CASM - Optimized Compilation of Abstract State Machines *

Roland Lezuo
Vienna University of Technology
Institute of Computer Languages
Vienna, Austria
roland.lezuo@tuwien.ac.at

Philipp Paulweber
Vienna University of Technology
Institute of Computer Languages
Vienna, Austria
p.paulweber@gmail.com

Andreas Krall
Vienna University of Technology
Institute of Computer Languages
Vienna, Austria
andi@complang.tuwien.ac.at

Abstract

In this paper we present CASM, a language based on Abstract State Machines (ASM), and its optimizing compiler. ASM is a well-defined (formal) method based on algebraic concepts. A distinct feature of ASM is its combination of parallel and sequential execution semantics. This makes it an excellent choice to formally specify and verify micro-architectures. We present a compilation scheme and an implementation of a runtime system supporting efficient execution of ASM. After introducing novel analysis techniques we present optimizations allowing us to eliminate many costly operations.

Benchmark results show that our baseline compiler is 2-3 magnitudes faster than other ASM implementations. The optimizations further increase the performance of the compiled programs up to 264%. The achieved performance allows our ASM implementation to be used with industry-size applications.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors – compilers, optimization, code generation

Keywords ASM, compilation, optimization, redundancy elimination, parallelism

1. Introduction

ASM is a formal method well suited to formalize semantics of micro-processors [24], programming languages [16] and instruction set simulators [18]. Formal specifications are a precondition for thorough verification of safety-critical embedded systems. We intensively use our ASM implementation (CASM) in a compiler verification project [17] as the formal foundation for the required proofs. Precise machine models for various micro-processors commonly used in embedded systems have been developed. These CASM models can be used to synthesize instruction set simulators. Available tools for ASM have the major drawback that they do not perform well enough to handle industry-size applications. In this paper we introduce an optimizing CASM compiler and present two effective optimizations. Ultimately the compiler is applied to a CASM formalization of the MIPS instruction set to synthesize compiled simulations for industry-size applications.

The remainder of the paper is structured as follows: In section 2 we introduce other implementations of ASM. Section 3 gives an overview of the CASM language and the most important features influencing the compilation. An overview of the CASM compiler is given in section 4 and section 5 describes the optimizations. We report on the performance in section 6. Section 7 discusses future work and section 8 finally concludes the paper.

2. Related Work

ASMs were introduced by Gurevich (originally named evolving algebras) in the Lipari Guide [12]. Core concepts of ASMs are the algebraic state and rules, which describe exactly how the state is changed by means of *updates* applied to the state. Evaluation of a rule itself is side-effect free, a concept introduced in functional programming.

The ideas of ASMs were further developed by Gurevich and others at Microsoft Research resulting in a powerful specification language called AsmL [14]. AsmL is designed to be simple, precise, executable, testable, inter operable, integrated, scalable and analyzable. The language is statically typed, supports object oriented features, has call-by-value semantics and supports exceptions. An efficient compiler for .NET has been developed and the language has been fully integrated into the .NET framework and the Microsoft development environment [5]. The tool environment comprehends *parameter generation* for providing method calls with parameter sets, *finite state machine* generation from an ASM, *sequence generation* for deriving test sequences and *runtime verification* for testing if an implementation performs conforming to the model. The tool environment around AsmL is the most advanced currently available.

One of the most performance critical issues in ASMs is the problem of partial updates. Gurevich and Tillmann discussed the problem in detail and showed how concurrent data modifications can be implemented efficiently [13]. Similar problems occur in version control systems on software merging [20]. Techniques which work only on the delta (the differences) of the data sets inspire optimizations on efficient update implementation in ASMs.

Castillo describes the ASM Workbench in [9]. Similar to CASM he added a type system to his language. The ASM Workbench is implemented in ML[1] in an extensible way. Castillo describes an interpreter and a plugin for a model checker, which allows

* This work is partially supported by the Austrian Research Promotion Agency (FFG) under contract 827485, *Correct Compilers for Correct Application Specific Processors* and Catena DSP GmbH.

LCTES '14, June 12–13, 2014, Edinburgh, UK.
Copyright © 2014 ACM 978-1-4503-2877-7/14/06. . . $15.00.
http://dx.doi.org/10.1145/10.1145/2597809.2597813

[1] `http://en.wikipedia.org/wiki/Standard_ML`

to translate certain restricted classes of abstract state machines to models for the SMV[2] model checker.

Schmid describes compiling ASM to C++ [23]. The compiler uses the ASM Workbench language as input. He proposes a double buffering technique avoiding implementing update sets at all. This approach is limited to parallel execution semantics only, though. CASM uses a so called *pseudo state* (more details are in section 4.3.3) to implement *update sets* efficiently.

Schmid also introduced AsmGofer in [22]. AsmGofer is an interpreter for an ASM based language. It is written in the Gofer[3] language (a subset of Haskell) and covers most of the features described in the Lipari guide. The author notes however that the implementation is aimed at prototype modeling and too slow for performance critical applications.

Anlauff introduces XASM, a component based ASM language compiled to C [3]. The novel feature of XASM is the introduction of a component model, allowing implementation of reusable components. XASM supports *functions* implemented in C using the *extern* keyword. CASM does not feature modularization, but can be extended using C code as well. XASM was used as the core of the gem-mex system, a graphical language for ASMs.

Farahbod designed CoreASM, an extensible ASM execution engine [10]. The CoreASM project is actively maintained and has a large user base. The CASM language is inspired by the CoreASM language, but over time they have diverged significantly.

Praun, Schneider and Gross presented an algorithm for load elimination in the presence of side-effects, concurrency and precise exceptions [21]. Even the most conservative variant without side-effect and concurrency analysis can eliminate up to 55% of the loads and whole program analysis in an ahead-of-time Java compiler can increase the reduction up to 70%. Barik and Sarkar achieve performance improvements up to a factor of 1.76 in a just-in-time compiler for the parallel language X10 in the Jikes RVM applying interprocedural load elimination [4]. Our optimizations also aim at redundancy elimination in concurrent (parallel execution) context. ASM's partial updates can be treated in a similar way as side-effects. This work demonstrates the high levels of redundancy elimination possible in such systems.

3. The CASM Language

This section gives a brief description of ASM based programming languages in general and highlights features specific to CASM. An excellent introduction to the formal semantics of ASM languages is given by Börger and Schmid in [7]. More details on the CASM language can be found in [19].

3.1 Semantics of ASM Based Languages

The core concepts of ASM are the state and the transactional semantics of language statements. Based on algebraic concepts the *state* of the machine is modeled using *functions*. The *function* is a mathematical object and has a domain and a range. A null-ary *function* is, roughly speaking, a global variable in C-like languages. N-ary functions can best be thought of as hash-maps. In contrast to C-like languages, *functions* are always defined on their whole domain. *Functions* take the special value *undef* on arguments for which no value has been defined explicitly. The name of a *function* together with concrete arguments is called a *location*.

Statements of an ASM language are evaluated using the current *state* of the machine (defined by the total of its *functions*). The effects of each statement (called an *update*) will affect the *next* state however. All statements of a block are executed in parallel and their updates are *merged* (union) into a so called *update set*. Applying the resulting *update set* to the global machine state is called a *step* of the machine.

```
1 function x : -> Int initially { 2 }
2 function y : -> Int initially { 3 }
3
4 rule swap =
5 {
6   x := y
7   y := x
8 }
```

Listing 1. Swapping of two Values (Parallel Semantics)

```
1  function t : -> Int initially { undef }
2  function x : -> Int initially { 2 }
3  function y : -> Int initially { 3 }
4
5  rule swap =
6  {
7   {|
8     t := x
9     x := y
10    y := t
11  |}
12  print t
13 }
```

Listing 2. Swapping of two Values (Sequential Semantics)

Listing 1 shows a code snippet swapping the contents of two null-ary functions utilizing the parallel execution semantics (denoted by braces) of CASM. The *update set* produced by the first update statement (line 6) is the set *{ x=3 }*. The second update statement (line 7) is evaluated using the same state as the first one, so it produces the update *{ y=2 }*. When leaving the block the updates are merged into the *update set { x=3, y=2 }*.

Parallel updates to the same *location* (non-empty set intersection) are a runtime error (a so called *inconsistent update*). CASM aborts the execution but we plan to invoke an error handler provided by the user in a future version.

TurboASM [11] extend ASM with the concept of sequential composition. Statements in a block using sequential execution semantics (denoted `{|` and `|}` in CASM) apply their updates before the subsequent statement is evaluated. The updates produced by each of the statements can overwrite each other in the resulting *update set*. When leaving the sequential block the original state (the one valid when entering the block) is restored. The intermediate states only exist temporarily (not the *update set* though). The rationale for this behavior is that the machine makes a *virtual step* after each statement in a sequential block.

Listing 2 illustrates this behavior. Inside the block with sequential execution semantics the values of the functions *x* and *y* are swapped using a temporary (function). Because of the sequential block the update in line 10 is evaluated in an intermediate state where the update to *t* (line 8) has been applied. The update set produced by the whole block therefore is *{ t=2, x=3, y=2 }*. The changes have not been committed to the global machine state though. The `print` statement in line 12 will see the initial value *undef* when reading the function *t*.

Procedures are called *rules* in ASM. A distinct *rule* (top-level) is invoked on program startup. When the top-level rule returns the machine makes a *step* (and applies the *update set* to the machine state). The top-level rule will then repeatedly be executed until the program explicitly terminates.

[2] `http://www.cs.cmu.edu/~modelcheck/smv.html`

[3] `http://web.cecs.pdx.edu/~mpj/goferarc/index.html`

3.2 CASM Specifics

The CASM language is statically typed (to ease programming a type inference system is implemented). Basic types are integer, sub range integer, float, rational, reference to rule and string. Custom types can be constructed using enumerations, tuples and lists. All *functions* and rule arguments must be explicitly typed. CASM performs no implicit type conversions. Builtins are used to convert boolean and enumeration types to integer representations. Integers can also be converted to boolean and enumeration types (for which a range check is performed). Variables (bindings created by *let* rules) may be typed explicitly but if they are not, their types will be inferred.

The original ASM specifies call-by-name semantics for rule invocation (procedure call). Call-by-name can be compiled using thunks [6], but doing so is not very efficient. We introduce a CASM specific *call* rule implementing call-by-value semantics.

The common control-flow statements, *i.e.* if-then-else, case, direct and indirect (using rule references) subroutine invocation (*call* rule) are available. As all state is global in CASM the only variables are values (expressions) bound to names (*let* rule).

CASM offers a set of built-in functions to operate on lists and stacks and can be extended with custom C functions. All built-ins must be side-effect free though.

3.3 Loops

Due to CASM's transactional semantics a loop counter can not be implemented. Commonly known loop constructs are therefore not part of the language. One way to express iterated execution is to utilize the property that the top-level rule will be executed repeatedly. This also solves the loop counter issue as after each iteration of the top-level rule the *update set* is applied to the state, which allows to model a loop counter. The semantics of the *forall* rule (*forall var in range*) is that the body is executed for each value (assigned to *var*) of the *range* in parallel.

The other way to implement a loop is by means of the *iterate* rule which sequentially composes the result of the loop body until the *update set* of a single iteration is empty. Iterate basically searches the fix-point of its body.

3.4 Notes for Implementors

The key issues to deal with when implementing an ASM language are: infinite domains for functions, n-ary functions, transactional and parallel semantics, intermediate and temporary states. One needs to implement a *lookup* mechanism for *locations* which is relative to the current state. *Updates* have set semantics. They need to be *merged* and must be checked for *inconsistency*. On the other hand the language is side-effect free.

4. The CASM Implementation

We developed a CASM interpreter and a CASM to C source-to-source compiler. Both the interpreter and the compiler are implemented in C++ and share the frontend and some parts of the runtime system.

4.1 Frontend

The abstract syntax tree (AST) is built using a Yacc parser. Type inference is performed on the AST using *apriori* known types (*functions*, built-ins, arguments) and propagates them through the AST. For all untyped variables *inferred* types are calculated. When a fix-point is reached the computed types are checked for completeness and consistency. The only difficulties are arising from the special value *undef* (which is compatible to all types) and empty list constants (as the type of the list element is undetermined). The typed AST can either be interpreted by a recursive AST interpreter (CASM-i) or can be compiled. In the remainder of this section we describe the compiler.

4.2 Backend

The typed AST is used by the code generator to emit low-level C code (source-to-source translation). For each rule of the CASM program a distinct compile unit is emitted which allows parallel compilation. The emitted code is designed to allow good optimizations by a C compiler.

Code generation is (with exception of *lookups* and *updates*, see 4.3.4) straight forward. For each CASM rule the compiler emits a C function, which is split when the compile unit becomes too large. CASM control-flow constructs (i.e. *if*, *case* and *call*) are mapped to their C counterparts. The *forall* and *iterate* rules are translated to corresponding C loops. All iterations of a *forall* loop are considered to be executed in parallel, while the iterations of an *iterate* rule behave like being executed in sequential execution mode. The code generator adds a surrounding block with appropriate semantics (if needed) and merges the updates produced by each iteration accordingly.

CASM variables (i.e. let rule and rule arguments) are mapped to (scoped) local variables. Expression trees are translated in post-order fashion (this is possible as CASM is side-effect free). The temporaries are stored in local variables called *registers*. Each register has a unique name derived from a numbering of the AST nodes. Our assumption is that the C compiler will be able to very efficiently compile such code.

4.3 Runtime

This section describes the implementation of an efficient runtime system for an ASM language. This section also motivates the optimizations.

4.3.1 Dynamic Memory Allocation

Only *functions* and updates need to be allocated dynamically. Due to the transactional semantics of ASM languages the life-span of an update is exactly one *step* of the machine. A pre-allocated memory pool is used to store updates until a *step* is made and all updates are committed to the function storage. This pool can simply be reused in subsequent steps (dump-allocation). The runtime therefore has virtually no memory management overheads.

4.3.2 Storage for CASM Functions

Set operations are necessary to properly implement *functions*. All *locations* not explicitly defined otherwise have the special value *undef* (demanding an *is-element-of* set operation). A distinct hash-map (with linear probing) is used as storage for each *function*. The function arguments are concatenated to form the key. Each slot of the map has two special properties, *undef* and *branded*. The *undef* property is set if the *location* has the special value *undef*. An update may set a previously defined *location* to *undef*, so such *locations* need to be tracked explicitly. A slot is *branded* when its corresponding *location* is accessed for the first time. (*Branding* allows to use other default values than *undef*, CASM supports this feature). The runtime uses the slot's address, which must be guaranteed to be stable, as a unique identifier.

After each *step* of the machine the hash-map can safely be enlarged should the load factor have become too large. In the rare case that during a single *step* the hash-map would overflow, additional memory is allocated. In-between the next machine *step* the hash-map is resized and the overflow memory gets merged.

If a sub range integer type is used for the domain of a CASM function, an array is used as *function* storage instead of a hash-map (for reasonable sizes of the domain). An additional bit is needed to keep track of the special value *undef*.

```
{
   stmt1
   {| stmt2 ; stmt3 |}
   {| stmt4 ; stmt5 |}
}
```

Listing 3. Interleaving PAR/SEQ

4.3.3 Updates and Pseudo States

Due to the interleaving of parallel and sequential execution semantics the state used to evaluate a statement and the state affected by its updates are in general not equal [11]. Listing 3 illustrates the problem. $stmt_1$ and the sequential blocks containing $stmt_2$ and $stmt_4$ are in a parallel block. Therefore they are evaluated under the same state S_0, their updates however are applied to different states. While updates produced by $stmt_1$ are applied to the S_0, updates produced by $stmt_2$ are used to create a *temporary* state S_1. The sequential composition with $stmt_3$ may modify updates produced by $stmt_2$ and only the resulting update set will be applied to S_0. The same situation arises with $stmt_4$ and $stmt_5$. As e.g. $stmt_4$ may contain a nested parallel block a tree-like structure of states is created. The nesting of update sets is very similar to nested transactions in software transactional memory (STM) [1]. The major difference is that an STM transaction aborts when reading an object for which a commit is pending while in ASM read access can never fail. Multiple updates to the same *location* in a parallel context is a runtime error (*inconsistent update*) in CASM.

Our assumption is that the number of updated *locations* (in a single ASM step) is much smaller than the whole state of the program. We therefore do not duplicate the state but keep track of all updates produced so far in a data structure called *update set*. When looking up a *location* the runtime has to query the *update set* for updates affecting the current state (due to sequential execution semantics).

We use the notation of *pseudo state* to keep track of updates affecting the current state. The *pseudo state* is a counter which is increased (at runtime) when a block with *different* execution semantics is entered. When a block is left (and control-flow returns into a block with *different* execution semantics) the *update set* is merged into the *update set* of the surrounding block. This is a serialization of the (partial) parallel execution semantics of ASM. Initially the system starts in parallel execution state, so *pseudo state* 0 denotes a block with parallel execution semantics. When entering a block with sequential semantics *pseudo state* will be increased to 1. By construction this counter is odd when executing a block with sequential execution semantics and even when in parallel mode.

The *update set* is implemented as a hash-map. The keys are 64 bit values, the lower 16 bits are the *pseudo state* of the block the update originates from, the remaining bits are the lower bits of the slot used to store the *location*. (This limits the number of nested states to 65536. The number of locations is limited to 2 to the power of 48. The maximum memory utilization of CASM therefore is 256 TiB, which is sufficient for any realistic application.)

Additionally the slots in the *update set* are forming a linked list with the latest update being the head. This property is used when merging update sets. Figure 1 shows the update set data structure.

4.3.4 Lookup and Update

A lookup for a specific *location* first needs to query the *functions* storage to acquire the address of the slot. This address and the current *pseudo state* are used to query the *update set* for any updates to this location which may be visible in the current state. By construction of the *update set* the corresponding keys can be efficiently calculated using the current key. The sequential states

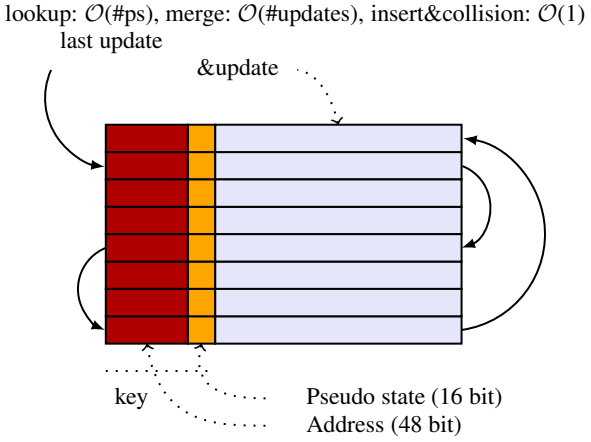

Figure 1. Update Set

are all odd numbered *pseudo states* with a number that is lower than the current one. The complexity of this operation is linear in the number of active *pseudo states* (dynamic nesting depth of parallel and sequential blocks).

An update also needs to query the functions storage to acquire the address of the slot corresponding to the location. This address and the current *pseudo state* form the key for the *update set*. If the slot in the *update set* already contains a value the further behavior depends on the current *pseudo state*. In sequential execution mode (odd pseudo state) the value will be overwritten, in parallel mode an *inconsistent update* error is triggered. The complexity of the *inconsistency* check is constant.

4.3.5 Merging of Update Sets

When leaving a block (with *different* execution semantics) the list property of the *update set* is exploited to efficiently merge all updates into the surrounding *update set*. The list is traversed backwards until the first update not belonging to the current update set is found (encoded in the lower 16 bits of the key). All updates are removed from the *update set* and re-inserted with the *pseudo state* part of their key reduced by one. Merging of *update sets* produced by sequential blocks may trigger inconsistent update errors as they are re-inserted into an *update set* with parallel execution semantics. The complexity of merging is linear in the size of the *update set* to be merged.

5. The Optimizing Compiler

In this section we describe the analyses and transformations performed by the optimizer. The optimizer is divided into multiple passes. Analyses only identify and mark opportunities while the transformations actually perform the changes.

5.1 Lookup and Update Elimination

The hash-maps used to implement the *update set* and *functions* are obviously very expensive in terms of performance. In this section we describe two optimizations called *lookup elimination* and *update elimination* that aim to reduce the number of hash-map operations. The first observation is that lookups from a parallel execution context will always retrieve the same value (for same *locations*). In such situations only the first lookup needs to query the function storage and the *update set* to retrieve the value. The second observation is that, in sequential execution context, updates and lookup behave like local variables in the language C.

```
{
  if X(3) = 3 then
    skip
  if X(3) = 4 then
    skip
}
```

```
{
  local X_3 = X(3) in
    if X_3 = 3 then
      skip
    if X_3 = 4 then
      skip
}
```

Table 1. Redundant Lookup and its Elimination

```
{|
  X(4) := foo
  if X(4) > 0 then
    skip
|}
```

```
local L_1 = foo in
  {|
    X(4) := L_1
    if L_1 > 0 then
      skip
  |}
```

Table 2. Preceded Lookup and its Elimination

```
{|
  X(5) := foo
  X(5) := bar
|}
```

```
{|

  X(5) := bar
|}
```

Table 3. Redundant Update and its Elimination

The idea is to introduce so called *local locations*. That is a rule-local storage which will be used by optimized lookup and update code. Once fetched, the *local location* can be used by subsequent lookups without the overheads of a hash-map. Table 1 illustrates the basic idea (*local* is not a valid CASM keyword).

Another pattern which allows the elimination of a lookup arises from updates (to the same location) preceding the lookup in a sequential context. In this case the value to be retrieved is known already and can be propagated instead of performing an expensive lookup. We call this pattern a *preceded lookup* for an example see table 2.

Update elimination tries to reduce the number of updates stored in the *update set*. If a specific *location* is updated multiple times in a sequential context, only the last update will be committed to the state. All preceding updates can safely be omitted. See table 3 for an example.

5.2 PAR/SEQ Control Flow Graph

We use an extension to the control flow graph (CFG) called the PAR/SEQ CFG to capture the nested parallel and sequential execution semantics. The nodes of this CFG are the instructions as they will be generated by the code generator, therefore it captures the semantics of the serialized statements. *Forall* rule bodies are executed in parallel leaving *iterate* as the only loop construct. We currently treat *iterate* as a black-box, like a *call*, hence our PAR/SEQ CFG is cycle free.

First we describe the generation of the PAR/SEQ CFG and discuss its properties later.

The PAR/SEQ CFG is generated from the AST representation of a single rule. Each node has a unique label, a type (e.g. IF, UPDATE, expression), the execution context (parallel or sequential) and its state nesting depth. The state nesting depth is a simple counter which is increased when entering a block and decreased when leaving (very similar to *pseudo states* described in 4.3.3). Our extension is to add synthetic nodes into the CFG when entering or leaving a block and when the control-flow of if-then-else merges. The state nesting depth of the synthetic nodes is the state nesting depth of the containing block.

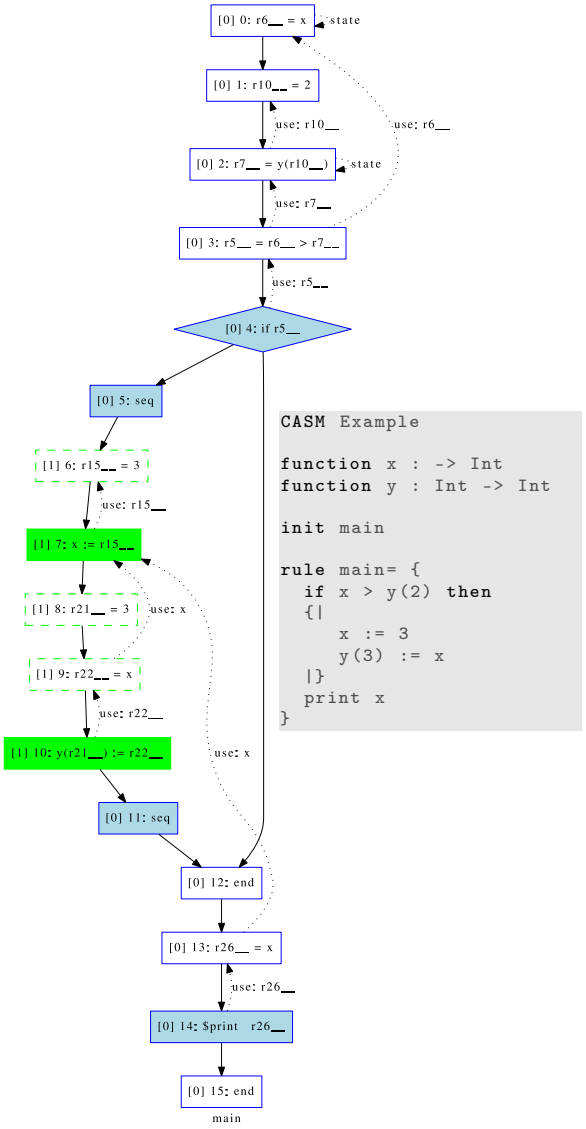

Figure 2. PAR/SEQ CFG (with use/def)

In figure 2 the PAR/SEQ CFG generated by the compiler and the corresponding CASM program is shown. Nodes filled with color correspond to CASM statements and white nodes result from expressions. Blue encodes parallel execution semantics and green is sequential. The nodes are labeled with their state nesting depth (in square brackets) followed by a unique id and their content. Examples for synthetic nodes are nodes number 5, 11 and 12. Results of the use/def analysis are shown as well and will be explained later.

5.3 PAR/SEQ Use/Def Analysis

We use a modified version of the classic use/def analysis called PAR/SEQ Use/Def to identify and categorize state lookups. It is based on the results of a state-unaware reaching-definition analysis [2]. Variables can be treated very simply (CASM is side-effect free, variables are bound and never updated). A lookup is the occurrence of a *function* name on the right side of an expression node. For each lookup its local definitions are considered. If there is no local definition the (expensive) hash-map operation must be performed, we call this a *state-lookup*. The occurrence is marked as such.

```
1  function x : -> Int
2
3  rule foo =
4
5  {|
6    print x  // 1 lookup
7    x := x+1 // 1 redundant update, 1 redundant lookup
8    x := x*x // 1 update, 2 preceded lookups
9  |}
```

Listing 4. Original Source

```
1  function x : -> Int
2
3  rule foo =
4  local L_1 = x in          // 1 lookup
5    {|
6      print L_1
7      local L_2 := L_1+1 in
8        x := L_2*L_2        // 1 update
9    |}
```

Listing 5. After Optimization

If there is exactly one definition we mark the lookup as *local-lookup*. Such lookups are candidates for the lookup elimination pass.

Multiple definitions are not analyzed any further. At the point the control-flow merges a virtual node performing a *pseudo-definition* of the *location* will be added by the lookup elimination pass (similar to phi nodes in SSA). Multiple subsequent lookups to the same *location* therefore see only one definition and can be further optimized.

5.4 Lookup Elimination

For a *local-lookup* the decisive question is: what is the execution semantics of the inner-most block containing the definition **and** the use? The analysis discovers all paths in the PAR/SEQ CFG and records the state nesting depth of each node. The nice property is that the node with the smallest number is their common inner-most block (called *common state*). This property holds because the synthetic nodes act like virtual instructions ensuring that each path of the CFG goes through its containing block. In a way this encodes the least common dominator into each path.

If the *common state* has parallel execution semantics the effects of the definition is not visible. These kinds of lookups are marked as *state-lookup* which are handled separately. If the *common state* is sequential and there is only one path in the CFG the *location* is promoted to a local one. We call this pattern a *preceded lookup* and the transformation is shown in table 2.

For each *location* marked as *state-lookup* the number of occurrences in a rule is counted. If there are more than 2 occurrences the *location* is speculatively promoted to a local one and the lookup is hoisted to the outer-most block possible. The scope of variables used to calculate the *location* are boundaries for hoisting. This transformation is speculative because a lookup from a rarely executed path can be moved to a more frequently executed one.

5.5 Update Elimination

Updates in the same sequential block are considered for elimination. The nodes of each sequential block of a rule are traversed backwards and updates to *locations* are recorded. When there are multiple updates to the same *location* all but the last one are removed. The generic pattern is shown in table 3.

5.6 An Example

We want to illustrate the effects of theses optimizations on a small example. Listing 4 shows a small CASM rule which prints the value of the function x and afterwards updates it to the value $(x + 1)$ to the power of 2. For printing the value a lookup is performed (line 6). This lookup is the very first lookup of that *location* in this rule, so it can't be eliminated. In line 7 the value of x is increased by one. Again a lookup is performed, but this one is redundant and can be eliminated. The function x is then updated to contain the incremented value. This update is followed by another one and can therefore be eliminated. In line 8 finally the square is calculated and x is updated to the new value. To calculate the square value

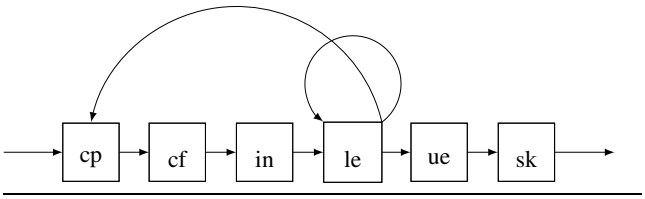

Figure 3. Compiler Passes

x is looked up twice. Theses lookups are preceded ones, as x was updated in line 7.

The optimizing compiler rewrites the source as given in listing 5 (though this isn't valid CASM syntax). The only lookup of x is now performed in line 4. This value (L_1) is used for printing (line 6) and to compute another *local location (*L_2*)* in line 7. This *local location* is used to eliminate the preceded lookups in line 8. Line 7 does not update x any more. The update to the *local location* does not involve the *update set* and is therefore a very cheap operation. In line 8 the two lookups are replaced by usage of the local location L_2, which does not involve *update set* operations.

In this small example the number of lookups was reduced from 4 to 1 and the number of updates from 2 to 1.

5.7 Supporting Optimizations

Lookup and update elimination strongly depend on i) exact analysis results for *all* paths ii) *locations* to be known at compile time, which is achieved by inlining.

The semantics of the *call* rule allows inlining by replacing the invocation with the AST tree of the inlined rule. Rule arguments must be evaluated (call-by-value) which can be achieved by adding *let* nodes to the AST. Maybe the names of an inlined rule's local variables (*let*) need to be renamed.

Because the CASM language is side-effect free, the analysis framework is able to perform constant folding for all built-ins. Using C macros we are even able to reuse the implementation in the analysis framework. Due to constant propagation a lot of dead code is identified and is removed as well.

Figure 3 gives an overview of the schedule of all transformations. Constant propagation (cp) and constant folding (cf) are executed before the inliner (in) is invoked. They may resolve indirect calls so the inliner is more effective. Afterwards lookup elimination (le) is performed until a fix point is reached. The reason for iterating lookup elimination is that the added *pseudo-definitions* may enable further optimizations. Elimination of *preceded lookups* may have propagated constants and therefore a fix point of those 4 optimizations is searched. Finally update elimination (ue) is performed and lookups, which have been hoisted to the beginning of the rule, are sunk (sk) to the least common dominator of all their (remaining) uses.

6. Evaluation

6.1 Baseline Compiler

In this section we evaluate the quality of the baseline compiler. For this purpose we compare it to other available implementations of ASM based languages, namely CoreASM and AsmL. CoreASM is an interpreter written in Java while the AsmL language is compiled to .NET code. A small suite of programs each stressing a different implementation detail of ASM languages has been implemented for each language.

The *bubblesort* program (a very naive implementation of the well known sorting algorithm) performs many steps with small update sets. It aims to benchmark the effectiveness of applying update sets to ASM functions. *Fibonacci* uses dynamic programming to calculate the well known numbers. It benchmarks rule invocation (recursive) and has a moderate size of the update set. *Quicksort* (the sorting algorithm) makes heavy use of sequential execution, although the update sets are very small. The *sieve* program is an implementation of Eratosthenes' famous prime number sieve. This program heavily stresses the implementation of the update set, everything is executed sequentially producing large update sets. The benchmark program *gray* calculates Gray codes for a given word length. It is the program with the most output and a mix of sequential execution, rule invocation and numeric operations. *Trivial* is the trivial program, immediately exiting without any operation. It is used to measure startup overheads of the various implementations.

The performance of the various implementations varies a lot. We use small data sets for the interpreters and larger sets for the compilers to have measurable execution times.

For benchmarking we use the CASM compiler (rev. 1a092c) and gcc 4.7.2 (as shipped with Ubuntu 12.10). We do not perform any CASM specific optimizations and disable optimizations of the C compiler (-O0 flag). The CASM interpreter (CASM-i) is the same version as the compiler.

The CoreASM engine version used is 1.5.6-beta using the command line driver Carma 0.7.3 (latest release). We executed CoreASM using Java 1.7 with the 64 bit Server VM (23.7-b01).

Microsoft's AsmL implementation compiles to .NET code and is freely available on `http://asml.codeplex.com/`. We downloaded version 80132 and followed their build instruction using Visual Studio C# 2005 Express Edition.

The benchmarks involving small data sets were executed on a Core i7-Q820 @1.73 GHz with 8 GiB memory under 64 bit Ubuntu 12.10. For the large data sets a dual boot system (Core i7-2600k @3.4 GHz, 8GiB memory) using 64 bit Windows 7 Enterprise SP1 and 64 bit Ubuntu 13.10 was used. We report on the average of 10 runs and started the AsmL binary once before the benchmark to exclude overheads induced by the .NET framework [4].

Our own implementation of a CASM interpreter (CASM-i) is designed to have very low startup times and is used to execute small programs only. It is used in the compiler verification project. The baseline compiler is a magnitude faster than the interpreter (up to 60 times) which is a good indicator that the baseline compiler performs well.

When it comes to performance CoreASM is clearly inferior to the other implementations. Programs compiled by our compiler perform up to 2500 times better and even our interpreter is one order of magnitude faster. The focus of CoreASM are high level models though.

The AsmL results are varying a lot. For *fibonacci* performance is on par with the CASM compiler (still 35% slower, though). But *fibonacci* is also the benchmark putting the least pressure on

[4] `http://msdn.microsoft.com/en-us/library/cc656914(v=vs.110).aspx`

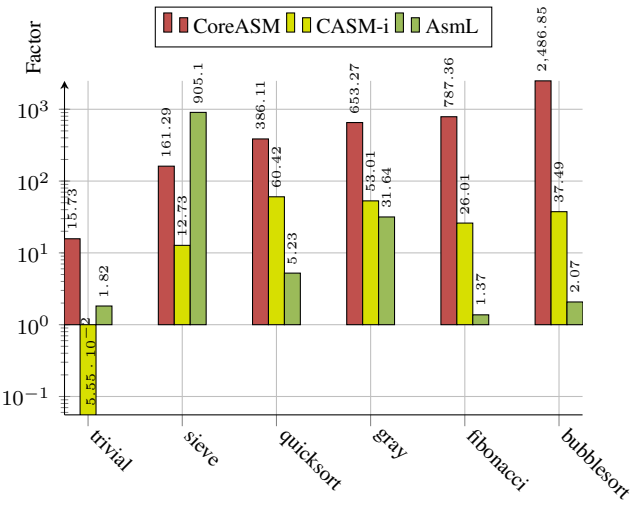

Figure 4. CASM relative Performance (Compiler as Baseline, smaller is better, log-scale)

ASM specifics. It uses mostly recursive function invocation with a comparably small update set. *Bubblesort* is slower by a moderate factor of 2 while *sieve* is slower by a factor of 900. A detailed examination showed that AsmL has quadratic runtime for increased sizes of the sieve. The main difference in the two programs is that *bubblesort* executes a large number of machine steps each with a small update set, while *sieve* exactly executes one step. The update set produced by *sieve* is quite large (the whole array) and a lot of updates need to be merged sequentially. This indicates that AsmL is not optimized for this case and agrees with the observed behavior of *quicksort* (small sequential update sets) and *gray* (moderate sized sequential update sets). Overall the performance of AsmL compiled programs is significantly lower than programs compiled by the CASM compiler.

Figure 4 shows the relative performance (with CASM compiler being the baseline) of the 4 implementations, please note the logarithmic scaling of the y axis. Numeric values are found in table 4. The CASM baseline compiler is by far the best performing ASM implementation.

6.2 Optimizing Compiler

In this section we evaluate the effectiveness of lookup and update elimination and investigate the scalability of the CASM compiler. We compare the performance data of our baseline compiler with the optimizing version. The other ASM implementations are simply not capable of executing programs of the desired size.

6.2.1 The Application

To create a large realistic benchmark we translate binary MIPS programs into a CASM representation and compile them to native code (that is a kind of compiled simulation). A Python script performs a very simple basic block analysis and a CASM rule is emitted for each identified basic block of the program. We add a semantic model of the MIPS architecture originally developed for compiler verification and provide a top-level executing the program's basic blocks. The basic structure of a program generated this way is presented in listing 6.

The benchmark programs are taken from the well known MiBench [15] suite. As our optimizations perform aggressive inlining we only want to optimize the kernel of the applications to

		small data sets					large date sets				
	trivial	sieve	quicksort	gray	fibonacci	bubblesort	sieve	quicksort	gray	fibonacci	bubblesort
CASM	0.0865	0.0857	0.0842	0.0882	0.0854	0.0859	0.0822	0.586	0.7702	3.0436	2.5458
AsmL	0.1292						74.39	3.0628	24.3702	4.1752	5.2748
CASM-i	0.0048	0.10	0.0212	0.2287	0.0107	0.0466	1.05	35.41	40.83	79.17	95.43
CoreASM	1.3604	13.82	32.51	57.61	67.24	213.62					

Table 4. Execution Time CoreASM, AsmL, CASM

```
enum FieldValues = { FV_RT, FV_IMM, FV_RS, ...
function BLOCK : -> Int
function GPR : Int -> Int

function (static) PARG: Int * FieldValues -> Int
  initially {
    [0x80001000,FV_RT] -> 28,
    [0x80001000,FV_IMM] -> 32769,
    [0x80001000,FV_RS] -> 0,
    ...

function (static) BASICBLOCK: Int -> RuleRef
  initially {
    0 -> @bb_0,
    701 -> @bb_701,
    ...

rule andi(addr : Int) =
let rs = PARG(addr, FV_RS) in
let rt = PARG(addr, FV_RT) in
let imm = PARG(addr, FV_IMM) in
  call write_reg
    (rt, BVand(32, GPR(rs), BVze( 16, 32,imm)))

rule bb_0 =
{|
  BLOCK:=630
  call bb_call(@lui, 0x80001000)
  call bb_call(@bb_bltzal, 0x80001004)
  call bb_call(@addiu, 0x80001008)
|}

rule run_program =
{|
    call (BASICBLOCK(BLOCK))
    if trapped then
      program(self) := undef
|}
```

Listing 6. Compiled Simulation in CASM

keep the increase in code size small. Our code generator can instrument the code to collect profiling information measuring the total execution time of each CASM rule (including time spent in invoked rules). Applying a simple heuristic all rules contributing at least 1% to the total runtime have been selected for optimization. Our assumption is that the effects on code size by inlining are small but the achieved effect (in terms of performance gains) large.

Lookup and update elimination work best on large rules so their impact should be high if the frequently executed rules are large and low for small rules. Figure 5 depicts the contribution of a rule's execution time to total program execution time (bars) and their size in LOC (crosses) for the rijndael program. (The rule contributing 100% to total program execution time is the top-level rule, the second bar is a dispatching rule. All further bars correspond to basic blocks or instructions.) Note that two of the most contributing rules each have 1000 LOC. We expect to see a high impact of lookup and update elimination for the rijndael program. Figure 7 on the other hand shows that the patricia program has many small rules and the first large rule does not contribute much to total execution time. A high impact can not be expected. Figure 6 shows the same diagram for the dijkstra program. Medium sized

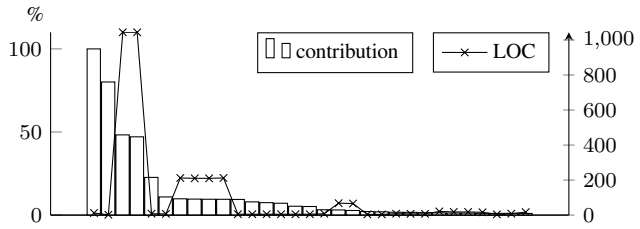

Figure 5. Rule Contribution and Size - Rijndael

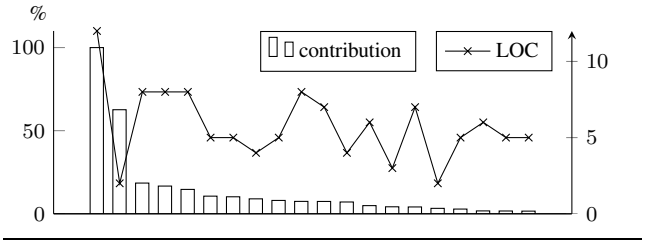

Figure 6. Rule Contribution and Size - Dijkstra

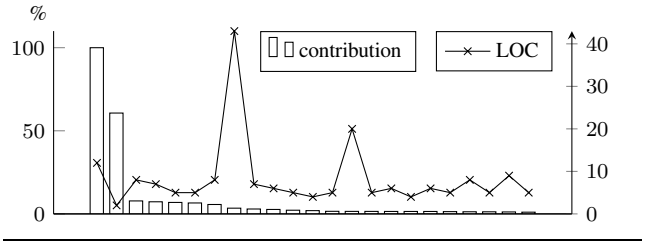

Figure 7. Rule Contribution and Size - Patricia

rules with moderate contribution. We expect our optimizations to have an impact for this program.

6.2.2 Results

We use 3 configurations of our compiler in this evaluation. Baseline is without CASM specific optimizations and without optimizations of the C compiler (-O0). The configuration titled *O0* has CASM specific, but no C compiler optimizations (-O0). *O3* has CASM and full C compiler optimizations (-O3). The benchmarks were executed on Xeon E5504 @ 2.00GHz with 8GiB memory (on the Infragrid cluster [5]). We used gcc 4.4.7 on a Red Hat Enterprise Linux Server release 6.4 for compilation. Due to the shared nature of the cluster we report on the best of 10 runs here. MiBench's small data sets have been used for all but the search benchmark.

Table 5 lists for each benchmark program the total number of rules and the number of rules optimized as well as the total number

[5] `http://hpc.uvt.ro/infrastructure/infragrid/`

	rules		optimizations		
	opt	total	cp	lookup	update
basicmath	30	4097	1440	236	22
bf	43	1226	8060	889	451
crc	17	3501	416	56	7
dijkstra	20	5455	494	52	1
patricia	23	5864	761	150	0
qsort	21	5393	720	65	1
rawcaudio	38	3293	656	65	1
rawdaudio	29	3293	656	65	1
rijndael	32	3431	42452	4394	2864
search	28	3239	2086	274	5
sha	26	3291	2840	381	3
susan	29	5337	7570	1192	224
toast	23	7812	6803	885	53
untoast	40	7812	1840	264	17

Table 5. CASM Optimizations

	LOC casm	w/o opt sec	full opt sec
basicmath	136871	8.16	18.10
bf	48693	3.67	50.06
crc	109625	3.50	4.24
dijkstra	208337	5.78	6.73
patricia	180455	5.60	7.57
qsort	165011	5.08	6.60
rawcaudio	106716	3.23	4.69
rawdaudio	106716	3.30	4.10
rijndael	149435	4.82	218.23
search	122043	3.18	5.62
sha	104539	3.25	8.42
susan	187091	5.46	50.19
toast	261206	7.22	45.78
untoast	261206	7.50	10.18

Table 6. CASM Compiler Statistics (compile time)

of optimizations performed. We report on the number of constant propagations (cp), lookup eliminations and update eliminations. As expected we see a large number of optimizations performed in the rijndael program. Although patricia is doing well in numbers the effects do not materialize due to the disadvantageous distribution of block contribution to the total runtime.

In table 6 the size of the test programs in LOC and the compilation times with and without optimizations are listed.

In table 7 we summarize the output produced by the CASM compiler. Generally we are generating a single C file for each rule but are merging smaller rules to reduce the number of files. For rijndael we see the by far largest increase in code size with moderate 10%. The increase in the size of the binary is approximately 20%.

To assure that the observed behavior is not solely due to optimizations of the C compiler we report on the effects of compiling optimized CASM programs with and without compiler optimizations. Figure 8 shows the relative impact of CASM optimizations with and without optimizations by the C compiler. The relative performance is clearly decreased but our optimizations still account for a factor of 2 (rijndael) to at least 1% for patricia. (On a side note: by using well-known compiled simulation techniques (e.g. [8]) the size of the basic blocks can be enlarged from which our compiler would profit immediately.) In figure 9 the overall speedup factors for the applications are shown along with absolute performance data. The speedup is relative from the non-optimized version to the fully optimized one. We are able to achieve factors 6

	C files	total LOC C w/o opt	total LOC C full opt	binary MiB w/o opt	binary MiB full opt
basicmath	4104	531769	+1357	29	35
bf	1233	179890	+10369	9.1	11
crc	3508	419546	+679	24	29
dijkstra	5462	691294	+866	38	46
patricia	5871	694523	+622	39	48
qsort	5400	641228	+1328	36	44
rawcaudio	3300	402138	+901	23	27
rawdaudio	3300	402138	+946	23	27
rijndael	3438	524481	+49198	26	32
search	3246	419649	+4660	23	28
sha	3298	408628	+5506	23	28
susan	5344	727943	+10029	28	46
toast	7819	972261	+10903	53	64
untoast	7819	972261	+3401	53	64

Table 7. Generated Output Statistics

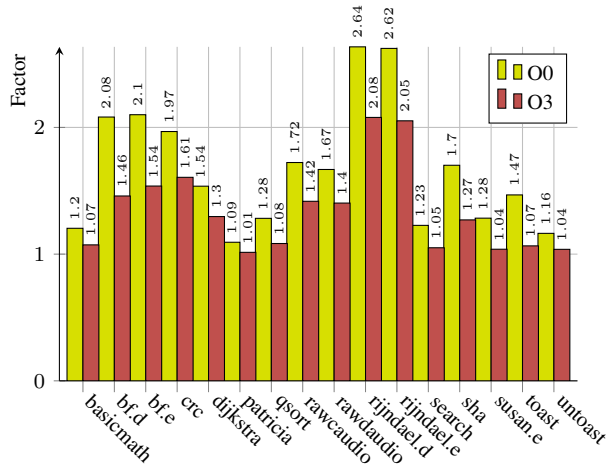

Figure 8. Impact of CASM Optimizations

and above here. For rijndael (factor 5.44) more than 50% of this speedup is due to CASM optimizations (the rest is due to the C compiler).

The MHz value relates the total number of simulated MIPS instructions to the absolute runtime of the programs We are able to achieve simulation speeds above 3 MHz which is an impressing result. The numbers also indicate that the performance without optimizations would be approximately 500 kHz. Search and susan show very low performance here. This is due to the very short execution time of these two programs (5 and 4 seconds). The startup time of the programs is approximately 1 second (the initial memory state of the MIPS programs (data section) is initialized by a CASM rule producing updates) therefore a significant reduction of simulation speed is expected.

The experimental data show a huge performance increase achieved by the CASM compiler. A speedup of more than factor 6 can be achieved. We showed that the C code generated by the CASM compiler can be very efficiently optimized. Our novel optimizations lookup elimination and update reduction can increase program performance up to 264%. This shows that they are highly effective.

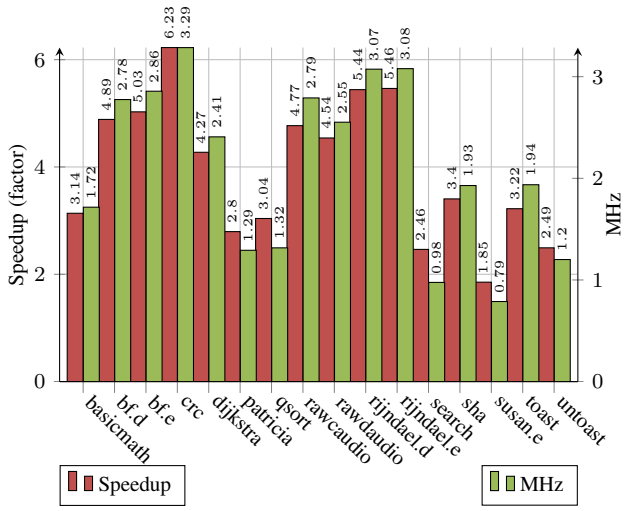

Figure 9. Improvements and Total Performance

7. Future Work

We are currently working on an inter-procedural analysis framework to reduce the amount of inlined code. Also the scope of the update elimination should be increased from sequential block scope to whole rule scope. Due to the highly effective constant propagation and constant folding we also see a large potential in implementing common subexpression elimination.

8. Conclusion

In this paper we introduced the Abstract State Machine based language CASM. We described the implementation of the runtime system and the compilation to C code. The novel PAR/SEQ Control Flow Graph representation and PAR/SEQ Use/Def Analysis based on it were presented. We then discuss how these data structures can be used to eliminate expensive runtime operations of ASM implementations. In a thorough evaluation we demonstrated that our baseline compiler alone outperforms other available implementations (including Microsoft's AsmL compiler) by 2-3 orders of magnitude. Finally we demonstrate the effectiveness of our code-generation achieving an overall speedup of up to factor 6. Our novel optimizations lookup and update elimination contribute up to 264% to the overall performance gain.

References

[1] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 163–174, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. .

[2] A. V. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2007. ISBN 978-0321547989.

[3] M. Anlauff. XASM- An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 69–90. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67959-2. .

[4] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 41–52. IEEE, 2009.

[5] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In *Formal Approaches to Software Testing, FATES 2003, volume 2931 of LNCS*, pages 264–280. Springer, 2003.

[6] J. Bergin and S. Greenfield. Teaching parameter passing by example using thunks in C and C++. *SIGCSE Bull.*, 25(1):10–14, Mar. 1993. ISSN 0097-8418. .

[7] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In *Computer Science Logic (Proceedings of CSL 2000), volume 1862 of LNCS*, pages 41–60. Springer-Verlag, 2000.

[8] F. Brandner, N. Horspool, and A. Krall. DSP Instruction Set Simulation. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*, pages 945–974. Springer New York, 2013. ISBN 978-1-4614-6858-5. .

[9] G. D. Castillo. The ASM workbench - A tool environment for computer-aided analysis and validation of abstract state machine models. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 578–581, London, UK, UK, 2001. Springer-Verlag.

[10] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77:1–33, 2007.

[11] N. G. Fruja and R. F. Stärk. The hidden computation steps of Turbo Abstract State Machines. In *Abstract State Machines — Advances in Theory and Applications, 10th International Workshop, ASM 2003*, pages 244–262. Springer-Verlag, 2003.

[12] Y. Gurevich. *Evolving algebras 1993: Lipari guide*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[13] Y. Gurevich and N. Tillmann. Partial updates. *Theoretical Computer Science*, 336(2):311–342, 2005.

[14] Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, Oct. 2005. ISSN 0304-3975. .

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. .

[16] J. K. Huggins and W. Shen. The static and dynamic semantics of C, 2000.

[17] R. Lezuo. *Scalable Translation Validation*. PhD thesis, Vienna University of Technology, 2014.

[18] R. Lezuo and A. Krall. Using the CASM language for simulator synthesis and model verification. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, pages 6:1–6:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1539-5. .

[19] R. Lezuo, G. Barany, and A. Krall. CASM: Implementing an Abstract State Machine based Programming Language. In S. Wagner and H. Lichter, editors, *Software Engineering 2013 Workshopband, 26. Februar - 1. März 2013 in Aachen*, volume 215 of *GI Edition - Lecture Notes in Informatics*, pages 75–90, February 2013. ISBN 978-3-88579-609-1. (6. Arbeitstagung Programmiersprachen (ATPS'13)).

[20] T. Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5):449–462, 2002.

[21] C. Praun, F. Schneider, and T. Gross. Load elimination in the presence of side effects, concurrency and precise exceptions. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 390–404. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21199-0. .

[22] J. Schmid. Introduction to AsmGofer, 2001. URL `http://www.tydo.de/AsmGofer`.

[23] J. Schmid. Compiling abstract state machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001.

[24] J. Teich, P. W. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using ASMs. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, ASM '00, pages 266–286, London, UK, 2000. Springer-Verlag. ISBN 3-540-67959-6.