SAC: Off-the-shelf Support for Data-parallelism on Multicores

Clemens Grelck[1]
University of Lübeck
Institute of Software Technology
and Programming Languages
Ratzeburger Allee 160
23538 Lübeck, Germany
grelck@isp.uni-luebeck.de

Sven-Bodo Scholz
University of Hertfordshire
Department of
Computer Science
College Lane
Hatfield, AL10 9AB, United Kingdom
s.scholz@herts.ac.uk

ABSTRACT

The advent of multicore processors has raised new demand for harnessing concurrency in the software mass market. We summarise our previous work on the data parallel, functional array processing language SAC. Its compiler technology is geared towards highly runtime-efficient support for shared memory multiprocessors and, thus, is readily applicable to today's off-the-shelf multicore systems. Following a brief introduction to the language itself, we identify the major compilation challenges and outline the solutions we have developed.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages, concurrent, distributed, and parallel languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures, dynamic storage management*

General Terms

Multicore programming

Keywords

SAC, Single Assignment C, generic array programming, automatic parallelisation, data parallel programming

1. INTRODUCTION

The advent of multicore processors has raised new demand for expressing and exploiting concurrency in mainstream software engineering. The systems available on the processor mass market today consist of a growing number of standard (von Neumann) cores, which are symmetrically connected to a shared memory, i.e., they effectively constitute shared memory architectures. From research in the last few decades it is well known that programming such systems efficiently can be a challenging task: Application problems need to be subdivided and scheduled to processing resources, and the granularity of individual subtasks needs to be chosen carefully; memory access must be controlled to avoid the cache coherence protocol or the memory subsystem as a whole to quickly develop into a bottleneck, etc. As a consequence, it requires considerable expertise in programming parallel systems in addition to application domain specific knowledge in order to expose concurrency to the executing machinery in a successful way.

An extra challenge arises from the fact that most commonly used approaches to parallel programming are rather low-level: Programs usually need to be fine-tuned to the particularities of the target hardware. Therefore, whenever applications are to be executed on new hardware they need to be at least partially rewritten. This contributes to the high cost of engineering parallel programs.

The only remedy to this situation lies in tools and languages that enable advanced compilers to automatically identify concurrency and to efficiently map it onto different multicores architectures. So far, this has only been achieved in application areas where specific concurrency patterns prevail. Amongst them, approaches based on data-parallel programming play a major role. Our contribution to this field of research is the data-parallel functional programming language SAC (Single Assignment C) [1, 2]. Partly inspired by earlier developments in the areas of SISAL [3], NESL [4] and APL [5], we geared the design of SAC towards a certain set of application domains: numerical kernels in scientific, signal processing and image processing applications. However, it turns out that the underlying array programming paradigm is suitable for a much broader range of application areas including financial modelling and data mining.

Our predominant motivation with SAC is to demonstrate how we can combine high-level declarative array programming with a runtime performance that is competitive to imperative solutions. The compiler technology that we have developed in this context aggressively exploits the conceptual advantages of the declarative paradigm for large-scale program transformation. Indeed, we manage to achieve sequential runtime performance levels that are competitive with FORTRAN code [6]. Compiler-directed generation of multithreaded code does not only allow SAC programs to take advantage of multicore processors and shared memory systems thereof without any additional programming effort [7], but given the competitive sequential performance generates real speedups over existing solutions [8, 9].

In this paper we summarise the key compiler technology that we have developed in order to be able to compile SAC-programs ef-

[1]The author is currently affiliated with the University of Hertfordshire (email: c.grelck@herts.ac.uk).

DAMP 2007 16 January 2007, Nice, France.

ficiently into multithreaded code for shared memory architectures. Since this architectural model matches today's mass market multicore processor architectures, we have a ready-to-use technology for harnessing the full power of these machines without enforcing the programmer to be aware of the target architecture in mind.

The remainder of this paper is organised as follows. We begin with a brief introduction to SAC in Section 2. It focuses on WITH-loops, the central data parallel construct. They prepare the grounds for an array-oriented programming style, which we illustrate in Section 3. A consequent application of this programming style leads to both: highly abstract code and intensive use of WITH-loops. How we actually compile these WITH-loops into multithreaded code is outlined in Section 4. Section 5 outlines the key optimisation techniques that improve the granularity of individual threads, and Section 6 describes the design of our memory management subsystem, which has proved crucial for achieving high performance. Eventually, we draw conclusions in Section 7.

2. INTRODUCING SAC

As the name "Single Assignment C" suggests, SAC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This is meant to facilitate adaptation for programmers with a background in imperative languages, the prevalent paradigm in our targeted application domains. Core SAC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions; details can be found in [1]. Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SAC code while the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation.

On top of this language kernel SAC provides genuine support for processing truly multidimensional and truly stateless/functional arrays using a shape-generic style of programming. Any SAC expression evaluates to an array. Arrays may be passed between functions without restrictions. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]`, and arrays of any rank, e.g. `int[*]`. The latter include scalars, which we consider rank-0 arrays with an empty shape vector. For convenience and equivalence with C we use `int` rather than the equivalent `int[]` as a type notation for scalars. The hierarchy of array types induces a subtype relationship, and SAC supports function overloading with respect to subtyping.

SAC only provides a small set of built-in array operations. Essentially, there are primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`) or its shape (`shape(`*array*`)`). A selection facility provides access to individual elements or entire subarrays using a familiar square bracket notation: *array*`[`*idxvec*`]`.

All aggregate array operations are specified using WITH-loop expressions, a SAC-specific array comprehension:

```
with {
  ( lower_bound <= idxvec < upper_bound ) : expr;
}: genarray( shape, default)
```

Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to loop variables in FOR-loops. However, we deliberately do not define any order on these index sets. We call the specification of such an index set a *generator* and associate it with some potentially complex SAC expression. Thus, we create a mapping between index vectors and values, in other words an array. As an example, consider the WITH-loop

```
with {
  ([0,0] <= iv < [3,5]) : 42;
}: genarray( [3,5], 0)
```

that defines a 3×5 matrix with all elements uniformly set to 42. The scope of *idxvec* (here named `iv`) is confined to the expression associated with the generator. It can be used to access the current index location. For example, the WITH-loop

```
with {
  ([0] <= iv < [5]) : iv[0];
}: genarray( [5], 0)
```

computes the vector `[0,1,2,3,4]`. Note that `iv` denotes a 1-element vector rather than a scalar. Therefore, we need to select the first (and only) element from `iv` to achieve the desired result. Actually, it is not the generator that defines the shape of the resulting array, but the first expression following the key word `genarray`. So far, the two have always coincided, but for example

```
with {
  ([1] <= iv < [4]) : 42;
}: genarray( [5], 0)
```

computes the vector `[0,42,42,42,0]`. We still create a 5-element vector, but only the three inner elements are defined as 42 while all others are set to the *default value*, which is given by the second expression following the key word `genarray`. Since the default expression is not within the scope of a generator, it has no access to the index. Hence, all array elements not covered by any generator are guaranteed to have the same value.

WITH-loops are not limited to a single generator. For example, the WITH-loop

```
with {
  ([1] <= iv < [4]) : 1;
  ([3] <= iv < [5]) : 2;
}: genarray( [6], 0)
```

defines the vector `[0,1,1,2,2,0]`. All elements of the resulting array still not covered by any of the generators are initialised with the value of the default expression, 0 in the example. Whenever the index sets defined by the various generators are not pairwise disjoint, the order of the generators matters: in the example the array's value at index location `[3]`, which is covered by both generators is set to 2 rather than to 1, i.e., the last generator dominates.

SAC actually features several variants of WITH-loops. Let us assume we have named the array defined by the previous WITH-loop `A`. Then, the `modarray`-WITH-loop

```
with {
  ([0] <= iv < [3]) : 3;
}: modarray( A)
```

computes the vector `[3,3,3,2,2,0]`. More precisely, it computes a new array that has exactly the same shape as the existing array referred to by the expression following the key word `modarray`. The computation of those elements covered by one or more generators follows exactly the same pattern as in the case of `genarray`-WITH-loops, but the remaining elements are defined by the values of the corresponding elements in the referenced array rather than by a common default value.

Another WITH-loop variant supports the definition of reduction operations. For example, the fold-WITH-loop

```
with {
  (iv < shape(A)) : A[iv];
}: fold( +, 0)
```

computes the sum of all elements of an existing array A. The key word fold is followed by the name of a binary associative and commutative function and an expression defining that function's neutral element. In analogy to the other WITH-loop variants, we compute a set of values, here the elements of A, and eventually apply the given fold operation. Note, that we have omitted the lower bound in the generator. For all WITH-loops this defaults to the zero vector of appropriate length. Likewise, we may omit the specification of an upper bound in genarray- and modarray-WITH-loops. If so, the upper bound coincides with the shape of the array to be created. Furthermore, the specification of a neutral element in a fold-WITH-loop may be omitted if the fold operation is built-in.

Another extension of WITH-loops that has significant consequences for the complexity of the compilation process affects generators. In addition to dense rectangular index ranges, as in all examples shown so far, SAC supports regular index grids within rectangular boundaries: Generators optionally feature a *step vector* that determines the periodicity in each dimension and a *width vector* that describes the block size within the periodic pattern. For example,

```
with {
 ([1] <= iv < [11] step [4] width [2]) : 1;
}: genarray( [12], 0)
```

yields the vector [0,1,1,0,0,1,1,0,0,1,1,0]. Like the boundary vectors step and width vectors are in fact expression positions and may be computed from function arguments at runtime.

3. PROGRAMMING METHODOLOGY

SAC propagates a programming methodology based on the principles of abstraction and composition. Rather than building entire application programs directly by means of WITH-loops, we merely utilise them to define small abstractions with a well defined meaning. They in turn form the building blocks for constructing more complex operations and eventually entire application programs by composition.

Fig. 1 illustrates the principle of abstraction by rank-invariant definitions of three common array operations. The overloaded definitions of the function abs and the infix operator >= extend the corresponding scalar functions to arrays of any rank and shape. The function any is a standard reduction operation, which yields true if any of the argument array elements is true, otherwise it yields false. Note that SAC only requires the annotation of types in function signatures while the compiler infers types of local variables.

In analogy to the examples in Fig. 1, the SAC standard library provides a plethora of array operations similar to the built-ins of APL [10, 11], J [12], NIAL [13] or FORTRAN-90. Among them are element-wise extensions of arithmetic and relational operators, typical reduction operations like sum and product, various subarray selection facilities as well as shift and rotate operations and many more.

Basic array operations defined by WITH-loops lay the foundation for constructing more complex operations by means of composition. Fig. 2 illustrates this principle by a generic convergence criterion for iterative algorithms of any kind, which is entirely defined by composition of basic array operations.

```
double[*] abs (double[*] a)
{
  res =  with {
           (iv) : abs( a[iv]);
         }: genarray( shape(a), 0);
  return( res);
}

bool[*] (>=) (double[*] a, double[*] b)
{
  res = with {
          (iv) : a[iv] >= b[iv];
        }: genarray( min( shape(a),
                          shape(b)), 0);
  return( res);
}

bool any (bool[*] a)
{
  res = with {
          (iv < shape(a)) : a[iv];
        }: fold( ||, false);
  return( res);
}
```

Figure 1: Defining rank-invariant array operations

The example of the shape-generic convergence criterion nicely demonstrate the power of shape-generic array programming: One may read the definition of the function continue as if it was applied to scalar arguments, and in fact it can be applied to scalars. However, the shape-generic definition of the individual array operations used as building blocks immediately makes the whole function applicable to arrays of arbitrary rank. Not only does this technique liberate programs from loop nestings and explicit indexing that obfuscate the true functionality of an operation, but it also makes the function more generally applicable and more easily maintainable.

```
bool continue (double[*] new,
               double[*] old,
               double eps)
{
  return( any( abs( new - old) >= eps));
}
```

Figure 2: Defining array operations by composition

4. EXPLOITING CONCURRENCY

SAC programs offer essentially two different sources of concurrency. Thanks to the functional semantics, the evaluation order of subexpressions is only limited by data dependencies. Although semicolon-separated lists of assignments, as adopted from C, insinuate a certain evaluation order to programmers with an imperative mindset, the SAC compiler is actually free to reorder them for optimisation purposes or to execute them in parallel if desired. This is in stark contrast to the similar looking do-notation of HASKELL, where semicolons are used to express a defined execution order in an otherwise purely functional context. The second source of concurrency in SAC programs are WITH-loops. Throughout Section 2 we stressed the fact that generators define *sets* of indices

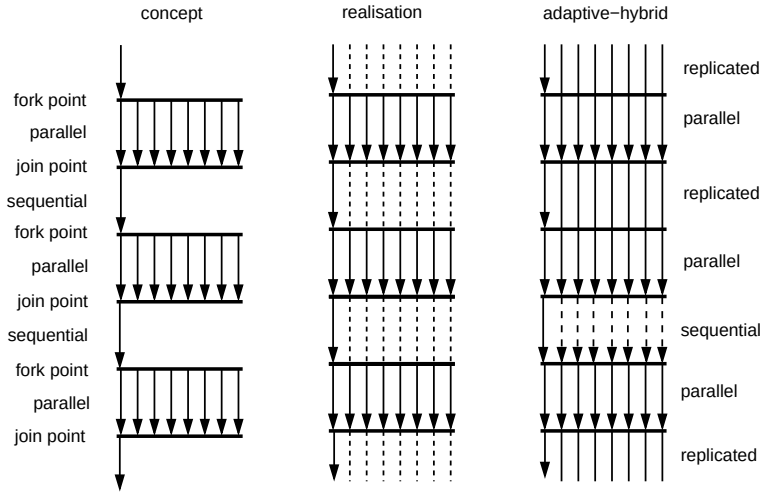


Figure 3: Multithreaded execution models

and, hence, WITH-loops describe sets of computations, either to initialise the elements of a new array or as basis for a reduction operation.

During the last two (if not more) decades a lot of research has gone into harnessing the first form of concurrency. The bottom line, however, is that an implicit approach to parallelisation is not feasible in practice. Although it is rather cheap to identify concurrently executable subexpressions, it turns out to be extremely difficult to identify those ones that actually justify their parallel execution by paying off the corresponding overhead for synchronisation and communication [14, 15, 16].

As a consequence, we utilise concurrency on the general expression level only for optimisation purposes and entirely focus on WITH-loops for parallelisation. Since any (library-defined) array operation in SAC in one way or another boils down to a set of WITH-loops, they are ubiquitous in intermediate SAC code and account for the vast majority of program execution time. From a compiler writer's perspective they offer a standardised interface to express a wide range of array operations in a data parallel way. For each element of the multidimensional index space the corresponding expression is independent of all others. In contrast to "parallel" loops in imperative languages, the functional semantics of SAC formally guarantees the absence of hidden dependencies.

We call the smallest entity of execution a *microthread*. A microthread evaluates exactly one associated expression for a single element of the index set. The data parallel approach usually leads to a massive unfolding of concurrency: The number of microthreads cooperatively executing an individual WITH-loop typically exceeds the number of available computing resources by orders of magnitude, leading to our first compilation challenge: In order to actually exploit multiple processing cores we need the assistance of the operating system. More precisely, we must express our computation in terms of operating system threads, which are then scheduled by the operating system for concurrent execution onto processor cores. Unfortunately, system level threads are usually quite expensive, and, hence, a one-to-one mapping of our microthreads to operating system threads is prohibitive. Our experience shows that employing a number of system threads similar to the number of independent computing resources is more likely to yield satisfactory performance. Therefore, we need to schedule microthreads to operating system threads. Three issues are essential for the successful scheduling of microthreads:

- Parallel execution of a WITH-loop leads to a synchronisation barrier. Hence, we need to schedule microthreads to system threads in a way that smoothly balances the computational workload.

- In our context the code associated with each microthread is dominated by array accesses, which more or less directly map to memory load instructions in compiled code. As a consequence, scheduling policies must take data locality into account.

- Thanks to multiple generators with periodic step and width specifications, loop structures in compiled code can already be complicated in the purely sequential case. Therefore, we must orthogonalise the realisation of microthread scheduling from the underlying code.

We have experimented with various scheduling policies; details can be found in [7]. Approaches with a central work queue from which system threads take microthreads for execution proved clearly unsatisfactory. This is not so much an effect of congestion upon access to the shared work queue, but essentially due to poor data locality.

Eventually, we have adopted two different scheduling policies: a purely static and a semi-dynamic one. Our experience shows that in many array processing codes workload is actually fairly evenly distributed among microthreads. As a consequence, static policies that subdivide the index space of microthreads into evenly sized rectangular blocks, one per system thread, perform reasonably well, in particular as they incur very little scheduling overhead at runtime.

Our semi-dynamic scheduling policy is inspired by affinity loop scheduling techniques [17, 18]; it is based on an a-priori static allocation of work similar to the static policy. However, each system thread further subdivides its pre-allocated index space of microthreads into a private and a public subspace: Whereas each system thread definitely executes the microthreads in its private subspace, it may actually share the workload associated with its public subspace. As soon as a system thread has completed all its pre-allocated microthreads, both from its private and from its public index subspace, it commences "stealing" microthreads from other system thread's public index subspaces.

Based on the assumption that the number of microthreads greatly exceeds the number of processor cores, we restrict ourselves to a

single level of parallelism. Hence, program execution is always in one of two states: either a single *master thread* executes the sequential part of the program or a fixed number of *worker threads* cooperatively execute a single WITH-loop. This is illustrated on the left hand side of Fig. 3. WITH-loops nested within others do not lead to a further unfolding of parallel activity, but are executed sequentially. This restriction allows us to come up with a much leaner and more efficient runtime system.

Since the number of worker threads is a runtime constant, the runtime system does not terminate worker threads after having completed one WITH-loop and restart them for computing the following one. In fact, our realisation of the fork/join execution model starts all worker threads at once upon program startup and keeps them alive until the whole program terminates. Fork and join points are implemented by tailor-made synchronisation primitives, as illustrated in the centre of Fig. 3. A detailed description of our multi-threaded runtime system can be found in [7].

Currently, we are experimenting with a less restricted execution model, which we illustrate on the right hand side of Fig. 3. In this model, which we call *adaptive-hybrid*, sequential code sections between two WITH-loops can be executed either by the master thread as before or in a replicated way by all worker threads (hence the term "hybrid"). The advantage of replication is that it reduces synchronisation and communication requirements. For example, replicating the computation of some value needed by all worker threads saves the synchronisation cost of having the worker threads wait for the master thread to finish computing that value and the communication cost of making the computed value available to the worker threads.

Unfortunately, replication of sequential code is not always feasible because we need to preserve the observable behaviour of the program and code may interact with the execution environment in one or another way. As a simple example consider code that produces some output. This output must appear only once and, hence, the code cannot be replicated. We decide about replicated or sequential execution in each individual case based on code analysis (hence the term "adaptive"). More details about this novel approach can be found in [19].

5. AGGREGATION OF WITH-LOOPS

Our programming methodology based on the composition of independently developed and tested components to larger components has its obvious merits with respect to code reuse and software engineering principles. Unfortunately, it also has a drawback when it comes to parallel execution: Each individual microthread only executes a small number of instructions. Take the convergence criterion introduced in Section 3 as an example. Applied to reasonably large arrays, it offers a huge amount of concurrency. However, the amount of computation per array element in each of the four WITH-loops is almost negligible. This situation adversely affects the ratio between productive computation and organisational overhead in case of parallel execution and inevitably leads to sub-optimal parallel performance.

To remedy this situation we have developed a compiler optimisation framework that systematically aggregates compositions of computationally light-weight WITH-loops into single, computationally heavy-weight WITH-loops. We have identified three different types of composition and address each by a tailor-made compiler optimisation. They are accompanied by and owe much of their effectiveness to a large number of standard optimisation techniques like function inlining, constant folding, loop unrolling, loop invariant removal or variable propagation to name just a few [1, 2].

Our first optimisation is named WITH-*loop folding*. It is similar in spirit to deforestation techniques [20, 21, 22] developed in the context of general-purpose functional languages and addresses computational pipelines like the one in the convergence criterion example: the result of each computational step, which is represented by a WITH-loop, becomes the argument of the subsequent step. Executed naively, this code results in the costly creation of three intermediate arrays before the final boolean result is computed. Given that individual WITH-loops are the basis for parallel program execution, the corresponding organisational overhead also arises three times. WITH-loop folding effectively performs a forward substitution of expressions from top to bottom, thus replacing the selection of an element from an intermediate array by the defining expression itself. Fig. 4 shows the result of applying WITH-loop folding (and of course function inlining) to the convergence criterion example: we compute the entire predicate by a single WITH-loop, that traverses both argument arrays exactly once and does not create any intermediate data structures.

```
bool continue (double[*] new,
               double[*] old,
               double eps)
{
  r = with {
       (iv < min( shape(new), shape(old))):
          abs( new[iv] - old[iv]) >= eps;
      }: fold( ||, false);
  return( r);
}
```

Figure 4: Intermediate representation of convergence criterion after WITH-loop folding

The convergence criterion only demonstrates the most simple optimisation case handled by WITH-loop folding: Each WITH-loop has a single generator only, the generators are all identical and there is no computation on the index variable in the associated expressions. Fig. 5 shows the more challenging example of a simple 1-dimensional relaxation kernel. The original code implicitly contains a total of 7 WITH-loops: two for each application of `rotate` and one for each arithmetic operator. By means of WITH-loop folding, supported by a large number of standard optimisations, this implementation of `relax` is internally transformed into the single WITH-loop shown in Fig. 5. For additional information on WITH-loop folding see [23].

Our second WITH-loop-specific optimisation is called WITH-*loop fusion*. It is similar in spirit to tupling techniques [24] in general-purpose functional languages or conventional loop fusion [25, 26] in imperative languages. WITH-loop fusion primarily addresses pairs of WITH-loops that, unlike in the above cases, have no data dependencies. Take as an example the definition of the function `minmaxval` in Fig. 6. Following our programming methodology its implementation is based on the composition of two simpler functions, `minval` and `maxval`, that in turn are implemented using WITH-loops. Executed naively, each search for a minimum or a maximum value is done in parallel, but strictly one after the other. The argument array is traversed twice and there is a synchronisation barrier in between.

WITH-loop fusion transforms the two original WITH-loops into a single one. More precisely, the generated WITH-loop is an internal extension of language-level WITH-loops. It has two operations (`fold(min)` and `fold(max)`) and, likewise, the generator is associated with two expressions (`a[iv]` and again `a[iv]`) and the entire WITH-loop yields two values (`minv` and `maxv`). It comes handy that

```
double[.] relax( double[.] v)
{
  return( (rotate( 1, v) + v + rotate( -1, v)) / 3.0);
}
```


```
double[.] relax( double[.] v)
{
  r = with {
        ([0]        <= iv < [1])        : (v[shape(iv)-1] + v[iv] + v[iv+1]) / 3.0;
        ([1]        <= iv < shape(v)-1) : (v[iv-1] + v[iv] + v[iv+1]) / 3.0;
        (shape(v)-1 <= iv < shape(v))   : (v[iv-1] + v[iv] + v[iv-shape(v)+1]) / 3.0;
      }: modarray( v) );
  return( r);
}
```

Figure 5: Effect of WITH-loop folding on 1-dimensional relaxation kernel

SAC also features functions that yield multiple values, as shown in Fig. 6. Further applications of standard optimisations, in particular common subexpression elimination, turn the pair of expressions `(a[iv],a[iv])` into a single expression block of the form

```
{ val = a[iv]; return( val, val); }
```

This further transformation reduces the number of memory load instructions by one half.

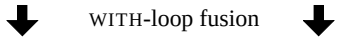
```
double minval( double[*] a)
{
  minv = with {
           (iv) : a[iv];
         }: fold( min);
  return( minv);
}

double maxval( double[*] a)
{
  maxv = with {
           (iv) : a[iv];
         }: fold( max);
  return( maxv);
}

double, double minmaxval( double[*] a)
{
  return( minval( a), maxval( a));
}
```


```
double, double minmaxval( double[*] a)
{
  minv, maxv = with {
                 (iv) : (a[iv], a[iv]);
               }: (fold( min), fold( max));
  return( minv, maxv);
}
```

Figure 6: Effect of WITH-loop fusion on function computing the minimum and the maximum value of an array

Again, the optimisation challenge lies in WITH-loops that are more complex than an introductory example: multiple generator-expression pairs, affine functions on the index variables, periodic generators, different operators, restricted data dependencies, etc. Fig. 7 shows a more challenging example: We combine the relaxation kernel of Fig. 5 with the convergence criterion of Section 3. Following a series of WITH-loop folding steps within each individual part of the computation, WITH-loop fusion (by the help of many

```
double[.], bool relaxstep( double[.] old,
                           double eps)
{
  new = relax( v);
  con = continue( new, old, eps);
  return( new, con);
}
```




```
double[.], bool relaxstep( double[.] old,
                           double eps)
{
 res, con =
  with {
   ([0]          <= iv < [1])          : {
     ...
   }
   ([1]          <= iv < shape(old)-1) : {
     new = (old[iv-1] + old[iv]
            + old[iv+1]) / 3.0;
     con = abs( new - old[iv]) >= eps;
     return( new, con);
   }
   (shape(old)-1 <= iv < shape(old))   : {
     ...
   }
  }: (modarray( old), fold( ||));
 return( res, con);
}
```

Figure 7: Effect of WITH-loop fusion on relaxation step with convergence test

standard optimisations) eventually manages to fuse the two remaining WITH-loops into a single one that computes both the new array and the convergence predicate in a joint step. A formal definition of WITH-loop fusion can be found in [27].

The last of our three WITH-loop condensing optimisation techniques is WITH-loop scalarisation. It addresses WITH-loops that are nested within each other and aims at creating WITH-loops that always operate on the element level of arrays. Nested WITH-loops typically arise whenever the element type of argument arrays is not one of the built-in scalar types, e.g. `int` or `double`, but itself a user-defined array type. Fig. 8 shows an excerpt from the SAC standard library module for complex numbers. We introduce complex numbers as 2-element `double` vectors and overload the plus operator with two further instances for complex numbers and arrays of complex numbers. Whereas the former instance uses casting to `double` vectors and the corresponding addition on them, the latter very much resembles the familiar pattern of element-wise array operations.

```
typedef double[2] complex;

complex (+) (complex a, complex b)
{
  return( (:complex) (((:double[2]) a)
                      + ((:double[2]) b)));
}

complex[*] (+) (complex[*] a, complex[*] b)
{
  r = with {
        (iv) : a[iv] + b[iv];
      }: genarray( min( shape(a),
                        shape(b)),
                   (:complex)[0.0,0.0]);
  return( r);
}
```

standard optimisations

```
complex[*] (+) (complex[*] a, complex[*] b)
{
  r = with {
        (iv) : with {
                 (cv) : a[iv][cv]
                        + b[iv][cv];
               }: genarray( [2], 0.0);
      }: genarray( min( shape(a),
                        shape(b)), ...);
  return( r);
}
```

WITH-loop scalarisation

```
double[*] (+) (double[*] a, double[*] b)
{
  r = with {
        (iv) : a[iv] + b[iv];
      }: genarray( min( shape(a),
                        shape(b)), ...);
  return( r);
}
```

Figure 8: Effect of WITH-loop scalarisation on arithmetic operations on arrays of complex numbers

WITH-loop scalarisation replaces the type `complex` by its definition and increases the rank of arrays as necessary. Likewise, the index space of the scalarised WITH-loop on the bottom of Fig. 8 is increased by an additional dimension. Unfortunately, the type system of SAC currently is not expressive enough to capture the fixed extent of two elements along the innermost axis of arrays while leaving the extent unspecified for all other axes. The effect of WITH-loop scalarisation is twofold: Firstly, it avoids the creation of temporary arrays for each element of the index space of the outer WITH-loop. Secondly, it makes scheduling of microthreads more flexible. A formal definition of WITH-loop scalarisation can be found in [28].

6. MEMORY MANAGEMENT

At first glance, implicit memory management does not seem to be a particularly special feature for a functional programming language like SAC. All functional languages and even modern imperative languages like Java or C# feature automatic garbage collection. The prevailing technique is tracing garbage collection [29]. The principle is as follows: Allocation of memory reduces to pushing a global pointer forward by the amount of memory requested. Memory is typically not de-allocated explicitly. Instead, allocation continues until all available memory is exhausted, upon which execution of the program itself is temporarily suspended for a garbage collection cycle. The garbage collector traces (hence the name) the entire program graph to mark all data that is still needed. Afterwards, the heap is reorganised such that all remaining data structures are at the beginning of the address space and the allocation pointer is reset to the first available address.

Unfortunately, tracing garbage collection is unsuitable for an array language like SAC. Firstly, arrays tend to be large. It is not uncommon for SAC programs that 90% of the heap is allocated to only a few different arrays. Re-locating these arrays in a garbage collection cycle is prohibitively expensive. Secondly, tracing garbage collection provides little help to overcome the *aggregate update problem* [30]. In a functional environment, data structures never change their values. Instead, new data structures are created that hold the newly computed values and typically also some values copied from the old instance of the data structure. This can be implemented fairly efficiently for the typical functional data structures like lists and trees, that are made up of many small cells. Instead of copying the whole list or tree, one only needs to copy individual cells and update a few pointers.

The opposite is true for arrays. Naively updating an array element in a functionally sound way requires copying the whole array, which is clearly prohibitive in terms of performance. Copying could be avoided though if we knew that the original array is not needed anywhere else and, hence, would immediately become garbage after the update operation. Unfortunately, this information is not available with tracing garbage collection unless provided through type system features like state monads [31] or uniqueness types [32]. However, these techniques also enforce a very imperative coding style on the programmer that runs counter the idea of declarative programming.

To overcome these pitfalls the SAC memory management subsystem is based on reference counting [33] and active heap management. Each array is associated with an additional counter that keeps track of the number of active references to this array. Compiled SAC code is augmented with instructions to increment and decrement these counters as necessary. As soon as the counter is

decremented to zero, the array and the counter can safely be de-allocated.

To a similar extent as tracing garbage collection proves unsuitable for SAC, the good reasons for not using reference counting in other environments vanish. Arrays in SAC are non-cyclic and, at least for the time being, also unnested. This avoids costly counter update and memory de-allocation cascades. Although we intend to support nested arrays in the future, we assume the typical nesting depth to be small. Moreover, reasonably large arrays make maintenance overhead for reference counters negligible, both in terms of extra storage for the counter and extra code for its manipulation.

Most importantly, reference counting provides us with a solution to the aggregate update problem. Before executing an element update operation, we consult the current state of the array's reference counter and only copy the whole array if necessary. In fact, reference counting information can be useful in a variety of situations. For example, the element-wise summation of two equally shaped arrays yields an entirely new array. Nevertheless, if the reference counter of one of the argument arrays equals one, we can safely reuse that array's memory for constructing the result array. As a consequence, we not only spare memory allocation and de-allocation overhead, but we also reduce the memory footprint of the operation by one third.

While the general case requires reference counters to be incremented and decremented at runtime, static analysis can do a lot to reduce the actual number of these operations. More importantly, static analysis often allows us to decide at compile time about memory de-allocation or reuse. In [34] we provide a basic scheme for augmenting SAC code with reference counting instructions and describe a number of optimisations that reduce the incurred runtime overhead through static analysis of code properties.

Unlike tracing garbage collection, reference counting relies on an additional memory allocator to organise the heap, from which memory is allocated and to which memory is released in an arbitrary sequence of allocation and de-allocation requests. Whereas efficient allocators are available for sequential execution, multithreaded access to a shared heap requires proper synchronisation. While it is simple to synchronise each allocation and de-allocation operation using a global lock, it also immediately becomes a concurrency bottleneck. In fact, we must organise the heap in a way that reduces locking to a minimum. To achieve this SAC uses its own memory allocator that is integrated with the multithreaded runtime system. This design allows us to exploit various side conditions and invariants that a general-purpose multithreaded memory allocator cannot rely on.

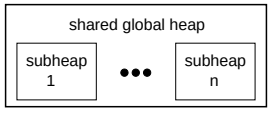


Figure 9: Architecture of parallel memory allocator

Fig. 9 illustrates the architecture of our memory allocator. Within the global address space the SAC memory allocator pre-allocates subheaps for each thread. These subheaps are exclusively available to the corresponding worker thread and, thus, can be accessed without synchronisation. Thread-specific subheaps are used for memory requests up to a certain configurable size. Only if a worker thread needs to allocate a chunk of memory that exceeds this size or if a worker thread needs to extend its subheap from the global heap, synchronisation is required. However, such events are rare in practice.

The organisation of the heap into concurrently accessible subheaps also solves the problem of false sharing [35] because worker threads use disjoint sections of the address space to allocate small data structures. Furthermore, our runtime system ensures that each worker thread can identify itself and directly access its private heap. In contrast, the restricted standard interface for memory allocators requires general-purpose solutions to rely on expensive thread-specific global data to access such information.

We can also statically distinguish between code sections that are executed in a single threaded manner and those that are actually executed in parallel. Therefore, we can entirely avoid synchronisation even on access to the global heap in many situations. This tight integration between multithreaded runtime system and heap organisation proved essential for achieving high performance with applications that require frequent allocation and de-allocation of arrays; details can be found in [36].

7. CONCLUSION

The ubiquity of parallelism in current and future computing hardware requires new programming models that expose concurrency at the right level of abstraction and new compilation technology that efficiently maps programs to execution machinery. We have presented the functional language SAC and its concept of generic, compositional array programming as one such approach. Furthermore, we have identified the major challenges in compiling SAC code into efficient code for modern multicore processors and outlined the most important aspects of the compilation framework that we have developed for SAC:

- efficient organisation of multithreaded program execution and aggregation of microthreads to system threads,
- systematic aggregation of compositions of computationally light-weight WITH-loops into fewer, computationally intensive WITH-loops,
- efficient memory management based on reference counting and active administration of the shared heap.

Space does not permit us to quantify the effects of our various techniques on the runtime performance of compiled SAC programs. However, several case studies [8, 9, 6] have shown that despite a generic, compositional style of programming and the use of functional stateless arrays with implicit memory management, our compilation framework succeeds in competing well with hand-optimised FORTRAN code. Moreover, our fully compiler-directed parallelisation yields substantial additional performance gains on symmetric shared memory multiprocessor architectures including today's off-the-shelf multicore processors. By matching high sequential performance with implicit parallelisation SAC realises real speedups over existing low-level solutions, whose manual parallelisation would be labour-intensive and costly.

8. REFERENCES

[1] Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. Journal of Functional Programming **13** (2003) 1005–1059

[2] Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427

[3] Cann, D.: Retire Fortran? A Debate Rekindled. Communications of the ACM **35** (1992) 81–89

[4] Blelloch, G.E.: Programming Parallel Algorithms. Communications of the ACM **39** (1996)

[5] International Standards Organization: Programming Language APL, Extended. ISO N93.03, ISO (1993)
[6] Shafarenko, A., Scholz, S.B., Herhut, S., Grelck, C., Trojahner, K.: Implementing a numerical solution of the KPI equation using Single Assignment C: lessons and experiences. In: Implementation and Application of Functional Languages, 17th International Workshop (IFL'05). Dublin, Ireland, Revised Selected Papers. Lecture Notes in Computer Science vol. 4015, Springer-Verlag (2006) 160–177
[7] Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
[8] Grelck, C.: Implementing the NAS Benchmark MG in SAC. In: 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, USA, IEEE Press (2002)
[9] Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In: Parallel Computing Technologies, 7th International Conference (PaCT'03), Nizhni Novgorod, Russia. Lecture Notes in Computer Science vol. 2763, Springer-Verlag (2003)
[10] Iverson, K.: A Programming Language. John Wiley, New York, USA (1962)
[11] Falkoff, A., Iverson, K.: The Design of APL. IBM Journal of Research and Development **17** (1973)
[12] Iverson, K.: J Introduction and Dictionary. Iverson Software Inc., Toronto, Canada. (1995)
[13] Jenkins, M.: Q'Nial: A Portable Interpreter for the Nested Interactive Array Language Nial. Software Practice and Experience **19** (1989) 111–126
[14] Hammond, K.: Parallel Functional Programming: An Introduction. In: International Symposium on Parallel Symbolic Computation (PASCO'94), Linz, Austria, World Scientific Publishing (1994) 181–193
[15] Trinder, P., Hammond, K., Loidl, H.W., Jones, S.P.: Algorithm + Strategy = Parallelism. Journal of Functional Programming **8** (1998) 23–60
[16] Hammond, K., Michaelson, G., eds.: Research Directions in Parallel Functional Programming. Springer-Verlag (1999)
[17] Markatos, E., LeBlanc, T.: Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems **5** (1994) 379–400
[18] Yan, Y., Jin, C., Zhang, X.: Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 70–81
[19] Grelck, C., Kuthe, S., Scholz, S.B.: A Hybrid Shared Memory Execution Model for a Data Parallel Language with I/O. Parallel Processing Letters, to appear.
[20] Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. Theoretical Computer Science **73** (1990) 231–248
[21] Gill, A., Launchbury, J., Peyton Jones, S.: A Short Cut to Deforestation. In: Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark, ACM Press (1993) 223–232
[22] van Arkel, D., van Groningen, J., Smetsers, S.: Fusion in Practice. In: Implementation of Functional Languages, 14th International Workshop (IFL'02), Madrid, Spain, Selected Papers. Lecture Notes in Computer Science vol. 2670, Springer-Verlag (2003) 51–67
[23] Scholz, S.B.: With-loop-folding in SAC — Condensing Consecutive Array Operations. In: Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers. Lecture Notes in Computer Science vol. 1467, Springer-Verlag (1998) 72–92
[24] Chin, W.: Towards an Automated Tupling Strategy. In: ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'97), Copenhagen, Denmark, ACM Press (1993) 119–132
[25] Bacon, D., Graham, S., Sharp, O.: Compiler Transformations for High-Performance Computing. ACM Computing Surveys **26** (1994) 345–420
[26] Manjikian, N., Abdelrahman, T.: Fusion of Loops for Parallelism and Locality. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 193–209
[27] Grelck, C., Hinckfuß, K., Scholz, S.B.: With-Loop Fusion for Data Locality and Parallelism. In: Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland, Revised Selected Papers. Lecture Notes in Computer Science vol. 4015, Springer-Verlag (2006) 178–195
[28] Grelck, C., Scholz, S.B., Trojahner, K.: With-Loop Scalarization: Merging Nested Array Operations. In: Implementation of Functional Languages, 15th International Workshop (IFL'03), Edinburgh, UK, Revised Selected Papers. Lecture Notes in Computer Science vol. 3145, Springer-Verlag (2004)
[29] Jones, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons (1999)
[30] Hudak, P., Bloss, A.: The Aggregate Update Problem in Functional Programming Systems. In: 12th ACM Symposium on Principles of Programming Languages (POPL'85), New Orleans, USA, ACM Press (1985) 300–313
[31] Peyton Jones, S., Wadler, P.: Imperative Functional Programming. In: 20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston, USA, ACM Press (1993) 71–84
[32] Barendsen, E., Smetsers, S.: Conventional and Uniqueness Typing in Graph Rewrite Systems. In: 13th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'93), Bombay, India. Lecture Notes in Computer Science vol. 761, Springer-Verlag (1993) 41–51
[33] Collins, G.E.: A Method for Overlapping and Erasure of Lists. Communications of the ACM **3** (1960)
[34] Grelck, C., Trojahner, K.: Implicit Memory Management for SAC. In: Implementation and Application of Functional Languages, 16th International Workshop (IFL'04). University of Kiel, Institute of Computer Science and Applied Mathematics, Technical Report 0408 (2004) 335–348
[35] Torellas, J., Lam, M., Hennessy, J.: False Sharing and Spatial Locality in Multiprocessor Caches. IEEE Transactions on Computers **43** (1994) 651–663
[36] Grelck, C.: Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC. Doctoral dissertation, Institute of Computer Science and Applied Mathematics, University of Kiel, Logos Verlag, Berlin (2001)