Controlling Chaos *

On Safe Side-Effects in Data-Parallel Operations

Stephan Herhut Sven-Bodo Scholz Clemens Grelck

University of Hertfordshire
School of Computer Science
Hatfield, United Kingdom
{s.a.herhut,s.scholz,c.grelck}@herts.ac.uk

Abstract

With the rising variety of hardware designs for multi-core systems, the effectiveness in exploiting implicit concurrency of programs plays a more vital role for programming such systems than ever before. We believe that a combination of a data-parallel approach with a declarative programming-style is up to that task: Data-parallel approaches are known to enable compilers to make efficient use of multi-processors without requiring low-level program annotations. Combining the data-parallel approach with a declarative programming-style guarantees semantic equivalence between sequential and concurrent executions of data parallel operations. Furthermore, the side-effect free setting and explicit model of dependencies enables compilers to maximise the size of the data-parallel program sections.

However, the strength of the rigidity of the declarative approach also constitutes its weakness: Being bound to observe all data dependencies categorically rules out the use of side-effecting operations within data-parallel sections. Not only does this limit the size of these regions in certain situations, but it may also hamper an effective workload distribution. Considering side effects such as plotting individual pixels of an image or output for debugging purposes, there are situations where a non-deterministic order of side-effects would not be considered harmful at all.

We propose a mechanism for enabling such non-determinism on the execution of side-effecting operations within data-parallel sections without sacrificing the side-effect free setting in general. Outside of the data-parallel sections we ensure single-threading of side-effecting operations using uniqueness typing. Within data-parallel operations however we allow the side-effecting operations of different threads to occur in any order, as long as effects of different threads are not interleaved. Furthermore, we still model the dependencies arising from the manipulated states within the data parallel sections. This measure preserves the explicitness of all data dependencies and therefore it preserves the transformational potential of any restructuring compiler.

* This work was funded by the European Union Apple-CORE project grant no. FP7 215216.

DAMP'09, January 20, 2009, Savannah, Georgia, USA.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Design, Languages, Performance

1. Introduction

Looking at the recent architectures of the main processor manufacturers, there is no doubt that multi-core has made it to the main stream. There has been a shift from ever increasing clock frequencies to increasing numbers of computing cores per processor. Be it homogeneous multi-core architectures with two, three, four and soon eight cores like Intel's Core range of CPUs (Gochman et al. 2006), AMD's multi-core Opteron processors (Advanced Micro Devices 2008) and Sun's Niagara (Kongetira et al. 2005), or inhomogeneous designs like the IBM Cell (Chen et al. 2007) and Cyclops (Almási et al. 2003) processors. Even rather specialized graphics processors are being used for numerical applications due to their wide SIMD architecture (Garland et al. 2008). And the future is promising even more potential for truly concurrent program execution with many-core processors with up to 100 cores being discussed (Held et al. 2006).

However, although theoretically program performance might scale linearly with the number of cores, the reality is gloom. Taming concurrency and exploiting massively parallel architectures for meaningful computation is a difficult and tedious task, so far only pursued by specialized and highly skilled programmers. For multi- and many-core systems to really arrive in the mainstream, a new, less complex and thus less error-prone programming model is needed.

A declarative programming style might be the way forward. Its guaranteed side-effect free setting and explicit model of dependencies eases the detection of independent parts of a program and their parallel computation. However, even in this setting it remains the responsibility of the programmer to write their programs in a way that actually exhibits enough parallelism for it to scale to a large number of processors.

We have performed extensive research in this area, leading to the programming language Single Assignment C, or SAC for short. Despite its syntactical proximity to the imperative language C, several carefully chosen restrictions of the core of SAC inhibit any form of side-effect and, thus, render it purely functional.[1] Inherently side-effecting operations such as I/O operations are safely

[1] For a discussion of the fusion between imperative appearance and declarative nature of SAC see (Grelck and Scholz 2006).

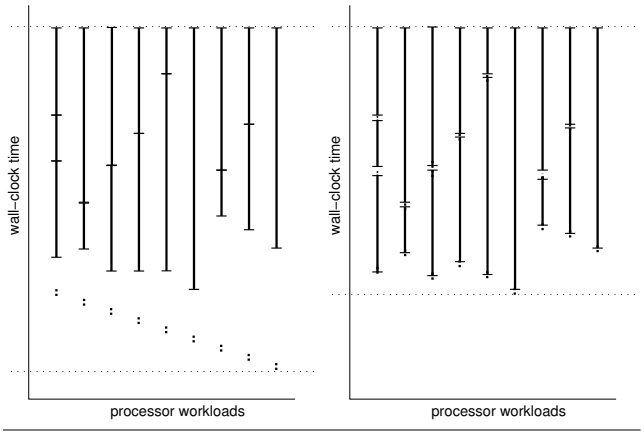

Figure 1. Utilisation of processing units for a map operation and a consecutive I/O operation. The left diagram shows the classical, fully deterministic sequential I/O operation whereas the right diagram depicts the non-deterministic semi-concurrent version.

added by using uniqueness typing based on the ideas of (Achten and Plasmeijer 1995).

The most prominent features of SAC are its support of stateless n-dimensional arrays as well as its array-centric programming model which is similar to that of APL (International Standards Organization 1993) and J (Hui and Iverson 2004). Featuring arrays and data-parallel operations thereon as the main means to express algorithms, SAC encourages the programmer to write programs in a naturally parallel fashion, without the need to explicitly build concurrency in. Careful design of the language and a range of tailored optimisations (Scholz 1998; Grelck et al. 2004, 2006) enable us to compile these high-level program specification into efficient code, en-par with industry-strength FORTRAN compilers (Shafarenko et al. 2006).

By exploiting the afore mentioned properties of functional languages, we have devised a compilation scheme and runtime system that furthermore allows us to automatically derive concurrent code (Grelck 2005; Grelck and Scholz 2006) with runtimes of some benchmarks scaling nearly linear with the number of processors (Grelck 2002).

Even though an array-centric style naturally leads to more parallel code, still inherently sequential parts remain. Apart from algorithmic reasons, one major source of dependencies that enforce a sequential execution are stateful operations like I/O. As Amdahl's law states (Amdahl 1967), the overall speedup that can be achieved is determined not by the parallel parts of a program but by its sequential segments. This becomes even more true with increasing numbers of processing units. Therefore, we believe that even partly lifting the requirement of strictly sequential I/O can be hugely beneficial.

As an example consider a data-parallel map operation followed by an output of its result. An abstract representation of a typical processor utilisation for this scenario is given in Figure 1. Each vertical line depicts the utilisation of a processing unit over time. The small horizontal ticks denote the start and end of single work units; a gap means that the processor is idling and thus not performing any useful computation. We have used solid lines to depict data-parallel operations whereas dotted lines represent stateful I/O operations. The dotted horizontal lines represent the start and end of the entire computation.

On the left-hand side, Figure 1 shows the processor utilisation for the classical, fully-deterministic approach. In this setting, the I/O operation is delayed until all processing units have finished the data-parallel task. Then, using a deterministic order, *e.g.*, from left to right, each processing unit after another processes the sequential I/O task. As can be seen, all but one processing unit idle during this phase. Even worse, with increasing number of processing units, the upper, data-parallel workload is likely to scale, as long as the number of work units is large enough. However, the wall-clock time needed to perform the I/O operation will remain the same, if not increase due to the growing synchronization overhead.

Of course, stateful operations like I/O are inherently sequential in that only one processing unit at a time may modify the state to yield predictable results. However, for certain applications like updating independent pixels of the framebuffer, printing indexed data or generating debugging output, the order in which the different tasks are performed is not of importance as long as the order of each single task stays intact. Thus, introducing a certain amount of non-determinism, *i.e.*, allowing different I/O tasks to be performed out of order, may not be harmful but considered beneficial.

The right-hand side of Figure 1 gives an example. Here, instead of waiting for all processing units to finish, each processing unit starts to perform its I/O task directly after finishing a work unit if no other processor is performing an I/O operation at the same time. Otherwise the processing unit idles until those I/O operations are finished. However, due to the imbalance in execution time of the different work units, it is likely that only one processing unit at a time is involved in I/O. Using this out-of-order scheduling of I/O operations thus creates an effective interleaving of I/O operations that does not force all but one processor to idle.

As illustrated in the example this may reduce the overall runtime quite significantly, even though the I/O workload per processing unit may be relatively small in comparison to the data-parallel workloads. Furthermore, using this approach the I/O operations will scale to a certain degree with increasing number of processing units, at least while some workload imbalance remains.

In short, the contributions of this paper are

- We present a novel semantics for combining data-parallel map operations with side-effecting computations.
- We devise a corresponding language extension for the programming language SAC.
- Using Mandelbrot fractals as an example, we demonstrate the applicability of our approach and give some runtime results.

The remainder of this paper is organized as follows. Section 2 gives a brief survey of SAC, focussing on its data-parallel map operation and the used explicit model for state. To motivate our extension to SAC, we give an example in Section 3. Section 4 introduces our approach and applies it to our example. A discussion of the impact of our extension on functional properties of SAC is given in Section 5. Finally, we give some first runtime results in Section 6 and related work in Section 7 before we conclude with Section 8.

2. Single Assignment C

Single Assignment C, or SAC for short, is a strict, first-order functional language geared at high-performance numerical computation. It combines the array-centric programming style of languages like APL and J with a syntax similar to C. In the context of this paper, we will concentrate our presentation on the data-parallel map construct of sac, the `genarray` withloop, and the handling of state through uniqueness typing. A more thorough discussion of SAC is given by (Scholz 2003).

2.1 Data-Parallel Map in SAC

For the purpose of this paper we consider program expressions that take the general form

with {
    (*lower* <= *idx_vec* < *upper*) : *expr* ;
} : **genarray**(*shape*, *default*)

where *idx_vec* is an identifier, *lower*, *upper*, and *shape* denote expressions that should evaluate to vectors of identical length, and *expr* and *default* denote arbitrary expressions. Such a WITH-loop defines an array of shape *shape*, whose elements are either computed by the expression *expr* or by the default expression *default*. Which of these two values is chosen for an individual element depends on its location, i.e., it depends on its index position. If the index is within the range specified by the lower bound *lower* and the upper bound *upper*, *expr* is chosen, otherwise *default* is taken. As a simple example, consider the WITH-loop

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

It computes the vector `[0, 2, 2, 2, 0]`. Note here, that the use of vectors for the shape of the result and the bounds of the index space (also referred to as the **generator**) allows WITH-loops to denote arrays of arbitrary rank. Furthermore, the **generator expression** *expr* may refer to the index position through the **generator variable** *idx_vec*. For example, the WITH-loop

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

yields the matrix $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 4 & 0 \\ 0 & 3 & 4 & 5 & 0 \end{pmatrix}$.

Figure 2 shows those parts of a big step operational semantics for SAC that are relevant for the context of this paper. As can be seen in rules CONST and VECT, every expression in SAC evaluates to an array. We represent these as a pair of vectors $< \vec{s}, \vec{d} >$ where $\vec{s}$ denotes the **shape-vector** of the array; it encodes the length of each axis of the array. The vector $\vec{d}$ represents the **data-vector** of the array, which contains the array's elements in row-major order.

The rule WITH formalises the semantics of the genarray WITH-loop. The first three premises ensure that the expressions for lower and upper bound, as well as for the shape, evaluate to vectors of equal length. Furthermore, we require the default expression to evaluate to a scalar value. Finally, we have to describe the evaluation of the body expression of the WITH-loop. We have adopted a λ-calculus style here, using the common semantics of abstraction (λ) and application ($()$). For each index within the index range defined by the upper and lower bound, we require that the function $\lambda\, Id\,.\, e_b$ applied to the index evaluates to a scalar value.

To assemble the result in the conclusion, we construct a data-vector by picking either the result of evaluating the body expression of the WITH-loop if the index is within the index range, or the value of the default expression otherwise. This is achieved by superscripting each value of the resulting data vector with its corresponding index within the n-dimensional shape of the result. Those indices that are within the generator set are determined by the final premise, all others equate to d as stated in the *where* clause. The shape-vector of the result is directly taken from the evaluated shape expression.

It should be mentioned here that this WITH-loop constitutes a restricted form of the WITH-loops in SAC. Exact definitions of fully-fledged WITH-loops and the formal definitions of transformations on them can be found elsewhere (Scholz 2003, 1998; Grelck et al. 2004).

```
external
World, Display initDisplay( World world, int[2] shape);

external
World destroyDisplay( World world, Display disp);

external
Display drawPixel( Display disp, int x, int y, int val);
```

Figure 3. Excerpt from the SAC binding for the SDL library.

2.2 Stateful Operations in SAC

We use a form of uniqueness typing as first developed in the context of the functional language CLEAN (Achten and Plasmeijer 1995) to model state in SAC. Each program is parametrized over a global environment, called `TheWorld`. This environment needs to be explicitly passed to every function that requires updating the global state of the program. To model this destructive update in a side-effect free setting, a function, at least conceptually, computes a new environment from the old one. Uniqueness typing ensures that each copy of the environment is read at most once, thereby effectively sequentializing stateful computations. To be able to interchange the order of operations affecting unrelated parts of the environment, *e.g.*, writing to different files, we allow the programmer to split off sub-environments from the global environment.

As an example, consider the SAC binding for the **SDL** library (Pazera 2002) given in Figure 3[2]. The function `initDisplay` splits off a sub-environment, in this case a graphical display, from the global environment of type `World`. The second argument gives the size of the screen to be created. As its result, the function `initDisplay` returns a modified global environment and a new sub-environment of type `Display`.

The function `destroyDisplay` implements the dual operation. Given the global environment of type `World` and a sub-environment of type `Display`, it consumes the sub-environment and returns an updated global environment, thereby conceptually merging the sub-environment back into the global environment.

Finally, as an example for a stateful operation, the function `drawPixel` outputs a pixel to a given display and returns the updated display. Apart from the display to output to, the function `drawPixel` expects the x- and y-cooridinate of the pixel to update, as well as the new colour.

Passing the environment explicitly can be cumbersome, in particular as it may require refactoring many function signatures when adding state manipulations deep down the call graph. To alleviate this, we have developed means to implicitly pass environments and automatically adapt function signatures. The interested reader is referred to (Grelck and Scholz 1995) for further details.

3. Mandelbrot, an Example

In the following we discuss a data-parallel implementation of the escape-time approximation algorithm for the Mandelbrot set combined with a visualisation of its results to further motivate the need of side-effecting operations within data-parallel operations. The Mandelbrot set consists of all elements z of the complex plane for which the complex polynomial $P_i = P_{i-1}^2 + z$ under iteration starting with $P_0 = z$ does not escape to infinity. The sequence of polynomials $(P)_i$ is known to escape, if for any i the absolute value $|P_i|$ is greater than 2.

[2] We have simplified the source code for presentation in this paper. The full version is available as part of the SAC standard library.

$$\text{CONST} : \frac{}{n \rightarrow < [], [n] >}$$

$$\text{VECT} : \frac{\forall i \in \{1, \ldots, n\} : e_i \rightarrow < [\, s_1, \ldots, s_m], [\, d_1^i, \ldots, d_p^i] >}{[\, e_1, \ldots, e_n] \rightarrow < [\, \mathtt{n},\ s_1, \ldots, s_m], [\ d_1^1, \ldots, d_p^1, \ldots,\ d_1^n, \ldots, d_p^n] >}$$

$$\text{WITH} : \frac{\begin{array}{c} e_l \rightarrow < [\, n], [\, l_1, \ldots, l_n] >\ e_u \rightarrow < [\, n], [\, u_1, \ldots, u_n] >\ e_{shp} \rightarrow < [\, n], [\, shp_1, \ldots, shp_n] >\ e_{def} \rightarrow < [], [\, d] > \\ \forall i_1 \in \{l_1, ..., u_1 - 1\} \ldots \forall i_n \in \{l_n, ..., u_n - 1\} : (\lambda\, Id.e_b\ [\ i_1,\ \ldots,\ i_n]) \rightarrow < [], d^{[i_1,\ldots,i_n]} > \\ \mathtt{with\ \{\ (}\ e_l\ \mathtt{<=}\ Id\ \mathtt{<}\ e_u\mathtt{)\ :}\ e_b\mathtt{;\ \}\ :\ genarray(}\ e_{shp}\mathtt{,}\ e_{def}\mathtt{)} \\ \rightarrow < [\, shp_1, \ldots, shp_n], [\ d^{[0,\ldots,0]}, \ldots,\ d^{[shp_1-1,\ldots,shp_n-1]}] > \end{array}}{}$$

$$\text{where } d^{[x_1,\ldots,x_n]} = d \text{ iff } \exists j \in \{1, ..., n\} : x_j \in \{0, ..., l_j - 1\} \cup \{u_j, ..., shp_j - 1\}$$

Figure 2. An operational semantics for the WITH-loop in SAC.

The escape-time approximation of the Mandelbrot set approximates set membership by computing a predetermined number of iterations. If for any iteration the absolute value is greater than 2, the corresponding complex number is not member of the set. All other complex numbers are assumed to be in the set.

Using this approach, set membership can be decided independently for each element of the complex plane, making computing the Mandelbrot set an ideal example of a data-parallel algorithm. Furthermore, the number of iterations needed to decide set membership greatly varies in between different elements of the complex plane, yielding a workload imbalance between independent computations, similar to the scenario shown in Figure 1

An implementation of the escape-time algorithm in SAC is given in Figure 4. The function `calcmandel` computes for a complex number `z` the number of iterations needed to decide whether `z` lies outside of the Mandelbrot set. The maximum number of iterations performed is given by the second argument `depth`. We have used a `while` loop to express the iterative evaluation of the complex polynomial. Despite their rather imperative appearance, while-loops in SAC are merely syntactic sugar for tail-end recursive functions.

The Mandelbrot set is commonly visualized as a 2D image, associating an interval of complex numbers with a matrix of pixels. The color values are picked depending on the number of iterations needed to decide on set membership. In the SAC implementation given in Figure 4 this is achieved by function `mandelbrot`. Its result is an array whose outer and inner axes denote the y- and x-axis of an image, respectively. An array element `A[[y,x]]` gives the number of iterations computed for the complex number $x + yi$. To be able to select different cuts of the complex plane, the function `mandelbrot` is parametrized by an offset on the x and y axis. The parameters `xres` and `yres` can be used to compute the Mandelbrot set in different resolutions. Finally, to prevent distortion of the final image, we compute the upper bound of the area to compute depending on a maximum value for the x axis and the given resolution.

Our implementation of the algorithm uses a `genarray` WITH-loop to map the escape check to an interval of complex numbers. As described in Section 2, the semantics of the WITH-loop, especially its guaranteed side-effect free setting, allow for all elements of the array to be computed independently. However, although very desirable in this case, when it comes to visualising the computed Mandelbrot set, the lack of side-effects proves to be harmful.

Consider displaying the computed color values on a screen. By nature, the screen is a mutable object, and, thus, its manipulation must be considered a side-effect. Using the drawing primitive introduced in Section 2, we can visualize the array as a 2D image by iterating over all its elements. However, as this operation is stateful, we cannot use SAC's data-parallel map construct, but must re-

```
int calcmandel(complex z, int depth)
{
  i=0; c=z;
  ic=imag(c);
  rc=real(c);

  while(((ic*ic + rc*rc) <= 4.0d) && (i <= depth)) {
    c=c*c+z; i++;
    ic=imag(c); rc=real(c);
  }

  return(i);
}

int[.,.] mandelbrot( double xmin, double xmax,
                     double ymin, int depth,
                     int xres, int yres)
{
  step_size = (xmax - xmin) / tod( xres);
  ymax = step_size * tod(yres) + ymin;

  res = with {
          ( [0,0] <= [y,x] < [yres, xres] ) {
            z = toc( tod(x) * step_size + xmin,
                     ymax - tod(y) * step_size );
            val = calcmandel(z, depth);
          } : val;
        } : genarray( [yres,xres], 0)

  return( res);
}

Display drawArray( Display disp, int[.,.] array)
{
  for (y = 0; y < shape( array)[[0]]; y++) {
    for (x = 0; x < shape( array)[[1]]; x++) {
      disp = drawPixel( disp, x, y, array[[y,x]]);
    }
  }

  return( disp);
}
```

Figure 4. Implementation of the escape-time algorithm to approximate the Mandelbrot set in SAC. To output the results, an inherently sequential nesting of for-loops is used.

sort to a nesting of inherently sequential for-loops. The function `drawArray` in Figure 4 shows a possible implementation in SAC.

Using the function `drawArray` to visualize the results of a concurrent execution of the data-parallel computation in function `mandelbrot` effectively results in a processor utilisation as shown on the left-hand side of Figure 1. The data-parallel computation of the array of colour values leads to a processor utilisation as sketched out in the upper half of the figure. In particular, we expect the Mandelbrot set computation to show strongly varying computation times for the different work units, *i.e.*, the time required to compute the depth for different array elements. The lower half of the picture depicts an abstract view on the workload distribution generated by the for loops. As they iterate over the entire array, a synchronisation barrier is required to ensure the array has been fully computed before it is printed to the screen. Therefore, although processing units might end up idle, interleaving the data-parallel with the sequential part is not possible.

4. Modelling Side-Effects in With-Loops

To achieve a workload distribution as shown in the right-hand side of Figure 1, we need to be able to perform stateful operations as part of a otherwise data-parallel loop.

As discussed in Section 2, we model side-effects in SAC by explicitly passing an environment and thereby introducing dependencies that enforce a certain order of evaluation. The compelling feature of WITH-loops on the other hand, especially when it comes to concurrent program execution, is the guaranteed independence between loop iterations. In particular, this requires WITH-loop bodies to be side-effect free. Thus, to combine the two, we inevitably have to give up concurrency or sequential order, or a bit of both.

Our key idea is to introduce a limited form of dependencies between loop iterations that sequentializes the evaluation of WITH-loops sufficiently enough to allow meaningful state manipulation but at the same time retains as much concurrency as possible. As our example shows, requiring a strict ordering of side-effecting operations can be unnecessary: For drawing the Mandelbrot fractal to the screen, it is not important in what order the pixels are drawn, as long as each is output exactly once in the end. Instead, it often suffices to perform side-effecting operations in any order, as long as each of them is performed in isolation. Again, looking at our example, the underlying **SDL** binding does not support concurrent drawing to the screen. Thus single updates still need to be sequentialized.

Summing up, we propose to allow side-effecting operations within WITH-loop bodies. However, instead of performing these in sequential order, we introduce a non-deterministic evaluation order: The side-effects of WITH-loop iterations can happen in any order, but the sequence within each individual iteration remains guaranteed. Conceptually, such a side-effecting WITH-loop, like regular side-effecting functions, needs to be parameterized over an explicit environment. We do so by adding an additional operator, `propagate`, and a further result to the WITH-loop. A WITH-loop in this extended syntax looks as follows:

with {
    (*lower* <= *idx_vec* < *upper*) : *expr* ;
} : (**genarray**(*shape*, *default*), **propagate**(*obj*))

The argument *obj* to the `propagate` operator gives the initial environment to evaluate the WITH-loop in. Intuitively, this environment is then passed to a single iteration of the loop, resulting in a modified environment, which can then be propagated to some other iteration. Finally, after all iterations have been computed, the resulting environment is returned by the WITH-loop. However, note that the `propagate` operator is non-deterministic in that it does not guarantee the order in which different loop iterations are performed.

```
Display, int[.,.] mandelbrot( double xmin, double xmax,
                              double ymin, int depth,
                              int xres, int yres)
{
  step_size = (xmax - xmin) / tod( xres);
  ymax = step_size * tod(yres) + ymin;

  res, disp = with {
                ( [0,0] <= [y,x] < [yres, xres] ) {
                  z = toc( tod(x) * step_size + xmin,
                           ymax - tod(y) * step_size );
                  val = calcmandel(z, depth);
                  disp = drawPixel( disp, x, y, val);
                } : (val, disp);
              } : ( genarray( [yres,xres], 0),
                    propagate( disp));

  return( disp, res);
}
```

Figure 6. Implementation of the escape-time algorithm for computing the Mandelbrot set in SAC using `propagate` for I/O operations.

This, at first glance, looks like a rather huge addition to the existing WITH-loop in SAC. In particular, extending the WITH-loop from a single `genarray` operator to multiple operators, *i.e.*, one `genarray` operator and an additional `propagate` operator, involves a major change to the semantics of SAC as presented in this paper so far. However, multiple operators have already been introduced to SAC in the context of WITH-loop-fusion (Grelck et al. 2006). What is indeed new is `propagate` as an operator.

A formal description of the extended semantics is given in Figure 5. In comparison to the original semantics presented in Figure 2, rule WITH contains an additional premise for the argument of the `propagate` operator Id_{obj}. We require that the identifier can be evaluated to an initial environment. To model the non-deterministic evaluation order, we introduce a bijective function ρ that maps elements of the iteration space I to indices into the sequence of intermediate environments $o_1, \ldots, o_\tau$ where τ denotes the overall number of iterations. To tie it all up, we then require that the function $\lambda\, Id_{iv} \,.\, \lambda\, Id_{obj} \,.\, e_b$ applied to an index vector and the previous intermediate state can be evaluated to a scalar value and the intermediate state corresponding to that index vector. Finally, the entire WITH-loop can then be evaluated as before but with the last intermediate environment as an additional result.

Using this new operator, we can now express the I/O part of our example as part of the data-parallel computation in the WITH-loop. Figure 6 shows the modified `mandelbrot` function. All that is needed to achieve this is to move the call to `drawPixel` from the for-loop nesting within function `drawArray` as shown in Figure 4 directly into the body of the WITH-loop that computes the array of colour values. This allows us to reuse the iteration space of the WITH-loop which renders the for-loop nesting redundant. As a consequence of moving the call to `drawPixel`, each pixel is now printed to the screen as soon as its value becomes available.

It should be noted here that in case the computed array is not referenced anywhere else in the program we could actually decide not to return the computed array which would enable the compiler to avoid its allocation entirely.

5. Non-Determinism and Referential Transparency

Introducing non-determinism in the order of side-effects as proposed in the previous section allows us to express the desired additional concurrency. However, the question remains what impact this has on functional properties of our language. In particular, one might wonder whether referential transparency still holds. For our

$$\text{WITH} \quad : \quad \frac{\begin{array}{c} e_l \rightarrow <[\,n],[\,l_1,\ldots,l_n]> \; e_u \rightarrow <[\,n],[\,u_1,\ldots,u_n]> \; e_{shp} \rightarrow <[\,n],[\,shp_1,\ldots,shp_n]> \\ e_{def} \rightarrow <[],[\,d]> \; Id_{obj} \rightarrow <[],[\,o_0]> \\ \exists \rho : I \leftrightarrow \mathbb{N}_\tau \; \exists o_1,\ldots,o_\tau \; \forall (i_0,\ldots,i_n) \in I \; : \; ((\lambda\, Id_{iv} \,.\, \lambda\, Id_{obj} \,.\, e_b \; [i_1,\; \ldots,\; i_n]) \; o_{\rho((i_1,\ldots,i_n))-1}) \\ \rightarrow (<[],d^{[i_1,\ldots,i_n]}>, <[],o_{\rho((i_1,\ldots,i_n))}>) \end{array}}{\begin{array}{c} \texttt{with \{ (} \; e_l \; \texttt{<=} \; Id_{iv} \; \texttt{<} \; e_u \texttt{) : } e_b \texttt{; \} : (genarray(} \; e_{shp}\texttt{,} \; e_{def}\texttt{), propagate(} \; Id_{obj}\texttt{))} \\ \rightarrow (<[\,shp_1,\ldots,shp_n],[\; d^{[0,\ldots,0]},\ldots,\; d^{[shp_1-1,\ldots,shp_n-1]}]>, <[],[o_\tau]>) \end{array}}$$

where $I = \{l_1,\ldots,u_1\} \times \ldots \times \{l_n,\ldots,u_n\}$, $\tau = \prod_{i=1}^{n}(u_i - l_i)$, and
$d^{[x_1,\ldots,x_n]} = d$ iff $\exists j \in \{1,...,n\} : x_j \in \{0,...,l_j - 1\} \cup \{u_j,...,shp_j - 1\}$

Figure 5. Extended operational semantics for WITH-loop with `propagate`.

```
external
RandomGen initRandomGen( int seed);

external
int, RandomGen nextNumber( RandomGen generator);
```

Figure 7. Signatures for a pseudo-random number generator in SAC.

```
int[10] trouble()
{
  gen = initRandomGen( 42);
  val, gen = with {
               ([0] <= iv < [10]) {
                 rand, gen2 = nextNumber( gen);
               } : ( rand, gen2);
             } : ( genarray( [10], 0),
                   propagate( gen));

  return( val);
}
```

Figure 8. Value non-determinism using `propagate`.

example, in which the order that side-effects are performed in has no impact on the overall result, this clearly is the case. However, in general we can no longer assume referential transparency.

As an example, consider the implementation of a pseudo-random number generator as indicated by the signatures shown in Figure 7.[3] The function `initRandomGen`, given an initial seed for the generation, creates a new pseudo-random number generator and returns the corresponding stateful object of type `RandomGen`. Note here, that in SAC the unique types are strictly separated from the non-unique ones. This implies that the type `RandomGen` can exclusively be used in single threaded fashion. For creating the actual random numbers from such an object, there is the function `nextNumber`. Similar to the `drawPixel` function presented in Figure 3, `nextNumber` consumes the old state of the generator and returns a new state as second result besides the next random number itself. Note here, that these two functions on their own are fully deterministic: Given two generators created with a common seed, performing the same sequence of applications of `nextNumber` on each generator will result in two identical sequences of numbers. Thus, for the functions given in Figure 7, referential transparency holds.

If we, however, add non-determinism by means of a WITH-loop with `propagate` operator, referential transparency is no longer given. Figure 8 presents an example. The function `trouble` computes a vector of 10 pseudo-random numbers by first creating a pseudo-random number generator `gen`, which is consecutively used in a data-parallel WITH-loop to generate the vector. As we use a static seed to initialise the generator, the set of generated numbers is the same for each application of the function `trouble`. However, the generated numbers are written into the vector in a non-deterministic order due to the use of the `propagate` operator. The actual value of different applications of the function `trouble` thus may differ, which invalidates referential transperancy.

Loosing referential transparency would be too high a price to pay. Yet, not everything is lost. Burton in (Burton 1988) demonstrates how to safely combine non-determinism with referential transparency in a functional language. His solution is to transform non-deterministic programs into deterministic ones by modelling non-determinism as data. He parametrizes all occurrences of non-deterministic operators by an explicit oracle, referred to as *decision* in his paper. These oracles are then passed to the program as additional arguments. To minimize the number of additional arguments that need to be passed along the call graph without unnecessarily sequentializing evaluation, Burton proposes to use an infinite lazy tree of oracles, as such an infinite tree can be split whenever a program spawns multiple threads of computation without limiting the number of oracles available to each thread. To preserve referential transparency, it then suffices to ensure that two applications of a non-deterministic operator to the same arguments, in particular to the same oracle, yield the same result.

We can apply this technique to our setting by parametrizing each `propagate` operation by an explicit oracle. Furthermore, we need to pass these oracles as arguments to the program and to distribute them along the call graph to the corresponding `propagate` operations. This could be done by a straight-forward rewriting of the source code. The only difficulty that remains is to ensure that each evaluation of an expression containing a `propagate` WITH-loop using the same oracle yields the same result. As the non-determinism in evaluation order is outside of our control, we cannot guarantee this for multiple evaluations of the same expression. However, we can ensure that each oracle is only used once using the existing uniqueness typing infrastructure. Instead of parametrizing a program by only the global state `TheWorld`, we could additionally add a second global state `TheOracles`. We could then allow, following Burton's proposal of an infinite lazy tree of oracles, to split `TheOracles` into sub-states representing parts of the tree. Finally, we would need to modify the `propagate` operator to additionally expect a tree of oracles which can be safely split and passed on into each iteration and where the remaining oracle subtrees would be recombined into a new tree. The only drawback of this solution is the increased amount of parameters that needs to be distributed.

Fortunately, this double parametrization is not necessary in many cases. The key insight here is that for sub-states of `TheWorld`, the splitting of the global state `TheWorld` always coincides with the splitting of the corresponding part of `TheOracles`. Thus, we can simply piggy-back the tree of oracles onto the global state `TheWorld`. Whenever we split a sub-state off `TheWorld`, we implicitly split off a sub-tree of the corresponding tree of oracles, as well. In the same way, the `propagate` operator can implicitly ex-

[3] The full implementation of a similar random-number generator can be found in the SAC standard library.

```
external
World, RandomGen initRandomGen( World world, int seed);

World, int[10] trouble(World world)
{
  world, gen = initRandomGen( world, 42);
  val, gen = with {
               ([0] <= iv < [10]) {
                 rand, gen2 = nextNumber( gen);
               } : ( rand, gen2);
             } : ( genarray( [10], 0),
                   propagate( gen));

  return( world, val);
}
```

Figure 9. Referentially transparent version of `trouble`.

tract a tree of oracles from the state it propagates. This implies that only sub-states of `TheWorld` can be legally used as argument to a `propagate` operator. However, as every state can be defined such that it is a sub-state of `TheWorld`, this imposes no major limitations. Furthermore, this restriction can easily be checked statically. Checking this restriction for our two examples immediately reveals that our Mandelbrot example from Figure 6 is fine whereas the example in Figure 8 is illegal.

Finally, as each oracle is only used once (due to the uniqueness property), we do not need to memorize its decision. In fact, apart from the conceptual idea, we do not need to materialize oracles or trees thereof in any way.

Using this approach, we can now define the function `trouble` without loosing referential transparency. We only need to redefine `RandomGen` as a sub-state of `TheWorld`. This is done in Figure 9 by adding an argument and result of type `World` to the function `initRandomGen`. This in turn forces the function `trouble` to be parametrized by `TheWorld`, as well. Now, two applications of `trouble` to the same arguments is a violation of the uniqueness property of `TheWorld` and thus illegal. Therefore, all legal programs are referentially transparent.

Summing up, to preserve referential transparency in the presence of `propagate` operations, it suffices to require that the `propagate` operator may only be used on objects that, at least ultimately, are sub-environments of the global environment `TheWorld`.

6. Concurrent Evaluation

To gain a first empirical insight into the effectiveness of our approach, we have added prototypical support for the `propagate` operator to our research compiler `sac2c`.[4] All executables for our measurements were compiled using revision 15805 of `sac2c 1.0-beta` with compiler flags `-O3 -noPHM -mt -minmtsize 16`. The flag `-O3` enables all default optimisations. However, due to a problem with the current implementation we were not able to use SAC's own heap manager, which therefore was disabled using `-noPHM`. Although this degrades runtime performance overall, it should not have an impact on the relative differences in runtime we are interested in. Finally, `-mt -minmtsize 16` enables the multi-threaded back-end and instructs the compiler to compute only arrays with more than 16 elements per axis concurrently.

We have conducted our experiments on a SunFire x4200 with two AMD Opteron 275 Series dual-core processors running at 2.2GHz, equipped with a total of 8GB memory. As operating system, we have employed the 32 bit version of Ubuntu Linux 7.04.

[4] The compiler including the **SDL** binding and the source code of our example can be downloaded from the project's homepage at `http://www.sac-home.org`

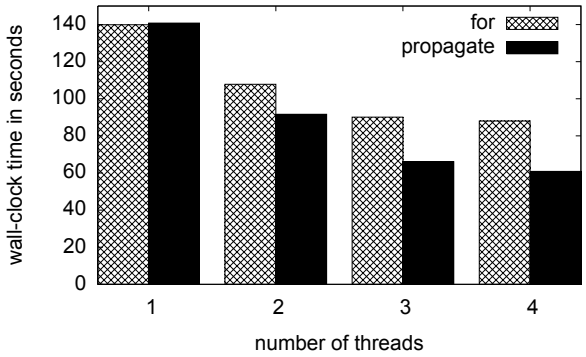

Figure 10. Wall-clock runtime of the Mandelbrot set example for different number of threads.

For our comparison, we have used two implementations of the escape-time algorithm similar to those presented in Figures 4 and 6, using for-loops and the `propagate` operator, respectively.[5] Each version computes the Mandelbrot fractal for 2 highly imbalanced sections of the complex plane with a resolution of 640×480 pixels. We have measured the wall-clock time of three iterations each for 1, 2, 3 and 4 concurrent threads, using the `\usr\bin\time` utility. The number of threads to employ was passed to the executables at runtime using the `-mt` argument. The results are shown in Figure 10.

As can be seen, the runtime of both versions does not significantly differ for the sequential case using only one thread. This suggests that the reduced loop overhead due to the fusion of the for loop that performs the I/O operation and the data-parallel WITH-loop does not significantly affect the overall runtime. However, the runtime measurements for more than one thread show that the additional concurrency introduced by the `propagate` operator indeed has a positive effect. The version using `propagate` completes significantly faster than the version using for-loops once two threads are used. For three threads, this advantage increases further. Using four processors, we do not observe a similar decrease in runtime. We attribute this to the fact that we did not have exclusive access to the server and thus other processes were running in the background. Furthermore, the X server and helper threads of the **SDL** runtime system required to perform the I/O operations might have impacted the runtime using more than three threads, as well.

To better visualize the different scaling behaviour of the two implementations, we have computed the speedup of the multi-threaded versions in comparison to the single-threaded for-loop version. As Figure 11 shows, the for-loop version does not scale well with increasing number of processors. For two threads, it is only 1.29 times faster than the sequential version. When using three threads this increases to a speedup of factor 1.55. The relatively low speedup can be explained by the for-loops that perform the I/O operations as these are inherently sequential and thus cannot make use of multiple threads.

For the version using `propagate` the speedups are significantly better (1.54 for two threads and 2.11 for three threads). We attribute this advantage to the ability to effectively interleave data-parallel computations with I/O operations, thereby hiding their sequential nature.

[5] The full source code is available as part of the `sac2c` compiler distribution.

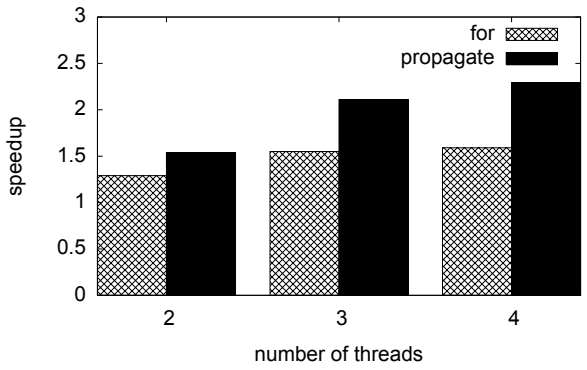

Figure 11. Speedup of the two implementations of the Mandelbrot fractal example in comparison to the single-threaded for-loop version for different number of threads.

7. Related Work

The basic idea behind our approach is by no means new. Hudak in (Hudak 1986) already proposed the introduction of non-determinism to cater for side-effecting operations within concurrently executed parts of functional programs. However, in that paper, the focus was not on I/O or other inherently sequential operations, but it was an approach to implement incremental in-place array updates concurrently. Nevertheless, the formalisation introduced there bears some similarity with the approach presented in this paper: the in-place array updates constitute our side-effecting operations. They can be used within a special mapping construct where they are considered functions that take an array and produce an updated one. The semantics definition of that construct chooses a non-deterministic order for the execution of these functions which enables concurrent executions. To our knowledge, these ideas have neither been implemented nor have they been generalised in the way proposed in this paper.

The need for concurrent I/O operations has also been identified in the distributed computing community. Gava describes in (Gava 2004) how concurrent I/O can be integrated into BSML an ML variant for bulk synchronous parallel (BSP) algorithms. The approach taken there does not attempt to model the underlying side-effect at all. Instead, special mechanisms are proposed that ensure some form of well-behaved functionality.

More recently, Terauchi and Aiken in (Terauchi and Aiken 2008) proposed a new approach for integrating side-effects into functional languages. They motivate their design by the aim "to add side effects without imposing parallelism-destroying sequentiality". By introducing so-called *witnesses*, they enable the programmer to specify explicitly how much synchronisation with respect to side-effects is desired. The main achievement is the identification of statically inferable criteria which suffice to prove confluence for their language. Although this approach can be used to reduce the sequential sections of a program, the observable result remains fully deterministic. This contrasts our approach, where the freedom to have non-deterministically chosen execution orders enables further concurrency.

An alternative semantics for the `propagate` operator would be to split the propagated state into one sub-state per iteration of the corresponding WITH-loop and to combine these sub-states again after all iterations have been computed. For our first example, this would require means to split the state representing the display into sub-states representing individual pixels and to afterwards glue these back together. In particular, such a decomposition would need non-unique arrays of unique objects. In principle, this should be possible using an extended version of uniqueness typing (Achten and Plasmeijer 1995). However, this technique is not as expressive as the approach we have chosen here. For instance, using a sequential pseudo-random number generator within a data-parallel operation as shown in Figure 8 cannot be expressed as no meaningful decomposition of the state into independent sub-states exists. Furthermore, as the execution order in this case affects the overall result, the inherently deterministic semantics of uniqueness typing would not allow for a concurrent execution. In terms of permitted concurrency, this alternative semantics thus resides in the same category as the approach by Terauchi and Aiken.

8. Conclusions

Amdahl's law states that the overall performance of a program is determined by its sequential components. In this paper, we have presented an approach to reduce this effect by combining a data-parallel map construct with side-effecting operations. Nevertheless, we preserve data dependencies between side-effecting operations within and across the data-parallel map construct. This property ensures the validity of all existing compiler optimisations as well as our mechanisms for exploiting concurrency on concurrent systems.

We have made some initial runtime experiments that demonstrate the effectiveness of our approach even on small commodity-market multi-processor systems. While runtime speedups degrade rapidly if side-effecting output is performed sequentially, our new scheme delivers clearly identifiable improvements. The remarkable observation here is that we see these improvements although the side-effecting operation under investigation needs to be executed exclusively by at most one thread at any given time. Key for this to be possible is the non-deterministic semantics of our extension. It enables the runtime system to effectively interleave computations with side-effects in such a way that the entire program is executed in a concurrent fashion.

Besides further runtime studies it remains to be observed what impact the availability of such a non-deterministic language construct has. On the one hand, its behaviour might be unexpected for the average programmer when being used extensively. On the other hand however, we may see other applications of it that enable novel ways of expressing non-deterministic problems.

References

P. Achten and R. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

Advanced Micro Devices. AMD Opteron. http://www.amd.com/us-en/Processors/ProductInformation/, November 2008.

George Almási, Călin Caşcaval, nos José G. Casta Monty Denneau, Derek Lieber, José E. Moreira, and Jr. Henry S. Warren. Dissecting cyclops: a detailed analysis of a multithreaded architecture. *SIGARCH Comput. Archit. News*, 31(1):26–38, 2003. ISSN 0163-5964.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS*, volume 30, page 483, Washington, DC, USA, 1967.

F. W. Burton. Nondeterminism with referential transparency in functional programming languages. *Comput. J.*, 31(3):243–247, 1988. ISSN 0010-4620. doi: http://dx.doi.org/10.1093/comjnl/31.3.243.

T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51(5):559–572, 2007. ISSN 0018-8646.

Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2008.57.

Frédéric Gava. Parallel I/O in bulk-synchronous parallel ML. In *International Conference on Computational Science*, pages 331–338, 2004.

S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo processor architecture. *Intel Technology Journal*, 10(2), 2006.

C. Grelck. Implementing the NAS Benchmark MG in SAC. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.

C. Grelck and S.-B. Scholz. A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

C. Grelck and S.B. Scholz. Classes and Objects as Basis for I/O in SAC. In T. Johnsson, editor, *Proceedings of the Workshop on the Implementation of Functional Languages'95*, pages 30–44. Chalmers University, 1995.

C. Grelck, S.-B. Scholz, and K. Trojahner. WITH-Loop Scalarization – Merging Nested Array Operations. In G. Michaelson and P. Trinder, editors, *Proc. of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, UK, Selected Papers*, volume 3145 of *LNCS*, pages 118–134. Springer, 2004.

C. Grelck, K. Hinkfuß, and S.-B. Scholz. With-Loop Fusion for Data Locality and Parallelism. In Frank Huch A. Butterfield, Clemens Grelck, editor, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05, Selected Papers*, volume 4015 of *LNCS*, pages 178–195. Springer, 2006.

Clemens Grelck. Shared Memory Multiprocessor Support for Functional Array Processing in SAC. *J. Funct. Program.*, 15(3):353–401, 2005. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796805005538.

Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview. White Paper, Intel Corporation, 2006.

Paul Hudak. Arrays, non-determinism, side-effects, and parallelism: A functional perspective. In *Graph Reduction*, pages 312–327, 1986.

R.K.W. Hui and K.E. Iverson. *J Introduction and Dictionary*. Jsoftware Inc., 2004.

International Standards Organization. Programming Language APL, Extended. ISO N93.03, ISO, 1993.

Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/MM.2005.35.

Ernest Pazera. *Focus On SDL*. Course Technology PTR, 1st edition, 2002.

S.-B. Scholz. With-loop-folding in SAC – Condensing Consecutive Array Operations. In C. Clack, K.Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, volume 1467 of *LNCS*, pages 72–92. Springer, 1998.

Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

A. Shafarenko, S.-B. Scholz, S. Herhut, C. Grelck, and K. Trojahner. Implementing a numerical solution for the KPI equation using Single Assignment C: lessons and experience. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05*, volume 4015 of *LNCS*, pages 160–177. Springer, 2006.

Tachio Terauchi and Alex Aiken. Witnessing side effects. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.