the form "$f(\bar{x}) := y$", it has the effect of modifying the function f at the point $\bar{x}$, i.e., the location $(f, \bar{x})$ so as to return y in the next state;
- *conditional* – when it has the form "**if** g **then** R_1 **else** R_2 **end**", where g is a boolean expression and R_1 and R_2 are transition rules. It has the effect of executing R_1, if the guard g evaluates in *true*, or executing R_2 otherwise;
- *block* – when it has the form "R_1, R_2", where R_1 and R_2 are transition rules, it has the effect of executing rules R_1 and R_2 in parallel.

Executing an ASM program consists of *repeatedly* applying its transition rule, with the restriction that updating rules only take effect in the next state. A rule R, when fired at state $\mathcal{S}$, generates an update set, denoted by $\mathit{Updates}(R, \mathcal{S})$, which is defined as:

- if $R \equiv$ "$f(\bar{x}) := y$", where $\bar{x} = \langle x_1, \cdots, x_n \rangle$, then

$$\mathit{Updates}(R, \mathcal{S}) = \{((f, \langle \mathcal{S}(x_1), \cdots, \mathcal{S}(x_n) \rangle), \mathcal{S}(y))\},$$

  where $\mathcal{S}(E)$ the evaluation of expression E in state $\mathcal{S}$;

- if $R \equiv$ "$R_1, \cdots, R_k$", then

$$Updates(R, \mathcal{S}) = \bigcup_{i=1}^{k} Updates(R_i, \mathcal{S});$$

- if $R \equiv$ "**if** g **then** R_1 **else** R_2 **end**", then

$$Updates(R, \mathcal{S}) = \begin{cases} Updates(R_1, \mathcal{S}), \text{ if } g = \text{true} \\ Updates(R_2, \mathcal{S}), \text{ otherwise.} \end{cases}$$

Firing R at $\mathcal{S}$ generates a new state $\mathcal{S}'$ in which the application of a function f to $\langle a_1, \cdots, a_n \rangle$ is defined as:

$$\mathcal{S}'(f(a_1, \cdots, a_n)) = \begin{cases} y, \text{ if } ((f, \langle a_1, \cdots, a_n \rangle), y) \in Updates(R, \mathcal{S}) \\ \mathcal{S}(f(a_1, \cdots, a_n)), \text{ otherwise.} \end{cases}$$

Thus, given an initial state $\mathcal{S}_0$, the repetitive execution of the transition rule R produces a sequence of states $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \cdots$, where $\mathcal{S}_{i+1}$ is the state generated by firing R at $\mathcal{S}_i$, $i \geq 0$.

It is important to emphasize that control flow in ASM is quite different from traditional imperative language model. First, rules in the same block are all executed in parallel. For instance, the block "$x := y, y := x$" has the effect, in ASM, of exchanging the values of x and y. The second difference is that in ASM there are not linguistic constructions for loops such as *while* commands or recursion. Transition rules just show how a new algebra can be built by means of assigning new values to the vocabulary symbols.

3 Naïve Compilation of ASM Programs

In this section, we show how C code can be generated from a transition rule R.

Basically, update rules are compiled into C commands to insert update sets into the update list. Conditional rules are compiled into appropriate flow of control commands. The rules of a block are compiled into a sequential C code, not necessarily in the same order as they occur.

At the end of the transition rule, the compiler generates code to fire all updates collected in the list and concludes the code with an unconditional jump to the first instruction of the transition rule, in order to implement the repetitive character of the execution of R.

The update list is a structure in which information on all updates in the current transition rule step are saved. At the end of the execution of each step, all updates in the list are fired. For instance, consider the transition rule "`f(x) := 1 + g(k), g(p) := 1 + f(x)`". The generated code is:

```
INIT: SAVE_UPDATE(f[x], 1 + g[k]);
```

```
if i < n then
   if k < i then
      f(k) := f(i),
      f(i) := 1 + f(k),
      k := k + 1
   else
      i := i + 1
   end
end
```

(A)

```
INIT:
   if (i < n)
   {
      if (k < i)
      {
         SAVE_UPDATE(&f[k], f[i]);
         SAVE_UPDATE(&f[i], 1 + f[k]);
         SAVE_UPDATE(&k, k + 1);
      }
      else
         SAVE_UPDATE(&i, i + 1);
      FIRE_UPDATES;
      goto INIT;
   }
```

(B)

Fig. 1. Example of program translation. **(A)** Original program in ASM. **(B)** Correspondent program in C.

```
SAVE_UPDATE(g[p], 1 + f[x]);
FIRE_UPDATES;
goto INIT;
```

`SAVE_UPDATE` is a C macro that, for any update "$f(\bar{x}) := y$", inserts the pair `(&f[x],y)` into the update list. `FIRE_UPDATES` is a C macro that, for each pair `(e,y)` found in the update list, where `e` is a location and `y` is a value, performs the C assignment `*e = y`. Note that in the above example we assumed that, in the generated code, the function f could be implemented as an array, so that the value of $f(x)$ is directly obtained by evaluating `f[x]`.

Although transition rules R_1 and R_2 of "R_1, R_2" are to be executed in parallel, in the compiled code, they are executed in sequence. This is correct because this implementation guarantees that the execution of an update rule does not affect the evaluation of any other rules, since updates are only committed in the following step, after the computation of all values. In the above example the expression "`1 + f[x]`" is evaluated *before* the value of `f[x]` be modified by the first update. Figures 1 **(A)** and 1 **(B)** show, respectively, an ASM program and its translation into C.

4 Main Sources of Optimization

Code of better quality could be produced if the updates could be carried out directly in place instead of being inserted into the update list and delayed until the end of each step. The compiler should be aware of this fact and identify the updates whose aims are not used in the subsequent instructions. These updates can be directly executed in place, with clear benefits to the execution speed.

Of course, there are updates which do not fall into this category. Thus, in order to increase the number of updates that can be executed in place, the compiler should perform *instruction scheduling* which consists of finding an instruction ordering that minimizes the length of the update list. These updates can be more efficiently directly executed in place, with clear benefits to the execution speed.

A second source of optimization is the possibility of avoiding re-evaluation of conditional rule guards, whenever the compiler could infer that their evaluation will reproduce the values of the previous step. The process of determining the best execution point to start the next step is called *branch optimization*.

The third optimization is to find an efficient way to implement the processing of the update list.

Although an optimizer compiler may be smart enough in finding updates which could be done in place, there exist programs whose updates have circular dependencies (see the example program of Section 3). In those programs, not all updates can be done in place, thus some have to be delayed using the update list. For this reason, it is important to implement update lists in an efficient way in order to guarantee execution efficiency.

Furthermore, a set of update rules enclosed by several boolean guard is a sort construction which constantly appears in ASM programs, because it may be used to simulate sequence execution in ASM. For this reason, additional optimization can be obtained by identifying such constructions in programs and compile it to the desired sequence, avoiding the evaluation of unnecessary boolean guards.

5 Instruction Scheduling

Let B be the block $R_1, ..., R_n$, where each R_i is an update rule or a conditional rule. Scheduling B consists of rearranging the rules R_i in order to minimize the number of insertions into the update list. For instance, suppose B is the block "`x := 1, f(x) := 2`". The compilation that preserves this execution order must save the value of `x` before performing the update "`x := 1`" because it is used in the second rule. However, if we exchange the order of these rules, it will not be necessary to save the value of `x` anymore, because this function will only be modified after all of its uses.

The scheduling algorithm operates in three phases.

In the first phase, the block is inspected in order to verify which locations are modified and which ones are read from. For instance, for the transition block of Figure 2 **(A)**, this phase determines, for each rule, which functions are modified and which ones are read, and constructs the table in Figure 3 **(A)**.

In the second phase, we build the conflict graph to the block. The vertices of this graph are the block components. There is an arc from a rule R_1 to a

```
1    f(x) := 1 + g(k),
2    f(y) := x - k * f(1),
3    a := f(1) + f(2),
4    x := g(y),
5    y := f(x)
```

(A)

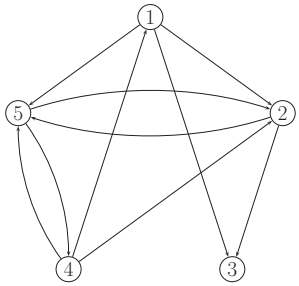
```
3    a := f(1) + f(2),
5    y := f(x),
2    f(y) := x - k * f(1),
1    f(x) := 1 + g(k),
4    x := g(y)
```

(B)

Fig. 2. Instruction Scheduling. **(A)** Original code. **(B)** Scheduled code.

Instruction	Modifies	Consults
`f(x) := 1 + g(k)`	`f`	`x g k`
`f(y) := 2 * x - k * f(1)`	`f`	`y x k f`
`a := f(1) + f(2)`	`a`	`f`
`x := g(y)`	`x`	`g y`
`y := f(x)`	`y`	`f x`

(A)



(B)

Fig. 3. Code Scheduling. **(A)** Information collect. **(B)** Conflict graph – weights have been omitted because they are all equal to 1.

rule R_2 when R_1 modifies any function consulted in R_2. For instance, in the example of Figure 2 **(A)**, there is an arc from instruction `1` to instruction `2`, since `f` is updated in `1` and consulted in `2`. To each arc, say R_1 to R_2, we associate a weight that represents the number of conflicts, i.e., the number of updates occurring in R_1 whose aims are consulted in R_2. The conflict graph of the example of Figure 3 **(A)** is shown in Figure 3 **(B)**.

In the third and last phase, scheduling is actually done. Instruction scheduling is a NP-Complete problem [14], so we must use heuristics to solve it. The kernel of the scheduling algorithm consists of criteriously taking off vertices from the conflict graph until it becomes empty. For each vertex removed from the graph the scheduler generates either a C code to perform the associated update immediately in place or a C instruction to insert the update into the update list. The selection of the next vertex to be removed from the graph obeys the following criteria:

(i) The preferable candidates are those vertices whose output degrees are equal to zero. These vertices may be removed and scheduled in any order for execution.

(ii) When there is no vertex whose output degree is equal to zero, the candidates are those which have the *maximum input degree* in the graph. If there is only one candidate, it is removed and scheduled for execution.

(iii) When criterion 2 produces more than one candidate, the vertex to be

removed and scheduled can be any of those whose output degree is minimum.

Vertices selected by criterion 1 are translated into code to perform the update immediately, whereas vertices determined by criteria 2 and 3 are compiled into instructions that insert update information into the update list of the execution step. The more vertices are selected by criterion 1, the better would be the code produced.

The reasoning behind this heuristics is as follows: criterion 1 has precedence over the others because vertices (updates) which do not affect other rules can be scheduled for execution in place. Criterion 2 is founded on the idea that whenever a vertex is eliminated from the graph, the output degree of some other vertices may be reduced, thus they may become eligible to satisfy criterion 1. Therefore, we should preferably schedule the vertices with the maximum input degree in the hope that a greater number of vertices will have their output degree reduced to zero. Criterion 3 tries to minimize the number of insertions into the update list.

This algorithm, applied to the previous example, would produce the ordering presented in Figure 2 **(B)**, in which only instructions 2 and 5, instead of all of them, cause insertions into the update list. Note that otherwise there would be necessary to perform four insertions.

Another example is the instruction scheduling of the following selection sort algorithm:

```
 1    if mode = 1 and i < n then
 2        k := i,
 3        j := i + 1,
 4        mode := 2
 5    elseif mode = 2 then
 6        if j > n then
 7            mode = 3
 8        else
 9            if f(j) < f(k) then
10                k := j
11            end,
12            j := j + 1
13        end
14    elseif mode = 3 then
15        if k != i then
16            f(k) := f(i),
17            f(i) := f(k)
18        end,
19        i := i + 1,
20        mode := 1
21    end
```

The blocks to be analyzed are B1 (2-4), B2 (9-12), B3 (16-17), and B4 (15-20). Block B1 has three components, the update rules of lines 2, 3, and 4. Block B2 contains the guarded updates of lines 9-11 and the update rule of line 12. Block B3 consists of the update rules of lines 16-17. Block B4 contains the guarded updates of lines 15-18 and the update rules of lines 19 and 20. For blocks B1, B2, and B4, no order exchanging is necessary, since no update modifies the functions used in other components. In block B3 instruction 16 modifies the function `f`, used in instruction 17 and vice-versa. Thus, the update of instruction 16 must be stored in the update list in order to avoid

conflict with instruction 17. Therefore only one insertion into the update list is necessary.

6 Branch Optimization

A transition rule R is compiled into an endless loop such as "`L: R; goto L;`", where `R` is the translation of the instructions in R. This conforms with the ASM theoretical model which defines that transition rules are to be executed continuously. However, if R is a guarded rule such as "**if** g **then** R' **end**", the execution may stop when guard g becomes false, since no further update will be done. This means that R can be advantageously compiled into "`L: if (g) { R; goto L; }`". For instance, the rule:

```
if i < n then sum := sum + f(i), i := i + 1 end
```

can be translated into C code as:

```
L: if (i < n) { sum = sum + f[i]; i = i + 1; goto L; }
```

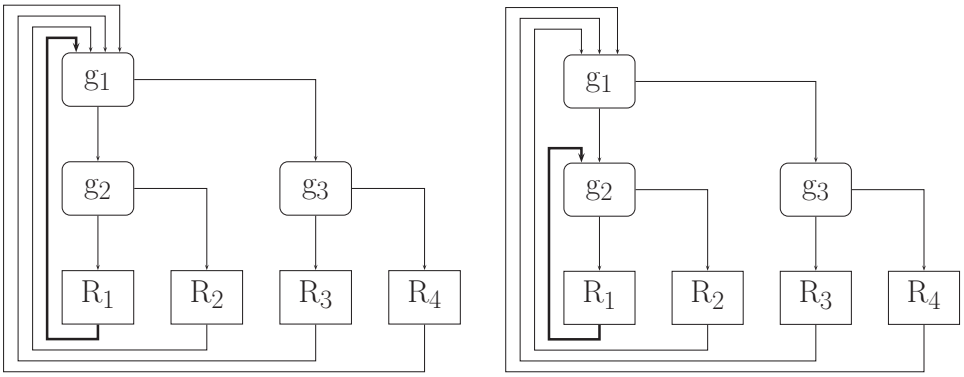
The compiler may infer which paths will be traversed until reaching the execution of each rule `R`i. In addition, by inspecting left-hand sides of update rules, it is possible to know which instructions modify guards that have already been evaluated. These information allows the compiler to determine which guards will necessarily produce the same result as in the previous step, and thus driving the execution through the same path. Therefore the re-evaluation of such guards can be avoided whenever their outcome could be inferred.

Branch optimization improves programs which consist of various nested conditional rules. Consider the transition rule:

```
if g1 then
    if g2 then R1 else R2 end
else
    if g3 then R3 else R4 end
end
```

Suppose `R1` only modifies values of operands of guard `g2`, and `R2`, `R3`, and `R4` modifies those of guard `g1`. This means that, after the execution of `R2`, `R3`, and `R4`, the compiler should generate a goto to the point in the code where guard `g1` is evaluated. However, since `R1` does not modify `g1`, after the execution of `R1` the program could jump directly to the evaluation of `g2`:

```
EvalG1: if (! g1) goto EvalG3;
EvalG2: if (! g2) goto ExecR2;
ExecR1: ... /* Execution of R1 */
        goto EvalG2; /* Avoids re-evaluation of g1 */
ExecR2: ... /* Execution of R2 */
        goto EvalG1; /* Jumps to the evaluation of g1 */
EvalG3: if (! g3) goto ExecR4;
ExecR3: ... /* Execution of R3 */
```



(A) **(B)**

Fig. 4. Control-flow graph **(A)** without branch optimization **(B)** with branch optimization.

```
        goto EvalG1; /* Jumps to the evaluation of g1 */
ExecR4: ... /* Execution of R4 */
        goto EvalG1; /* Jumps to the evaluation of g1 */
```

Control-flow graphs of Figure 4 show a graphic representation of the optimization which has been performed.

7 Update Lists Implementation

This section shows the memory organization used in the generated code and how the C macros `SAVE_UPDATE` and `FIRE_UPDATES` are implemented.

Elements of the update list are pairs of the form `(addr,val)`, where `addr` is the location of the update and `val` is the value to be assigned. We can either organize lists by type, i.e., one update list for each type, or create a polymorphic list, which may contain all updates collected in the step.

In the first approach, all lists, even the empty ones, have to be processed at the end of all steps. The second approach requires the use of an additional level of indirection in order to implement polymorphism. The pair `(addr,val)` should be stored in a structure whose first field is of type `void**` and the second of type `void*`. Therefore, it would be necessary to use indirection in accessing both the address and the value of an update. Firing the update stored in this pair would be implemented as the C command "`*addr = val;`".

For this reason, an element of type integer should be stored, in the object code, as a pointer to integer instead of the integer itself. This implies an additional level of indirection for all accesses to values. Furthermore, all updates would require a dynamic allocation of memory to contain the new value to be assigned, which would turn this approach unbearably inefficient.

A possible solution to reduce this inefficiency of update lists consists of

using "mirror copies" for all values in the ASM specification, i.e., a second memory allocation for all functions whose update information can be inserted into the update list. The functions that will use this memory organization are determined in the instruction scheduling phase.

In this approach, all updates to an element modify its mirror copy and all accesses are from the original copy. At the end of each step, mirror copies of locations in the update list become the original copy and vice-versa. Thus, the `fire` operation consists of turning mirror copies into actual copies for all updated functions. To implement this flipping mechanism we use, for each function, a variable `cur`, whose value can be `0` or `1`. For each function, there will also be a variable `t` containing the value `1` if a flip must be done, or `0`, otherwise. Therefore, the operation `fire` consists of changing the values of `cur` from `0` to `1`, or from `1` to `0`, if the value of `t` is `1`, so that, in the next step, the value of the original copy will be the value assigned in the current step.

For instance, consider the block "`x := y - 1, y := 1 + x * z`", in which one of the updates have to be saved in the update list. Choosing `x` to be saved, this variable must be allocated as a pair original-mirror copies. Therefore, the declaration of `x` in the compiled code will be:

```
struct { char cur, t; int content[2]; } x;
```

The block "`x := y - 1, y := 1 + x * z`" will be compiled into:

```
x.content[1 - x.cur] = y - 1; x.t = 1;
y = 1 + x.content[x.cur] * z;
insertUpdate(&x);
```

The operation `fire` will modify the value of `cur` to point to the updated value. The compiler has to be aware that when more than one update is done to the same location, only one of the updates should prevail. Even when there are more than one update for the same location in the update list, then the value of `cur` can be modified only one time. Therefore, alternating the value of `cur` is forbidden when `t` is equal to `0`. Hence, the value of `t` must be set to zero when the value of `cur` is modified. Fortunately, the test of `t` and the modification of `cur` can be done by only one XOR operation. This approach is interesting because it eliminates the test necessary to the conditional modification of `t` implying a faster execution (see [13], Chapters 2–4).

Thus, `fire` can be implemented as:

```
#define addrE ((char*)e)
...
For each element "e" in the update list
{
    *addrE = *addrE ^ *(addrE + 1); /* cur = cur XOR t; */
    *(addrE + 1) = 0; /* t = 0 */
```

```
}
```

This code returns the complement of `cur`, if `t` is `1`, and returns the original value of `cur`, if `t` is `0`.

8 Evaluation of the Proposed Techniques

This section describes the results of the experiments done to evaluate the optimization techniques proposed in Sections 5 (scheduling), 6 (branch optimization), and 7 (update list implementation). The benchmark programs are the Selection Sort algorithm of Section 5 and an algorithm for finding prime numbers using the Eratostenes Sieve.

Figures 5, 6, and 7 show for each optimization technique the average elapsed time of the execution of each program in the benchmark varying the size of the input. The values were normalized by the smallest time in each experiment.

When comparing the times of execution of non-optimized programs to the times of programs optimized using efficient update lists, we see an improvement of about 65% in the case of the "primes" program and about 57% in that of the sort program. By applying instruction scheduling to the optimized programs we achieved an improvement of about 78% to the first program and about 62% to the second. Applying the branch optimization technique to the scheduled program we obtain an improvement of 1% to the first program and of 23% to the second. We observed that this technique did not show interesting results in the first program because almost all blocks modify some function used in the first guard, so this result was expected.

The application of all the three optimization techniques produced code which is circa 16.6 times faster to the first program and 7.9 times faster to the second program, i.e., an improvement of about 94% and 87% respectively. Note that the greatest improvement was obtained by using scheduling techniques; this indicates that it is worthwhile to perform detailed analysis of the code, in order to reduce the number of insertions in the update list.

9 Conclusion

In this paper we presented optimization techniques which can be used to improve the performance of programs written in ASM-based languages.

We proposed a memory organization that reduces the inefficiency of update lists. The main improvement comes from eliminating memory allocation for all objects inserted into the list. Obviously, it demands more memory, but brings very high improvements when available space is not a problem.

Code scheduling technique allowed the greatest reduction in the execution time, which is reduced in up to 4.5 times. Basically, this technique consists of

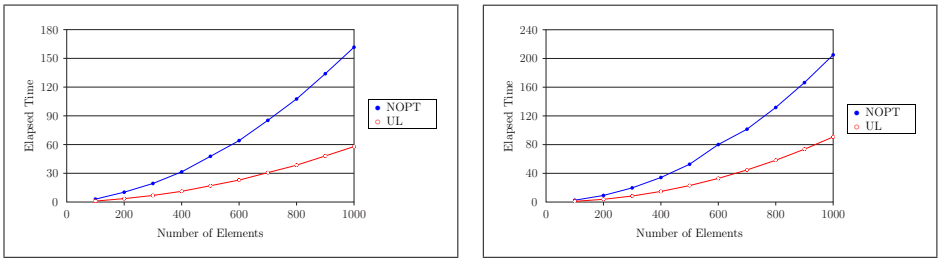


Fig. 5. Elapsed times of non-optimized programs and optimized programs using efficient update lists. **(A)** Primes program. **(B)** Sorting program.

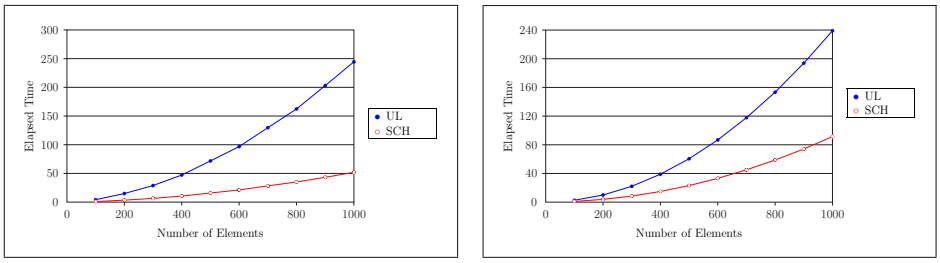


Fig. 6. Elapsed times of programs using update lists and scheduled programs. **(A)** Primes program. **(B)** Sorting program.

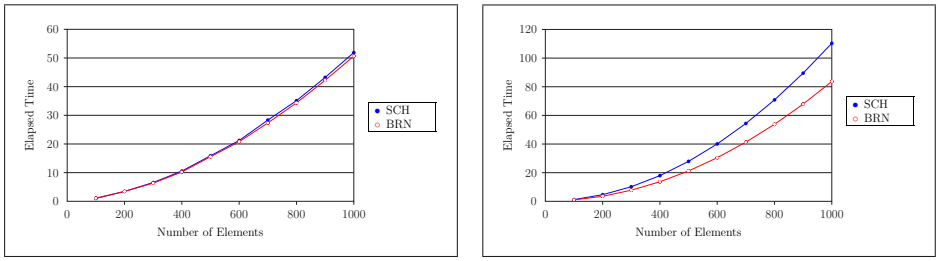


Fig. 7. Elapsed times of scheduled programs and branch-optimized programs. **(A)** Primes program. **(B)** Sorting program.

finding an ordering for instructions in which the number of insertions in the update list is minimum. Since it is an NP-Complete problem, we proposed a heuristic to solve this problem.

Branch optimization attacks one of the sources of inefficiency of ASM specification: the absence of referential locality of programs. Many times, small portions of code depend on many boolean guards, which have to be unnecessarily re-evaluated in many successive steps. Applying this technique showed an improvement of about 23% to one program in the benchmark.

Experiments showed that these techniques, when applied to ASM-based code, may bring impressive improvements in the execution time, reducing it in up to 94%.

References

[1] M. Anlauff, P. Kutter, and A. Pierantonio. Tool Support for Language Design and Prototyping with Montages. In *Proceedings of Compiler Construction (CC'99)*. Springer Lecture Notes in Computer Science, 1999.

[2] D. Bèauquier and A. Slissenko. On Semantics of Algorithms with Continuous Time. Technical Report 97-15, Dept. of Informatics, Université Paris-12, October 1997.

[3] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer, 1995.

[4] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[5] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1997.

[6] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[7] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[8] G. Del Castillo, I. Durdanovic, and U. Glässer. The Evolving Algebra Interpreter Version 2.0. pages 191–214, 1996.

[9] G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

[10] U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[11] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[12] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 295–328. Oxford University Press, 1995.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.

[14] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[15] J. Visser. Evolving Algebras. Master's thesis, Delft University of Technology, 1996.