Symbiotic Expressions

Robert Bernecky[1], Stephan Herhut[2], and Sven-Bodo Scholz[2]

[1] Snake Island Research Inc
Toronto, Canada
bernecky@snakeisland.com
[2] University of Hertfordshire, U.K.
{s.a.herhut,s.scholz}@herts.ac.uk

Abstract. We introduce *symbiotic expressions*, a method for algebraic simplification within a compiler, in lieu of an SMT solver, such as Yices or the Omega Calculator. Symbiotic expressions are compiler-generated expressions, temporarily injected into a program's abstract syntax tree (AST). The compiler's normal optimizations interpret and simplify those expressions, making their results available for the compiler to use as a basis for decisions about further optimization of the source program. The expressions are symbiotic, in the sense that both parties benefit: an optimization benefits, by using the compiler itself to simplify expressions that have been attached, lamprey-like, to the AST by the optimization; the program being compiled benefits, from improved run-time in both serial and parallel environments.

We show the utility of symbiotic expressions by using them to extend the SAC compiler's With-Loop-Folding optimization, currently limited to Arrays of Known Shape (AKS), to Arrays of Known Dimensionality (AKD). We show that, in conjunction with array-based constant-folding, injection and propagation of array extrema, and compiler-based expression simplification, symbiotic expressions are an effective tool for implementing advanced array optimizations. Symbiotic expressions are also simpler and more likely to be correct than hard-coded analysis, and are flexible and relatively easy to use. Finally, symbiotic expressions are synergistic: they take immediate advantage of new or improved optimizations in the compiler. Symbiotic expressions are a useful addition to a compiler writer's toolkit, giving the compiler a restricted subset of the analysis power of an SMT solver.

1 Introduction

Compilers use a variety of algebraic expression analysis techniques to make code optimization decisions. For example, loop fusion in SISAL [1,2,3] requires that two loops have the same bounds; an array-bounds-check removal in Fortran must determine if an index vector lies entirely within the array from which it selects; the array language SAC's With-Loop-Folding (WLF) optimization [4,5] must compute the intersection of two index-vector sets. Although these operations are often straightforward for humans to solve by inspection, a compiler may require

M.T. Morazán and S.-B. Scholz (Eds.): IFL 2009, LNCS 6041, pp. 107–124, 2010.

some effort to achieve the same end. Common compiler-based approaches for performing this task include *ad hoc* analysis code, tailor-made for each specific optimization, such as [6], or use of an existing SMT solver, such as Yices [7] or the Omega Library [8,9]. In order to motivate a different approach to the problem, consider an $\mathtt{N}^{th}$-difference array computation, used for signal processing or time-series analysis or, with `N = 1`, to compute delta-modulation values from a digitized audio signal, `X`, written as the SaC expression:

```
X - (genarray( [N], 0) ++ drop(-N, X))
```

That is, create a zero-vector of length `N`; catenate to it the original vector `X` after dropping its last `N` elements; then subtract that from `X`. Instead of performing these operations piecemeal, each creating array-valued intermediate results, we would like the compiler's WLF optimization to fuse those operations into a single loop. To do this, the optimizer has to prove that the shape of the catenate result matches the shape of `X`. In certain limited contexts, idiom recognition can detect common code patterns of this sort. However, in the more general case, the compiler must solve algebraic expressions involving array shapes and values, using standard rules of arithmetic on arrays, distributive law, associative law, de Morgan's laws, Boolean algebra, array versions of constant and value propagation, constant folding, common sub-expression elimination, etc. A common compiler design approach here is to restrict the domain of an optimization to fixed-shape arrays, and to simple index expressions. The required algebraic analysis is then hard-coded into the optimization itself.

Going beyond this point with hard coding is tedious, it does not generalize, and is not good software engineering. Although SMT solvers can often simplify these expressions, there are a few problems with that approach. First, there is an impedance mismatch between the intermediate language (IL) of the compiler and the API of the solver: the relevant IL data must be converted to a form acceptable to the solver, and when the solver finishes, its results must be converted back to IL form. Second, and somewhat harder, is the question of exactly what data should be provided to the solver. In the above example, we would need information about the argument and result shapes of a number of functions, potential values of parameters, etc. The task of deciding exactly what metadata–array shapes, index vector bounds, etc.–have to be passed to the solver can be nearly as difficult as solving the problem itself. Finally, it is difficult to use partial results generated by an SMT solver, since the answers tend to be of the yes-no variety: results from partial constraint resolution are not easy to exploit. Given this, we decided to tackle the problem in a different way.

Our solution entailed some modest extensions to the SaC constant-folding (CF) optimizations, to include some of the constraint resolution capabilities used in SMT solvers. Traditional constant folding comprises simple term-rewriting rules, such as replacing the vector expression `([2,3,4] + 4)` by `[6,7,8]`. The SaC compiler extends traditional constant folding to arrays in several ways, shown in Figure 1. Symbolic constant simplification (SCS) simplifies array-valued expressions on Boolean, arithmetic, and other functions. Constant Folding also performs removal of run-time guards and compiler-internal primitives, in the

form of SAA-constant folding (SAACF). This is driven by SAA-derived array rank and shape information [10]. Structural-constant constant folding (SCCF) replaces an array expression by its elements when those parts are arrays, and eliminates indexed references to array-valued intermediate results, thereby implementing array contraction [2].

Type	Expression	Result
SCS	`Array + 0`	`Array`
SCS	`Array - Array`	`genarray(shape(Array), 0)`
SCS	`Vec * VecOfZeros`	`VecOfZeros`
SCS	`BoolArray | FALSE`	`BoolArray`
SCS	`BoolArray | TRUE`	`genarray(shape(BoolArray), TRUE)`
SCS	`max( Array, Array)`	`Array`
SCS	`Array <= Array`	`genarray(shape(Array), TRUE)`
SCCF	`sel( [2], [a, b, c, d]`	`c`
SCCF	`[a, b] ++ [c, d]`	`[a, b, c, d]`
SCCF	`X = modarray( M, iv, V)`	`z = V`
	`z = sel( iv, X)`	...
SAACF	`take( shape(Vec), Vec)`	`Vec`

Fig. 1. Array-based constant folding examples

The remainder of this paper is structured as follows: Section 2 introduces *Symbiotic Expressions*, an alternate solution, within our limited context, to an SMT solver, introduces *array extrema*, and presents a running example of our solver injecting, propagating, simplifying, and exploiting symbiotic extrema expressions; Section 3 offers an example of using symbiotic extrema expressions to implement a new optimization, Algebraic With-Loop Folding (AWLF), including the performance characteristics of the new optimization; Section 4 presents the impact of symbiotic expressions within the compiler itself; Section 5 discusses related work; we end with our conclusions and future work in Section 6.

2 Symbiotic Expression Design and Implementation

The SaC compiler's optimizers are heavy-duty array tools, including associative law (AL), arithmetic simplification (AS), array constant folding (CF), common-subexpression elimination (CSE), constant propagation (CP), distributive law (DL), value propagation (VP), and many others.

The availability of such capabilities led us to conjecture that the SaC compiler, with some enhancements, might be powerful enough to perform the algebraic simplifications required by new optimizations. Our idea was to have the compiler inject appropriate algebraic expressions and/or SaC source code into the Abstract Syntax Tree (AST) of the program being compiled, run them through the optimizer cycle, and then see if the expressions had been simplified enough to guide further optimization.

Our proposed approach offered several potential advantages over an SMT solver. First, the entire metadata question became a non-issue, because all information needed by the compiler-based solver was directly available as AST nodes. Second, if our approach worked, it would constitute a generic technique that could be used for other algebraic expression simplification work within the compiler, giving it on-going, extensible value. Third, any new optimizations within the compiler would themselves become available for use by the rest of the compiler, perhaps improving the performance of symbiotic-expression-based optimizations that have no apparent connection to the new optimization.

We now describe the design and implementation of *Symbiotic Expressions*, and how we have used such expressions to implement Algebraic With-Loop Folding (AWLF), an extended array optimization. We use a trivial SaC program as a running example, to illuminate salient steps of the compiler's actions.

2.1 Running Example

Our running example is a trivial SaC program that computes a vector of the first `N` integers, then reverses that vector and displays the result. This could be written as `print(reverse(iota(N)))`, using SaC standard libraries, but inasmuch as we want to highlight the code simplification process, we provide the required functions as source code. We start with the program shown in Figure 2. Our goal in the example is to have the compiler fold with-loops into a single, data-parallel with-loop, and to perform other beneficial code improvements.

The fundamental structure of interest here is the SaC *with-loop*, a data-parallel array comprehension construct comprising two basic components: the first is a *shape* descriptor, such as `genarray(s,c)`, that specifies creation of an array with frame shape `s`, and cell shape `c`, to produce a result of shape `shp++shape(c)`, with sub-array `c` as the contents of its cells. For example, `with : genarray([2,3], 42)` creates an array of shape `[2,3]`, populated with `42`. The second component is zero or more *generators* that specify the contents of sub-arrays within that array. For example, adding the generator `([0,0] <= i <= [0,1]) : 666` creates a two-row matrix, in which each row is `[666, 42]`.

2.2 Symbiotic Expressions

Symbiotic Expressions are compiler-generated expressions, written in SaC itself or the IL, that are attached, lamprey-like, as AST nodes near relevant primitives. The compiler simplifies these expressions in exactly the same way that it simplifies the rest of the AST, because the injected expressions are indistinguishable from other AST nodes. Unlike the lamprey, however, symbiotic expressions, once simplified, are exploited by the compiler's optimizers to improve code even further. This benefits the "host" AST and the compiler, hence the term *symbiotic*.

We represent symbiotic expressions as primitive functions with an arbitrary number of arguments, the first of which becomes the result of the function, allowing us to insert symbiotic expressions into the AST with the assurance that data flow will preserve them over optimizations. Remaining arguments have

```
use Array:{-,<,shape,sel};
int main()
{ N = StdIO::readInt();
  ints = iota(N);
  z = reverse(ints);
  StdIO::print(z);
  return(0);
}

inline int[.] reverse( int[.] v)
{ lim = shape(v) - 1;
  z = with { ( [0] <= [j] < shape(v) ) : v[lim - j];
        } : genarray( shape(v), 0 );
  return( z);
}

inline int[.] iota( int y)
{ z = with { ( [0] <= [k] < [y]) : k;
        } : genarray( [y], 0 );
 return(z);
}
```

Fig. 2. Running example source code

semantic meaning only in the context of their application: their expressions are simplified by the optimizers, but the compiler is otherwise unaware of them. We might re-implement symbiotic expressions using just one primitive function, attaching, as the second argument, a tag that would identify the particular type of symbiotic expression being computed.

Symbiotic expressions are removed from the AST by a post-optimizer traversal and by dead code removal (DCR), so they have no detrimental impact on code generation. In this regard, they are merely code annotations, similar in spirit to the PHI functions of Static Single Assignment form [11]. The code that was optimized, thanks to the presence of symbiotic expressions, of course, remains optimized. Now, we discuss the injection, propagation, simplification, and exploitation of symbiotic expressions in slightly more detail.

2.3 Extrema

We introduced *extrema* into the SaC compiler to allow us to analyze array shapes and index vector sets. Extrema are expressions, attached to index vector descriptors, that give estimates of the minimum and maximum values of an array, to be used in a manner similar to integer range analysis [6]. For instance, the index scalar `j` in the `reverse` function has a minimum value of the vector `[0]` and a maximum value of `shape(v)`. For compatibility with the compiler's internal canonical representation for with-loop bounds, the maximum extremum is greater by one than the actual maximum index value, which is `shape(v)-1`.

2.4 Extrema Injection

The SaC compiler uses a functional AST, in the sense that all of AST nodes must be connected by data flow to the main computation, or they are deemed dead, to be deleted from the AST by DCR. Hence, symbiotic expressions that are inserted into the AST must be hooked into it functionally, so that they will be preserved over optimizations; we discuss this requirement in [12].

Extrema are associated with-loop induction variables as arguments to an internal primitive function, `_attachextrema_`, that serves to preserve the extrema via data flow, and to associate extrema values with each variable. The `reverse` function, after introduction of extrema, is shown in Figure 3. By this time, any compiler optimizations on `j` can exploit its extrema. However, the index operation, `v[idx]`, which could exploit extrema, has no extrema on `idx` as yet; that information becomes available later, through extrema propagation.

```
inline int[.] reverse(int[.] v)
{ lim = shape(v) - 1;
  lb = [0];
  ub = shape(v);
  z = with { (lb <= [j] < ub) {
           j = _attachextrema_(j, lb, ub);
           idx = lim - j;
           el = v[idx];
           } : el;
         } : genarray(shape(v), 0 );
  return(z);
}
```

Fig. 3. IL after expression injection

2.5 Extrema Propagation

In order to be useful, extrema must be propagated from their origins to the referents of the variables with which they are associated. In our example, `idx` is offset from the with-loop's index vector, `j`. When one of a primitive function's arguments has extrema, the compiler's *extrema propagation* phase will inject symbiotic expressions to compute extrema for the primitive, attaching them to the AST as was done for with-loop index vectors. From a Hoare logic perspective, extrema propagation thus derives post-conditions for primitive operations given an existing pre-condition. In our example, the vector index offset computation `(lim-j)` meets this criterion, so two subtraction expressions are created, to compute the new extrema for `idx`, using `lim` and the extrema of `j` as arguments. The resulting expressions are injected into the AST as shown in Figure 4, then subjected to the compiler's normal optimizations, where they will be simplified, hopefully enough to be exploited by the compiler.

```
inline int[.] reverse(int[.] v)
{ lim = shape(v) - 1;
  lb = [0];
  ub = shape(v);
  z = with { (lb <= [j] < ub) {
          j = _attachextrema_(j, lb, ub);
          idx = lim - j;
          maxidx = lim - lb + 1;
          minidx = lim - (ub - 1);
          idx = _attachextrema_(idx, minidx, maxidx);
          el = v[idx];
          } : el;
        } : genarray(shape(v), 0 );
  return(z);
}
```

Fig. 4. IL after expression propagation

2.6 Symbiotic Expression Simplification and Exploitation

Symbiotic expression simplification is performed entirely by compiler optimization cycles. We will summarize the process as it operates on our running example. With-loop invariant removal (WLIR) moves `maxidx` and `minidx` out of the with-loop; array-based Constant Folding (CF) removes the idempotent subtraction of vector zero in `maxidx`; Constant Folding (CF) reduces the addition and subtraction of one to zero, ultimately leading to `shape(v)` as value of `maxidx`. Constant and Value Propagation (CVP) replaces `maxidx` by `ub`, making `maxidx` dead code, removed by Dead Code Removal (DCR), giving the IL shown in Figure 5.

```
inline int[.] reverse(int[.] v)
{ lim =  shape(v) - 1;
  lb = [0];
  ub = shape(v);
  minidx = lim - (ub - 1);
  /* minidx: ((shape(v)-1) - (shape(v)-1)) */
  z = with { (lb <= [j] < ub) {
          j = _attachextrema_(j, lb, ub);
          idx = lim - j;
          idx = _attachextrema_(idx, minidx, ub);
          el = v[idx];
          } : el;
        } : genarray(shape(v), 0 );
  return(z);
}
```

Fig. 5. IL after optimization cycle 1

At this point, Algebraic With-Loop Folding Inference (AWLFI) injects code, slightly more complex than that shown in Figure 6, to compute the intersection of the with-loop bounds in `iota()` that generated `v` with `idx`'s index set extrema, using the `_attachintersect_` primitive to attach the calculation to the AST, in a manner similar to guards [12] and integer range analysis [6]. As with `_attachextrema_`, this primitive uses data flow to associate its first argument with its result; the remaining arguments act as annotations to be simplified by the optimizers, and provide a mechanism to let the AWLF traversal find the requisite intersection information for the subsequent index operation. Figure 7 shows how in SaC a symbiotic expression can be written.

Concurrently, AL, AS, and DL rearranged the terms comprising `minidx`, from:

```
((shape(v)-1) - (shape(v)-1))
```

to:

```
((shape(v)-shape(v)) + (1-1))
```

Next, CF replaces both terms by `[0]`, and later eliminates the addition. Since `lb` is also `[0]`, CF replaces `minint` by `[0]`. The term `maxint` is similarly simplified,

```
inline int[.] reverse(int[.] v)
{ lim =  shape(v) - 1;
  lb = [0];
  ub = shape(v);
  minidx = lim - (ub - 1);
  z = with { (lb <= [j] < ub) {
          j = _attachextrema_(j, lb, ub);
          idx = lim - j;
          idx = _attachextrema_(idx, minidx, ub);
          minint = sacprelude:partitionIntersectMin(lb, minidx);
          maxint = sacprelude:partitionIntersectMax(ub, ub);
          idx = _attachintersect_(idx, minint, maxint);
          el = v[idx];
          } : el;
        } : genarray shape(v), 0 );
  return( z);
}
```

Fig. 6. IL after array intersection computation insertion

```
inline int[.] partitionIntersectMax( int[.] idxmin, int[.] bound1)
{
  dif  = _sub_VxV_( idxmin, bound1);
  p = _ge_VxS_( dif, 0);
  z = _mesh_VxVxV_( p, idxmin, bound1);
  return(z);
}
```

Fig. 7. SAC-defined symbiotic expression function

first to `max(shape(v),shape(v))`, and then to `shape(v)`, giving the IL shown in Figure 8. At this point, the intersect terms `minint` and `maxint` match the extrema of `idx`, which satisfies the needs of AWLF.

```
inline int[.] reverse( int[.] v)
{ lim =  shape(v) - 1;
  lb = [0];
  ub = shape(v);
  z = with { ( lb <= [j] < ub) {
           j = _attachextrema_( j, lb, ub);
           idx = lim - j;
           idx = _attachextrema_( idx, lb, ub);
           idx = _attachintersect_( idx, lb, ub);
           el = v[idx];
           } : el;
         } : genarray( shape(v), 0 );
  return( z);
}
```

Fig. 8. IL after constant folding

The intersect calculation is the information AWLF needs to replace `v[idx]` with the cell computation body from `iota()`. In this trivial case, the code body is merely the with-loop's induction variable, so we end up, after extrema and dead code deletion, with the single, well-optimized with-loop shown in Figure 9.

```
inline int[.] reverse( int[.] v)
{ lim =  shape(v) - 1;
  lb = [0];
  ub = shape(v);
  z = with { ( lb <= [j] < ub) {
           el = lim - j;
           } : el;
         } : genarray( shape(v), 0 );
  return( z);
}
```

Fig. 9. IL at completion

3 Algebraic With-Loop Folding

Now that we understand what symbiotic expressions are, and how they work, we shall see how well they work in practice. With-loop folding (WLF) is the fundamental loop optimization in the SAC compiler. It enables APL-like array operations to be combined into a single data-parallel loop, eliminating array-valued intermediate results and increasing the parallelism available within the

program being compiled. WLF operates only on Arrays of Known Shape (AKS)– the arrays of Fortran 77–but many array language applications deal with Arrays of Known Dimension (AKD), but unknown shape. Although the AKS and AKD versions of code may be nearly identical, AKD performance can be much worse than AKS performance, because the compiler may not be able to deduce exact shapes of all arrays. This led us to write a new optimization, *algebraic with-loop folding* (AWLF), able to fold AKS and AKD arrays by using symbiotic-expressions to perform the partial symbolic evaluation of algebraic with-loop index sets and their set intersections required by the optimization. We then conducted a series of experiments to quantify the relative performance of AWLF against WLF.

3.1 Experimental Setup

We evaluated the utility of symbiotic expressions by measuring their effectiveness in performing AWLF *vs.* WLF, and also by measuring the performance of each optimization on AKS *vs.* AKD arrays. Our platform was a 4GB AMD Opteron 165, running at 1.8GHz, running Ubuntu 8.10, gcc 4.3.2 and sac2c product Build #16338 (`www.sac-home.org`). We used CPU time measurements, taken with PAPIEX [13,14], as our metric for this paper.

3.2 Experimental Findings and Performance

First, we compared the performance of AWLF *vs.* WLF on AKS array problems, which gives the edge to WLF, because WLF exploits knowledge of fixed-shape arrays. Figure 10 shows CPU-time ratios for AWLF *vs.* WLF on AKS-based benchmarks, taken from the APEX test suite [15]. Our hope was that AWLF would approach the same level of performance as WLF. Measured performance levels turned out to be mixed, but quite respectable, for about two-thirds of the tests. The benchmarks in which AWLF performs poorly, such as `logd2AKS` and `primesAKS`, include those that require *with-loop partition slicing*, a WLF optimization that we have not yet completely implemented in AWLF. In general, we deem these results acceptable, as we understand the areas where AWLF is deficient and we are taking corrective measures.

We then turned to the performance of AWLF on AKD arrays. Since these arrays frequently appear in applications such as data base queries, stock exchange trading histories, stock market portfolio analysis, etc., we consider AKD performance to be at least as important as AKS performance. The benchmark programs for AKS and AKD are, essentially, identical. They differ only in that the AKD versions hide the problem size from the compiler, preventing it from inferring array shapes, as it does for the AKS versions. As with the AKS experiments, we measured CPU times, shown in Figure 11, as the speedup of AWLF over WLF[1]. Here, the benefits of AWLF emerge, showing most benchmarks slightly faster under AWLF than under WLF; several are considerably faster.

[1] We have intentionally restricted the y-axis scales on these charts, so that detail near the bottom remains clear.

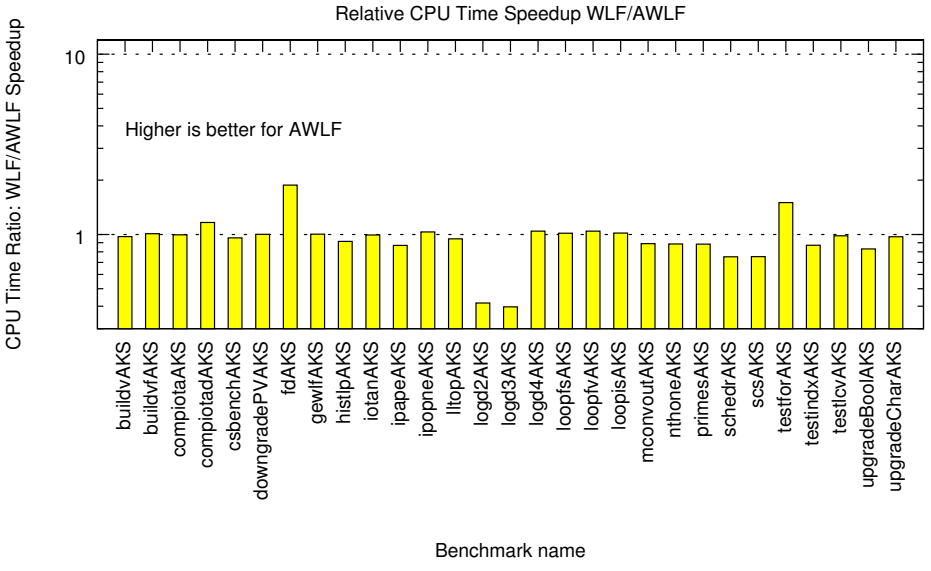

Fig. 10. APEX AKS CPU time performance WLF *vs.* AWLF

The remaining problem areas in AWLF are attributable to two factors. First, the uncompleted implementation of with-loop partition slicing affects some benchmarks. Because AWLF makes use of Symbolic Array Attribute (SAA) [10] information, it is implemented in a different optimimzation cycle from WLF. Both optimization cycles can be enabled independently; eventually, we will de-support the WLF cycle. This problem does not, therefore, detract from the inherent benefits of symbiotic expressions and AWLF.

We wanted to measure the performance of AWLF on AKS data *vs.* AKD data. Ideally, both problems would execute with the same speed, except for degradation due to the need to pass array shapes into, and out of, functions. Hence, if all results showed 100% for relative AKS and AKD performance, we could claim success in masking the differences between AKS and AKD problems in AWLF. The reality, is slightly different. Figure 12 presents the relative CPU time measurements for AWLF operating on AKS and AKD problems; WLF measurements are provided for comparison purposes. AWLF nearly always matches or exceeds WLF performance, in the sense of making performance of AKD-based problems approach AKS-based ones. If AWLF matches WLF performance on a benchmark, it means either that the benchmark is essentially entirely AKS-based, or that AWLF is missing some potential folding opportunities.

It is clear that AWLF often performs better, offering AKD performance that rivals AKS. Both optimizations, for reasons we do not yet understand, occasionally do better on AKD problems than on AKS problems. These are the places where the bar dips below 100%. The cause is likely some code unrelated to WLF and AWLF, because these optimizations have no code in common.

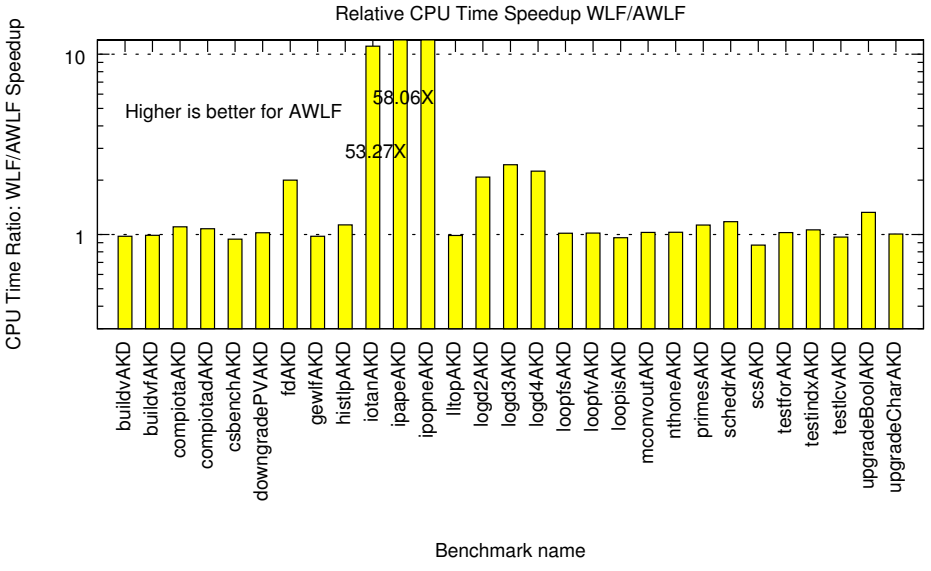

Fig. 11. APEX AKD CPU time performance WLF *vs.* AWLF

Cache performance generally improves with AWLF over WLF, as can be seen in Figure 13 and Figure 14. L1 cache performance under AWLF is, with two exceptions, either identical to, or superior to, that under WLF, sometimes by one or more orders of magnitude. L2 cache performance shows similar results.

4 Compiler Impact

Given the poor theoretical worst-case performance of SMT solvers such as Yices and Omega, it is reasonable to ask how well the SaC compiler performs when it solves symbiotic expressions. We do not know, as we have not yet made any controlled experiments to measure it, but we have observed that the SAACYC optimization cycle runs more trips when it is solving those expressions. The cost of our injected symbiotic expressions is, at least, several extra operations per primitive involved in the chain between index vector creation and index vector use, those operations being the ones that compute the new index vector extrema and attach them to the AST. Multiple injections may be inserted and solved concurrently, so the number of additional optimization cycles probably is not as bad as it could be. Also, since the injected expressions have a tendency to be quite simple, they are evaluated relatively quickly. Even so, their resolution may require several iterations of the optimization cycle.

If the increased iteration count becomes a problem in practice, we might want to consider introducing *micro-optimization cycles*, which we conceive of as allowing a subset of optimizations to be explicitly invoked on a specific user-defined function. The observation here is that optimizations such as CF, AS,

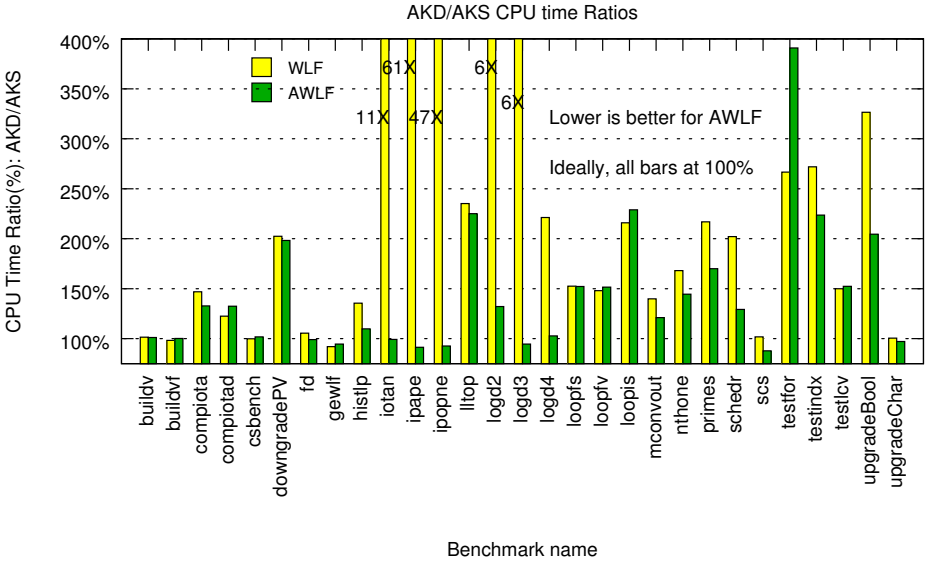

Fig. 12. APEX CPU time performance AKD *vs.* AKS

DL, AL, VP, and CSE are useful for simplifying our symbiotic expressions used in AWLF, other optimizations, such as those that manipulate with-loops, are not, and could be omitted in a micro-optimization cycle.

In terms of compiler complexity, the work required to support symbiotic expressions was minimal, partly due to the excellent structure of the SaC compiler as a research tool. Adding new primitives to the SaC compiler is merely a matter of adding an entry to one table, and defining the relationship between the function result and its arguments, basically a cut-and-paste operation. Simple abstract syntax tree traversals were added to introduce, propagate, exploit, and delete the two new compiler primitives and extrema. This was facilitated by Stephan Herhut's creation of XML code that allows new compiler traversals to be added with just a few lines of XML. Exploitation of extrema required more effort, since it has to drive the entire AWLF optimization. Introduction and propagation of extrema is straightforward; deletion of extrema is trivial.

In [16], which discusses some data flow problems similar to those in [12], the authors state "A refinement-style representation also obscures optimization opportunities by introducing multiple names for the same value." The SaC compiler uses pattern-matching to chase back across multiple assignments, etc., so this problem was largely solved for us already. Trivial changes were required to several optimizations to make them skip over the extrema and intersect primitives, usually no more than a name change in a pattern-matching function, from one that traces back over assign chains, to one that traces back over both assign chains and attach primitives.

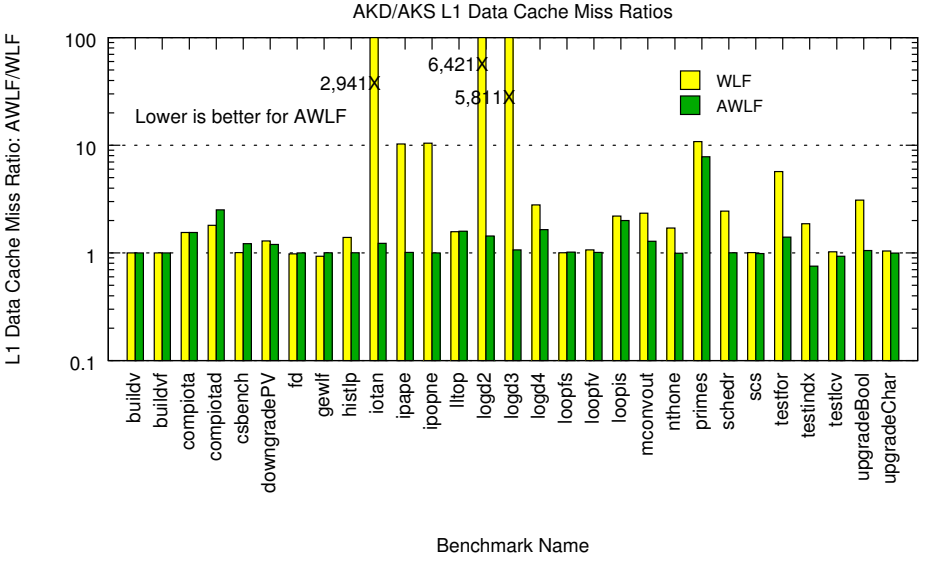

Fig. 13. APEX L1 cache performance AKD *vs.* AKS

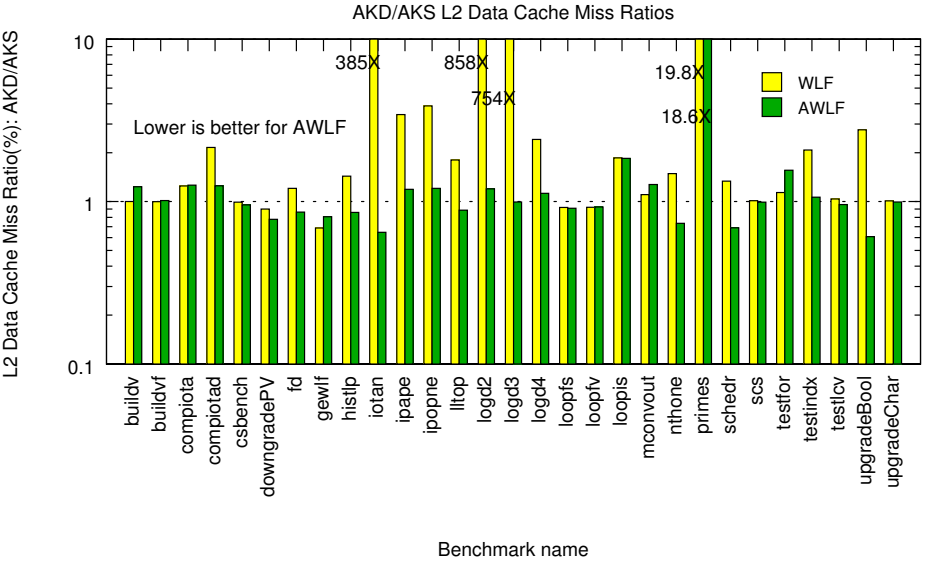

Fig. 14. APEX L2 cache performance AKD *vs.* AKS

Most compiler traversals merely ignore primitives that are not relevant to them, so the new primitives did not present any significant problems. The semantics of the new primitives are very simple, from the standpoint of most of the compiler; they can be thought of as idempotent functions whose result is their first argument; remaining arguments are, essentially, annotations, even though they are the symbiotic expressions that are simplified by the compiler.

Two WLF-related problems surfaced during the course of our experiments. First, we noticed that AKD matrix products on character data, `ipapeAKD` and `ipopneAKD`, performed about fifty times slower under WLF than under AWLF. We considered discarding these benchmarks from our test suite, because they reflect a performance problem that seems to go beyond WLF: other matrix product codes, such as `ipddAKD`, do not have this problem. However, the very presence of such performance problems makes it clear that the ease of using built-in optimizations to simplify algebraic expressions, rather than hand-crafting hard-coded analysis, has payoffs beyond mere convenience for the compiler writer. Hard-coded solutions may harbor code faults that only rarely raise their ugly heads. Such faults are usually only found by careful examination of generated IL code using a debugger to trace the cause of the fault.

Second, the `dtb` class of APEX benchmarks do not appear here because the WLF tests crashed the SaC compiler, due to an erroneously detected array bounds violation in WLF array offset computation. The AWLF code does not exhibit this failure. Again, this demonstrates the virtue of using existing optimizations to do the heavy lifting for new ones.

5 Related Work

Presently, we have not investigated the use of symbiotic expressions in other compiler contexts nor the use of array-based constant folding in other compiler projects. However, APL interpreters have always included special-case code to detect algebraic identities, such as those shown in the introduction. Many of the array-based constant-folding optimizations in SaC have their roots in APL.

In part, the extrema work was inspired by, and is a generalization of, earlier work on Symbolic Array Attributes, Whereas SAA annotates AST array descriptors with array dimension and shape information, extrema annotate them with array value information; both methods then exploit the optimization process itself to enable further optimizations; SAA information can, therefore, also be considered symbiotic expressions, in the sense described here.

The main aim of the introduction of SAA expressions was to enable optimizations that require shape equalities as well as shape information across function boundaries. After optimization, again in the spirit of symbiotic expressions, unused shape information is eliminated from the syntax tree.

The symbiotic expressions introduced in this paper are a generalization of the SAA approach, albeit applied in a rather different context. While making shape information explicit can be seen as a change in representation that facilitates further optimizations, extrema information adds new context information

about the range of values possible at runtime. This information is attached and propagated for selected values only, rather than for all values. Furthermore, the constraints of interest here are not just equalities, but also inequalities of all sorts. In our experience, simplification of shape expressions requires, for the purposes of AWLF, exploitation of both SAA and extrema information.

The use of symbiotic expressions to narrow the range of index values in array comprehensions has parallels with refinement types [17,18], also referred to as predicate sub-typing [19]. Similar to refinement types, we annotate type information with predicates–the minimal and maximal value of an index vector at runtime. However, other than in refinement type systems, these refinements are annotated automatically by the compiler and exploited only for the optimizations. Also, we do not use symbiotic expressions to prove user-defined properties of programs. The main difference of our approach is that our annotations are regular SaC expressions. They are neither encoded in the type system itself nor are they a specific extension for refinement types. This allows us to reuse existing optimizations, opposed to requiring a specialized SMT solver.

For the same reasons, symbiotic expressions are not dependent types. However, symbiotic expression might be useful in the context of dependent types to prove user-annotated constraints and infer type information.

6 Conclusions and Future Work

Symbiotic expressions are source language or IL expressions. They are inserted into the AST by the compiler, simplified by its optimization phases, then the results are used by the compiler in further optimization, or for other desired purposes, after which they are deleted from the AST, so they do not appear in run-time code. Symbiotic expressions are a completely general approach for letting a compiler use itself to solve its own problems, whatever they may be.

Our use of symbiotic expressions to implement AWLF within the SaC compiler has paid off well. The optimization worked correctly, essentially, from day zero; the injection and propagation of symbiotic expressions was simple and straightforward. It is a general approach that is applicable to many common compiler problems.

Our success with symbiotic expressions suggests that, even if a compiler may not have all the power of an SMT solver, our approach can be of use in other contexts where SMT-like solvers or hand-coded expression simplifiers are used today. Specifically, symbiotic expressions should be usable in almost any compiler environment, including imperative, object-oriented, or functional settings. The main contribution of our approach is that it makes all of the AST metadata available in a common framework, where it is directly shared by optimizers and other compiler components. We suggest that extending the optimization capabilities of compilers to allow them to act as tightly integrated SMT solvers would offer even greater benefits than we have obtained already.

Several benefits accrued from our use of symbiotic expressions to implement AWLF. First, AWLF often brings AKD problem performance very close to AKS

performance. Second, AWLF significantly reduces cache miss rates over WLF. Third, because with-loops are the fundamental unit of parallel execution in SaC AWLF increases the size of data-parallel code blocks and reduces the number of parallel synchronization barriers. Since AWLF is able to fold more with-loops than WLF, AWLF increases the available parallelism in AKD-dominated programs over WLF. Although space limitations preclude detailed discussion of parallel performance measurements or performance against Fortran dialects, we note that the `logd2` benchmark executed in 1493msec under Fortran 95, 433msec under Fortran 77, and 295msec under SaC in a serial environment.

We have begun to apply symbiotic expressions to the partial evaluation of run-time guards. This project has the potential to increase significantly the number of guards that can be statically removed from SaC programs. Also, as we noted in [12], guards can allow many optimizations to be performed in the absence of precise information about the arrays upon which they operate. These *guarded optimizations*, or *optimistic optimizations*, can create the desirable situation whereby introduction of safety features, such as array bounds checking, can materially speed programs up, rather than slow programs down.

We have not yet tried to scale symbiotic expressions beyond annotating and exploiting range information for index values. It would be interesting to investigate if our approach can be extended to provide some or all of the power of full refinement types. This would require more general annotations and a more powerful optimization-based solver. To determine whether this would bring our approach on a par with SMT-solver based solutions remains future work.

Acknowledgements

We thank the anonymous referees for their thoughtful comments and insights.

References

1. Cann, D.C.: Compilation Techniques for High Performance Applicative Computation. PhD thesis, Computer Science Department, Colorado State University (1989)
2. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Computing Surveys 26, 345–420 (1994)
3. Padua, D., Wolfe, M.: Advanced Compiler Optimizations for Supercomputers. ACM Comm. 29, 1184–1201 (1986)
4. Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13, 1005–1059 (2003)
5. Scholz, S.B.: With-loop-folding in SAC–Condensing Consecutive Array Operations. In: Clack, C., Hammond, K., Davie, T. (eds.) IFL 1997. LNCS, vol. 1467, pp. 72–92. Springer, Heidelberg (1998)
6. Rugina, R., Rinard, M.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In: PLDI 2000 Conference Proceedings, pp. 182–195. ACM, New York (2000)

7. Dutertre, B., de Moura, L.: The yices smt solver. Technical report, SRI International (2006)
8. Pugh, W.: A practical algorithm for exact array dependence analysis. CACM 35, 102–115 (1992)
9. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The OMEGA library, version 1.1.0 interface guide. Technical report, University of Maryland (1996)
10. Trojahner, K., Grelck, C., Scholz, S.B.: On Optimising Shape-Generic Array Language Programs using Symbolic Structural Information. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 1–18. Springer, Heidelberg (2007)
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method for computing static single assignment form. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pp. 23–35 (1989)
12. Herhut, S., Scholz, S.B., Bernecky, R., Grelck, C., Trojahner, K.: From contracts towards dependent types: Proofs by partial evaluation. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 254–273. Springer, Heidelberg (2008)
13. Browne, S., Deane, C., Ho, G., Mucci, P.: Papi: A portable interface to hardware performance counters. In: HPCMP Users Group Conference, U.S Department of Defense (1999)
14. Mucci, P.: Papiex - execute arbitrary application and measure hardware performance counters with papi (2009)
15. Bernecky, R.: APEX: The APL Parallel Executor. Master's thesis, University of Toronto (1997)
16. Menon, V.S., Glew, N., Murphy, B.R., McCreight, A., Shpeisman, T., Tabatabai, A.-R., Petersen, L.: A verifiable SSA program representation for aggressive compiler optimization. In: POPL 2006 Conference Proceedings, pp. 397–408. ACM, New York (2006)
17. Freeman, T., Pfenning, F.: Refinement types for ml. SIGPLAN Not. 26, 268–277 (1991)
18. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: POPL 1999, pp. 214–227. ACM Press, New York (1999)
19. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. IEEE Transactions on Software Engineering 24, 709–720 (1998)