Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors*

Clemens Grelck[1,2], Stephan Herhut[1], Chris Jesshope[2], Carl Joslin[1], Mike Lankamp[2], Sven-Bodo Scholz[1], and Alex Shafarenko[1]

[1] University of Hertfordshire, School of Computer Science, Hatfield, United Kingdom
{c.grelck,s.a.herhut,c.a.joslin,s.scholz,a.shafarenko}@herts.ac.uk
[2] University of Amsterdam, Institute for Informatics, Amsterdam, The Netherlands
jesshope@science.uva.nl,{c.grelck,m.lankamp}@uva.nl

Abstract. We present preliminary results from compiling the high-level, functional and data-parallel programming language SaC into a novel multi-core design: Microgrids of Self-Adaptive Virtual Processors (SVPs).
The side-effect free nature of SaC in conjunction with its data-parallel foundation make it an ideal candidate for auto-parallelisation. Notwithstanding these favourable pre-conditions, scheduling and data-placement pose major challenges for effective parallelisation of irregular applications because fine-grained dynamic approaches inflict large overheads on conventional architectures. The Microgrid architecture promises a radical shift: thread creation and context switches are implemented in hardware and cause negligible overhead. Likewise, scheduling of tasks to computing resources is catered for by hardware.
This paper investigates aspects of the Microgrid architecture which may influence the overall performance of compiled data-parallel programs. In particular, we look at alternative thread creation schemes for n-dimensional, data-parallel operations and their effect on overall performance. Furthermore, we investigate the architecture's capability to cope with workload imbalances within such operations.
The paper presents a sequence of experiments on a cycle-accurate emulator of the Microgrid architecture from which we derive some guiding principles for an effective compilation of data parallel operations. Based on these principles, we present a compilation scheme for the data-parallel core of SaC.

1 Introduction

Single Assignment C (SaC) is a data-parallel, purely functional array programming language that aims at high-level, generic program specifications [**?**,**?**]. With

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

SaC, programmers can express algorithms in a succinct way, close to abstract mathematical notations. All forms of resource management in SaC, *e.g.*, memory management for arrays or thread management for parallel execution, are dealt with implicitly by compiler and runtime system. The design of SaC encourages an extensive use of data-parallel operations. They are defined by a SaC-specific language construct called a WITH-loop. Several optimisations have been developed that aim at combining several small-grain WITH-loops into fewer, more complex ones [**?**]. The benefits of these optimisations are twofold: Some redundant computations can be avoided and the need for temporary arrays is reduced. Both of these effeccts can iimprove vastly the overall memory bandwidth requirements.

Such optimised WITH-loops are good candidates for compilation into concurrently executable code [**?**]. When targeting traditional architectures, however, the high overhead inflicted by thread management and synchronisation forces us to adapt the amount of concurrency in compiled code to the available resources in a rather static manner. The Scheduling of computable tasks to executable threads mostly happens statically to avoid excessive runtime overhead; dynamic and adaptive scheduling schemes can only be applied carefully and conservatively.

The novel Microgrid architecture [**?**] promises a radical departure from such restrictions. It is based on a processor design that supports thread and resource management very efficiently in hardware through dedicated extensions to a standard processor ISA. Its implementation adheres to the SVP model (Self-Adaptive Virtual Processor) [**?**], a hardware abstraction for highly parallel, adaptive systems in general.

As a software abstraction layer for Microgrids, the language μTC [**?**] has been developed. It augments standard C with a few additional language constructs that directly map to the ISA extensions of the SVP. The most important such extension is the `create` construct for creating a family of dynamically managed, light-weight micro-threads. Immediate hardware support suggests almost negligible overhead for thread creation and synchronisation. This property makes μTC an ideal compilation target for SaC: In contrast to traditional architectures, we can exhibit the full amount of concurrency characteristic for WITH-loops to the underlying execution machinery and have the hardware schedule execution according to its own needs for the benefit of high overall performance.

In this paper we present our initial results from developing a first compilation scheme for SaC targeting μTC. In particular, we present some experimental results for alternative approaches to implementing n-dimensional data-parallel operations in the SVP model in general, and a formal compilation scheme for the core data-parallel language construct of SaC.

The paper is structured as follows: In Section 2 we give a brief overview on SaC. The next section explains the key features of the SVP followed by a section on its programming interface μTC. Section 5 investigates the runtime effects of different code generation strategies for data-parallel algorithms on arrays in

general. Based on these findings, Section 6 provides a first compilation scheme for WITH-loops in SAC. Conclusions are drawn in Section 7.

2 SaC — Single Assignment C

As the name suggests, SAC is a functional subset of C, extended by multi-dimensional arrays as first class citizens. We have adopted as much of the syntax of C as possible to ease adaptation for programmers with a background in imperative programming. Despite its C-like appearance, the semantics of SAC code is defined by context-free substitution of expressions. "Imperative" language features like assignment chains, branches, or loops are semantically explained and internally represented as nested `let`-expressions, conditional expressions, and tail-end recursive functions, respectively. Nevertheless, wherever SAC code syntactically coincides with C code, the functional semantics of SAC also coincides with the imperative semantics of C. As a consequence, programmers may keep their preferred model of thinking while the SAC compiler may exploit the functional semantics for advanced optimisation [**?**].

In contrast to other array languages, SAC provides only a very small set of built-in operations on arrays: primitives to retrieve data pertaining to the structure and contents of arrays, *e.g.*, an array's rank (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements (*array*`[`*index-vector*`]`). Aggregate array operations are specified in SAC itself using powerful array comprehensions, called WITH-loops. Their syntax is defined in Figure 1.

$$
\begin{array}{lll}
\mathit{Expr} & \Rightarrow & \ldots \\
 & \mid & \textbf{with}\ \{\ \left\lceil \mathit{Generator} : \mathit{Expr}\ ;\ \right\rceil^{+}\ \}\ :\ \mathit{Operation} \\
\mathit{Generator} & \Rightarrow & (\ \mathit{Expr}\ \texttt{<=}\ \mathit{Identifier}\ \texttt{<}\ \mathit{Expr} \left\lceil\ \mathit{Filter}\ \right\rceil\ ) \\
\mathit{Filter} & \Rightarrow & \textbf{step}\ \mathit{Expr} \left\lceil\ \textbf{width}\ \mathit{Expr}\ \right\rceil \\
\mathit{Operation} & \Rightarrow & \textbf{genarray}\ \ (\ \mathit{Expr} \left\lceil\ ,\ \mathit{Expr}\ \right\rceil\ )
\end{array}
$$

Fig. 1: Syntax of with-loop expressions

A WITH-loop is a complex expression: It starts with the keyword `with` followed by a non-empty list of *generator*–expression pairs enclosed in curly brackets. They define mappings from indices to values. Subsequently, an *operation* determines what to do with these values, *i.e.*, it defines the overall meaning of the WITH-loop.

Due to space limitations, we restrict ourselves to one operation, the `genarray` operation, which is prototypical for all major operations available in full SAC.[3] Using `genarray(` ***shp, default*** `)` as operation, a WITH-loop creates a new array of shape ***shp***.

[3] Compilation schemes for the other operations can be found via the web site of the Apple-CORE project (`www.apple-core.info`) and via the home page of SAC (`www.sac-home.org`).

Each generator defines a set of indices, more precisely index vectors, along with an index variable representing elements of this set. Two expressions, which must evaluate to integer vectors of equal length, define lower and upper bounds of a rectangular index vector range. For each element of this set of index vectors the associated expression is evaluated. It constitutes the corresponding element value of the array to be created. In the case of a `genarray`-WITH-loop, elements of the result array that are not covered by the generator are initialised by the (optional) default expression in the operation part. For example, the WITH-loop

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

yields the matrix $\begin{pmatrix} 0&0&0&0&0\\ 0&2&3&4&0\\ 0&3&4&5&0 \end{pmatrix}$. The generator in this example WITH-loop defines the set of 2-element vectors in the range between `[1,1]` and `[3,4]`. The index variable `iv` represents elements from this set, *i.e.*, 2-element vectors, in the associated expression `iv[0] + iv[1]`. Therefore, we compute each element of the result array as the sum of the two components of the index vector, whereas the remaining elements are initialised with the value of the default expression.

Multiple generator–expression pairs allow us to map different index sets to entirely different expressions. As a simple example, the WITH-loop

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

yields the matrix $\begin{pmatrix} 1&0&0&0&0\\ 1&2&3&4&0\\ 1&3&4&5&0 \end{pmatrix}$. In case the generators define index sets that are not pairwise disjoint, their (textual) sequence matters: the last mapping from the list is taken.

An optional filter may be used to further restrict generators to periodic grid-like patterns, *e.g.*,

```
with {
  ([1,1] <= iv < [3,8] step [1,3] width [1,2]) : 1;
}: genarray( [3,10], 0)
```

yields the matrix $\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0\\ 0&1&1&0&1&1&0&1&0&0\\ 0&1&1&0&1&1&0&1&0&0 \end{pmatrix}$. A missing width specification defaults to 1 in each dimension; a missing step specification likewise defaults to 1, *i.e.*, no stepping whatsoever. A width specification that equals or exceeds the step in some dimension is equivalent to a dense generator. It is noteworthy that lower and upper bounds, as well as step and width specifications are fully-fledged expressions and not restricted to constants as in our illustrative examples.

More formal descriptions of WITH-loops along with detailed introductions to SAC and its programming methodology can be found in [**?**,**?**,**?**].

3 Microgrids of Self-Adaptive Virtual Processors

The key idea of the Microgrid architecture is to create a common framework for standard processing cores which have been extended in their ISAs so that they can be used concurrently in a cooperative fashion. As a unifying concept, all such cores need to adhere to the model of a Self-Adaptive Virtual Processor (SVP).

In a Microgrid, the unit of work is a family of homogeneous and indexed threads, called *microthreads*. Families of microthreads are created at *places*. A place is an abstraction of a set of processing resources configured into a ring network somewhere on chip. A place accepts a remote SVP action to create a family of threads from another processor in a different place. This is called a *delegation*. Delegation is a mechanism for coarse-grain distribution of work in a Microgrid. The place accepts the create parameters, distributes and executes the threads that make up the family and responds to any SVP control actions for that family. In the conventional setting, this is equivalent to the remote execution of a unit of work (a job) on a set of processors. For more information about the SVP model see [**?**] and for more information about Microgrids see [**?**].

The threads in a family are automatically distributed to the cluster of processors but not necessarily all at the same time. If the number of threads exceeds the resources available there, thread creation may be delayed on those resources. The distribution is, however, completely deterministic and built into the implementation of the create instruction in the DRISC processors comprising the cluster. The distribution is primarily parameterised by

1. the number of threads in a family (N),
2. the number of processors in the place (P) and
3. the distribution strategy which is either local or global.

A local distribution creates all threads in one processor whereas a global distribution tries to involve all processors of a place evenly. However, this rule may be overriden whenever a paucity of resources occurs on a processor.

As the programming model for Microgrids is recursive, threads that have been distributed can themselves create subordinate families. For example, a 2-dimensional data-parallel operation could be implemented as a family of threads, each of which creates a subordinate family. Alternatively, we could create a single flat family of threads, one per element. With the former choice a pair of indices is generated automatically, whereas in the latter case we have a single index. For a nested create, each such subordinate family is distributed relative to its own parent's location in the ring. Note that there is no theoretical limit to the depth of recursion. However, a practical limit exists based on the resources used by existing threads on a processor.

It should also be noted that in order to make global identification of a distributed family feasible, we have implemented a sequentialisation of creates over the cluster's ring network. Only one thread at a time may acquire a token in the ring allowing it to create a family at a central registration place referred to as default place. Local creates can be executed concurrently across processors.

4 Programming Microgrids through μTC

The programming language μTC is an extension of standard C. In essence, it adds language support for thread creation and thread management. For the context of this paper, it suffices to introduce the two most important language constructs of μTC: `create` and `sync`. Their syntax is summarised in Figure 2.

Statememt ⇒ ...
| **create** (*Id* ; *Expr* ; *Expr* ; *Expr* ; *Expr* ; *Location* ; *Expr*) *Block*
| **sync** (*Id*)

Location ⇒ (**local** | **global**)

Fig. 2: Syntax of μTC extensions

A `create`-statement of the form

```
create(fid; start; limit; step; block; location; timer)
                        statement-block
```

creates a new family of threads, where each thread executes the statement block *statement-block*. The various parameters of the `create`-statement have the following meaning:

fid is an integer variable provided by the creating context; it receives the unique family identifier, which is needed for subsequent synchronisation or termination of the family.

start is an integer expression that defines the start of the index sequence for the family of threads (default value: 0).

limit is an integer expression that defines the limit of the index sequence for the family of threads (default value: unlimited).

step is an integer expression that defines the increment between index values (default value: 1).

block is an integer expression that defines the maximum number of threads allocated per processor in a single allocation round (default value: system defined).

location defines the resource on which the family will execute. A special resource called `local` forces execution of this family on the same processor as the creating environment while the `global` resource delegates the scheduling of work onto resources to the system (default value: system defined).

timer is an integer expression that restricts the number of allocation rounds to at most one per tick of a clock (default value: threads created as resources become available, subject to the constraints imposed by block).

All parameter expressions are evaluated exactly once upon execution of the `create`-statement. Any parameter except for the family identifier may be left out in favour of the default value as defined above.

Complementary to the `create`-statement, a `sync`-statement of the form

`sync(fid)`

synchronises the family of threads identified by the variable `fid`. Execution of the thread that issues the `sync`-statement is delayed until all members of the given family of threads have completed.

In addition to the `create` and `sync` statements, μTC features type qualifiers that can be used within the statement block of a thread. Of these, only the qualifier `index` is relevant in the context of this paper. A variable attributed as `index` must be of type `int`; it provides access to the thread's index within a family. A full definition of μTC including further statements and qualifiers can be found in [**?**].

5 Towards Compiling SaC to μTC

At first glance, μTC seems to be an ideal target language for a data-parallel language like SaC: Its built-in `create` instruction provides the required parallel map-instruction and the `shared` variables allow for linear dependencies where needed. However, there is still a large gap between the levels of abstraction of the two languages that needs to be bridged. In particular, compiling the truly n-dimensional data-parallel map of SaC, *i.e.*, the WITH-loop, to the strictly one-dimensional map instruction of μTC, *i.e.*, the `create` instruction, opens up an interesting design space.

On the one hand, we could compile the WITH-loop into a single, flat `create` instruction. Yet, this comes at a high price. In general, the body of a WITH-loop may reference the current index position, an n-element vector. Thus, when using a one-dimensional `create` instruction, we would need to recompute this vector from the current index of the `create` instruction for each iteration. Furthermore, for multi-generator WITH-loops, we would have to dynamically choose the appropriate generator depending on the `create` index, leading to the introduction of a sequence of conditionals into the body of the `create` instruction. Again, these conditionals would be evaluated for each iteration.

Using nested creates on the other hand would resolve all these issues. In the nested setting, we could directly derive the index vector of the WITH-loop from the indices of the `create` instructions corresponding to each dimension. Moreover, we would be able to model multi-generator WITH-loops by issuing multiple `create` instructions per dimension, thereby saving the conditionals in the body of the `create` instructions. However, the excessive use of `create` instructions for thread creation might have an adverse effect on the runtime behaviour, even though a single `create` instruction has only a negligible impact. Also, introducing nesting gives rise to the question of how the created threads should be mapped to the place. We could either use a global mapping on all levels and thus achieve a very fine grained workload distribution. Or we could only use global distribution on the outer levels and distribute all inner `create` instructions locally. This would reduce the communication cost incurred by global thread creation but would yield a more coarse grained work distribution, which might be more susceptible for workload imbalances.

So, to choose an appropriate translation of WITH-loops in SaC to `create` instructions in μTC, we need to answer the following questions:

1. What impact does the use of nested creates versus the use of one flat create have on the overall runtime?
2. How does using local distribution on inner dimensions versus using global distribution on all dimensions affect workload distribution and overall runtime?

To gain an insight, we have measured the runtimes of a data-parallel operation over 100x100-element arrays, either expressed by a one-dimensional `create` instruction or by a two-level nesting of `create` instructions. In both cases we thereby use an overall number of 10,000 threads to compute the result. However, in the nested case we need a further 100 threads to model the nesting. We have then distributed these threads over a place of P cores with P varying from 1 to 128. This, even with 128 cores, leaves at least 78 threads per core.

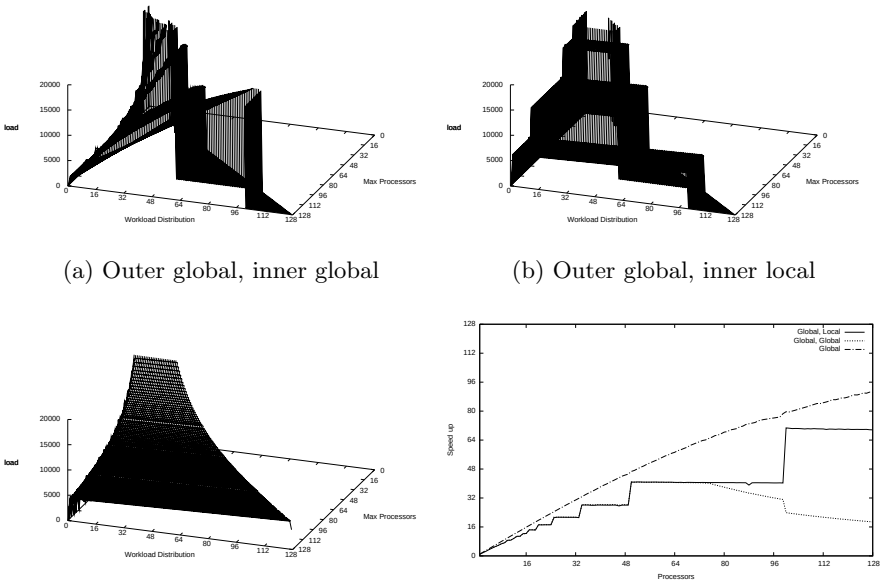
For the flat create, we have always used a global distribution. In the nested case, we have measured the runtime for two scenarios: First, we have measured a fully global distribution where all threads are created globally. We will refer to this version as *global/global* in the remainder of this section. In a second run, we have distributed the inner threads locally and have used the global distribution only for the outer ones. We have named this version *global/local*.

To investigate the impact of different distribution schemes not only on the overall runtime but on the workload balancing, as well, we have performed all experiments with two sets of workloads. The first, in the sequel referred to as *flat*, applies a fixed workload $W_{i,j} = D$ for some constant D to all threads. Thus, the work is perfectly balanced between threads. To measure the impact of unbalanced workloads, we have used a second set of workloads where the workload per thread increases with the indices on both axes, *i.e.*, for the thread with indices i and j, we use a workload of $W_{i,j} = \frac{i+j}{2*100}D$. We will refer to this workload as *diagonal*.

All tests were executed on a Microgrid emulator[**?**]. For each test case, we have measured the number of cycles required by each core. The overall runtime of each test case is then given by the number of cycles that have elapsed from the start of the computation until the last core has finished. To compare these runtimes, we have computed speedups by relating the runtime of each test case on a certain number of cores to the best runtime over all test cases on a single core.

5.1 Flat workload with $D = 12$

In our first experiment, we have used the flat workload distribution with a relatively small constant workload of $D = 12$. The results are shown in Figure 3. In all runtime graphs, *e.g.*, Figures 3a, 3b, and 3c, the y-axis gives the number of cycles that each core along the x-axis required to complete its workload. Moreover, we have plotted the results for decreasing number of maximum cores available along the z-axis.



(a) Outer global, inner global

(b) Outer global, inner local

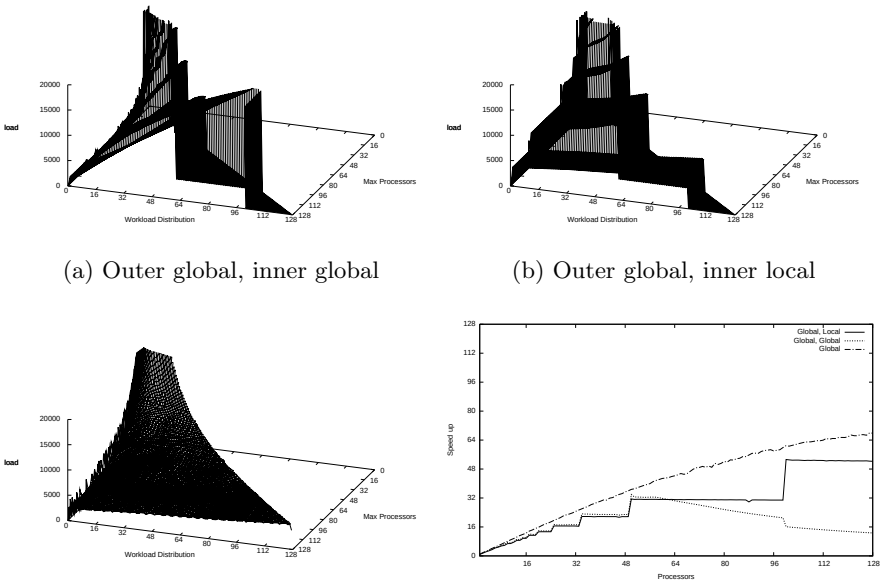
(c) Non-nested global

(d) Speed up

Fig. 3: Microgrid emulator - Flat distribution ($W_{i,j} = D$) with a workload of $D = 12$ on a 100x100 matrix.

Of all three setups, the one-dimensional `create` instruction shown in Figure 3c has the best balanced workload distribution. It uses all cores available and evenly distributes the work amongst them. For the nested `create` instruction using the global/local distribution (cf. Figure 3b), the workload distribution is equally balanced. However, not all available cores are used. Instead, only factors of 100 are utilised. We ascribe this to the scheduling algorithm used by the Microgrid emulator. Given P cores and T threads, it allocates $\lceil T/P \rceil$ threads to each core until it runs out of threads. The same effect is observable for the global/global version as shown in Figure 3a. However, the workload distribution is heavily skewed towards higher-numbered cores.

A look at the speedup graph in Figure 3d reveals the impact of the different workload distributions on the overall runtime. The one-dimensional `create` statement achieves the best runtime and scales nearly linearly with increasing number of cores. We attribute the slight decline in scaling to the increased communication cost with growing number of cores. Both nested versions show a pronounced stepping in speedups due to the imbalanced scheduling described above. However, for factors of 100 they scale nearly as nicely as the non-nested version. The remaining difference can be explained by the increased setup cost of a nesting of `create` instructions compared to the single `create` instruction in the one-dimensional case. One interesting artefact is the decline in performance of the global/global version above about 75 cores. We suspect this to be a con-



(a) Outer global, inner global

(b) Outer global, inner local

(c) Non-nested global

(d) Speed up

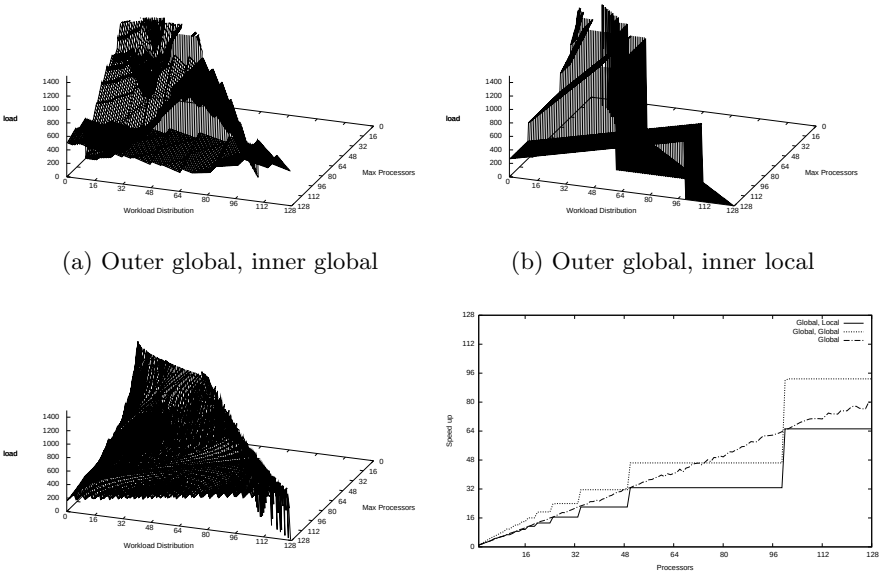
Fig. 4: Microgrid emulator - Diagonal distribution ($W_{i,j} = \frac{i+j}{2.0*100} * D$) with a workload of $D = 12$ on a 100x100 matrix.

sequence of the saturation of the ring network needed to communicate global thread creations across cores.

5.2 Diagonal workload with $D = 12$

To investigate how imbalanced workloads among different threads impact the three different scenarios, we have repeated our measurements with the diagonal workload distribution and a base workload of $D = 12$. As Figure 4c unveils, the workload when using a single one-dimensional `create` statement is still very well balanced. However, a slight skewing towards higher numbered processors can be seen. For the global/local version shown in Figure 4b, the workload distribution is still reasonably balanced, as well, exhibiting only the same slight skew towards higher numbered processors. Overall, the hardware scheduling of the Microgrid architecture is, for these two examples, able to tolerate workload imbalance between threads. In case of the global/global version, the existing imbalance of workloads (cf. Figure 3a) is further worsened by the imbalance between single threads, as shown in Figure 4a.

The speedup graph presented in Figure 4d draws a picture similar to the flat workload distribution (cf. Figure 3d). However, the overall speedup is smaller. Whereas for the flat workload distribution a speedup of nearly a factor of 90 was achieved for 128 cores, the diagonal distribution limits the speedup to a



(a) Outer global, inner global

(b) Outer global, inner local

(c) Non-nested global

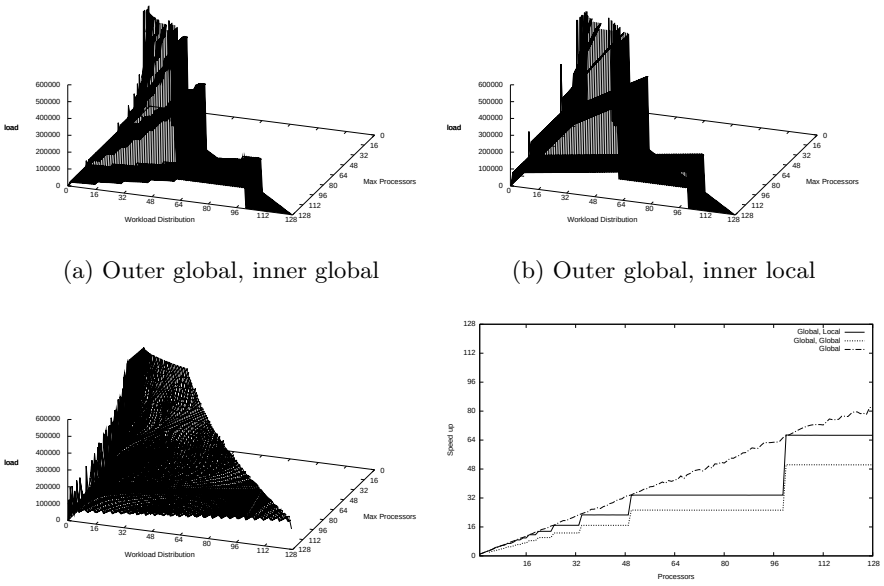
(d) Speed up

Fig. 5: SAC simulator - Diagonal distribution ($W_{i,j} = \frac{i+j}{2.0*100} * D$) with a workload of $D = 12$ on a 100x100 matrix.

factor of about 70 on the same number of cores. We assume this to be a direct consequence of the less even workload distribution.

In both speedup graphs (cf. Figures 3d and 4d), the nested versions perform slightly worse than the version using only a single `create` instruction. To probe whether this is due to the different workload distributions or due to the overhead of thread creation, we have implemented a workload distribution simulator for this particular setting in SAC. It computes only the workload distribution for all three scenarios without taking any overheads into account. The resulting data for a diagonal workload distribution with a base workload of $D = 12$ is presented in Figure 5. Figures (b) and (c) are nearly identical to their counterparts in Figure 4. However, for the global/global case as plotted in Figure 5a, the picture is rather different. Foremost, the anomaly for more than about 75 cores is gone. As the SAC simulator does not take overheads into account, this supports our theory that the anomaly is created by a saturation of the ring network.

This also manifests itself in the speedup graphs. As can be seen in Figure 5d, the global/global version outperforms the non-nested version for number of cores close to factors of 100. More strikingly, we can see that the global/global version should outperform the global/local version due to its much more favorable workload distribution. Comparing this to the measurements of the cycle-accurate simulator reveals that indeed a considerable overhead for global creates has to be paid.



(a) Outer global, inner global

(b) Outer global, inner local

(c) Non-nested global

(d) Speed up

Fig. 6: Microgrid emulator - Diagonal distribution ($W_{i,j} = \frac{i+j}{2.0*100} * D$) with a workload of $D = 1024$ on a 100x100 matrix.

5.3 Diagonal workload with $D = 1024$

Finally, to further investigate whether the anomaly for the global/global version is due to communication overheads, we have increased the base workload to $D = 1024$. This reduces the thread creation frequency, as the cores need longer to compute each workload. The results in Figure 6 show that the anomaly indeed disappears. However, the global/global version is still outperformed by the other versions. We attribute this to a general overhead for global creates, as well as the slightly more imbalanced workload distribution in the global/global case.

5.4 Lessons Learnt

Looking at the above experiments, there are two clear lessons to be learnt:

Flat is better than nested. Our experiments clearly show an advantage of the flat, non-nested version over the nested versions. Apart from overhead due to the higher number of `create` instructions and threads created, this is mainly due to the better work balancing of non-nested creates.

Only distribute globally on the outer level. If a nested use of `create` instructions is unavoidable, *e.g.*, due to the inherently n-dimensional nature of computations like in the case of some WITH-loops, only distribute the outermost create globally and perform the remaining creates on the local core. As

our experiments show, a nesting of global `create` instructions might otherwise saturate the ring network of the Microgrid. Furthermore, such a nesting led, for our examples, to less balanced workload distributions.

Using these two design principles as guideline, we will in the next section develop a compilation scheme for SaC WITH-loops to μTC.

6 Exposing the Data-Parallelism of SaC in μTC

All data-parallelism in SaC is expressed in terms of the multi-generator WITH-loops as introduced in Section 2. During the compilation process of SaC programs, these are transformed by means of several optimisations (see [**?**] for details). These transformations try to avoid the creation of arrays that hold intermediate results and, more importantly, they ensure that the resulting multi-generator WITH-loops have non-overlapping generators. This property together with the side-effect free nature of SaC guarantees that all index-vector sets in any given multi-generator WITH-loop can be traversed in arbitrary order without affecting the overal result, i.e., they can be translated directly into `create` instructions of μTC.

Putting the lessons learnt in the last section into action, we should try to generate `create` instructions with large numbers of threads rather than a nesting of smaller ones. Furthermore, we have learnt that if nested `create` instructions cannot be avoided, we should only distribute the outermost dimension globally and use the local distribution for the remaining dimensions.

Although it would be possible to use one `create` instruction for an entire multi-generator WITH-loop, such an approach, in general, would introduce too much overhead, as discussed in the previous section. Consequently, we compile each generator into a nesting of `create` instructions, where each dimension of the generator leads to one `create` instruction. Only if the `width` option is being used, we may actually create two nested `create` instructions per dimension. However, to minimise the penalty for using nested `create` instructions, we only distribute the first create on the outermost dimension globally.

Figure 7 shows a formalisation of the basic compilation scheme for multi-generator `genarray`-WITH-loops. It consists of rules of the form $\mathcal{C}[\![\mathcal{D},\ expr]\!] = expr'$, which denote context-free substitutions of SaC program fragments *expr* by μTC program fragments *expr′*. We use the argument $\mathcal{D}$ to distinguish between the two distribution modes `global` and `local`. $\mathcal{D}$ can initially be set to any value.

Rule (1) allocates the memory for the result using `a = MALLOC(` *shp*`);` and it triggers the successive compilation of the individual generators. An explicit initialization of the result array is not required as the generator sets are guaranteed to be a partition of all legal index vectors. In the applications of the compilation scheme to the individual generators, the expressions to be evaluated are transformed into assignments of the form `a[iv] = Op( iv)`, which ensures correct insertion of the computed values into the result array. Furthermore, as we transform the outermost dimension next, we set the distribution to `global`.

$$\mathcal{C}\left[\!\!\left[\mathcal{D},\ \begin{array}{l}\texttt{a = with \{}\\ \quad \texttt{(l}_1\texttt{ <= iv <= u}_1\texttt{ step s}_1\texttt{ width w}_1\texttt{)} \quad \texttt{: Op}_1\texttt{(iv);}\\ \quad \vdots \qquad\qquad\qquad\qquad \vdots\\ \quad \texttt{(l}_m\texttt{ <= iv <= u}_m\texttt{ step s}_m\texttt{ width w}_m\texttt{)} \texttt{ : Op}_m\texttt{(iv);}\\ \texttt{\} : genarray(}\ shp\texttt{);}\end{array}\right]\!\!\right] \tag{1}$$

$$= \begin{cases} \texttt{a = MALLOC(}\ shp\texttt{);} \\ \mathcal{C}\texttt{[global, (l}_1\texttt{ <= iv <= u}_1\texttt{ step s}_1\texttt{ width w}_1\texttt{) : a[iv] = Op}_1\texttt{(iv)]} \\ \vdots \\ \mathcal{C}\texttt{[global, (l}_m\texttt{ <= iv <= u}_m\texttt{ step s}_m\texttt{ width w}_m\texttt{) : a[iv] = Op}_m\texttt{(iv)]} \end{cases}$$

$$\mathcal{C}\left[\!\!\left[\mathcal{D},\ \begin{array}{l}\texttt{([}l_i\texttt{...}l_{n-1}\texttt{] <= [}iv_i\texttt{...}iv_{n-1}\texttt{] <= [}u_i\texttt{...}u_{n-1}\texttt{]}\\ \quad \texttt{step [}s_i\texttt{...}s_{n-1}\texttt{] width [}w_i\texttt{...}w_{n-1}\texttt{]) : Ass}\end{array}\right]\!\!\right] \tag{2}$$

$$= \begin{cases} \texttt{\{} \\ \quad \texttt{int fid;} \\ \quad \texttt{create(fid; }l_i\texttt{; }u_i\texttt{; }s_i\texttt{;;}\mathcal{D}\texttt{;) \{} \\ \quad\quad \texttt{index int }iv_i\texttt{;} \\ \quad\quad \texttt{int stop = MIN(}\ iv_i\texttt{+}w_i\texttt{-1, }u_i\texttt{);} \\ \quad\quad \texttt{int fid;} \\ \quad\quad \texttt{create(fid; }iv_i\texttt{, stop; 1;;local;) \{} \\ \quad\quad\quad \texttt{index int }iv_i\texttt{;} \\ \quad\quad\quad \mathcal{C}\left[\!\!\left[\texttt{local},\ \begin{array}{l}\texttt{([}l_{i+1}\texttt{...}l_{n-1}\texttt{] <= [}iv_{i+1}\texttt{...}iv_{n-1}\texttt{] <= [}u_{i+1}\texttt{...}u_{n-1}\texttt{]}\\ \quad\quad \texttt{step [}s_{i+1}\texttt{...}s_{n-1}\texttt{] width [}w_{i+1}\texttt{...}w_{n-1}\texttt{]) : Ass}\end{array}\right]\!\!\right] \\ \quad\quad \texttt{\}} \\ \quad\quad \texttt{sync(fid);} \\ \quad \texttt{\}} \\ \quad \texttt{sync(fid);} \\ \texttt{\}} \end{cases}$$

$$\mathcal{C}[\![\mathcal{D}, \texttt{([] <= iv <= [] step [] width []) : Ass}\ ]\!] = \texttt{Ass;} \tag{3}$$

Fig. 7: Compilation of multi generator `genarray`-WITH-loops.

The last two rules concern the compilation of generator expressions into a nesting of `create`-instructions. As shown in rule (2), for each component of the indexing vector `iv`, two nested `create`-instructions are created: An outer `create` which creates threads using the lower bound l_i, upper bound u_i and the step s_i as thread indices. Hence, the index of that create can directly be utilised as index component iv_i. The inner `create` is used for treating width components larger than `1`. Note here that the inner `create`, as well as the subsequent `sync` can safely be omitted whenever the width component under consideration is 1. The body of the inner `create` derives from recursively applying the compilation scheme to the generator with its leading index vector components being eliminated. We use the `local` distribution scheme during the recursive descend and for inner `create` operations. This ensures that only the outermost `create` instruction of a WITH-loop is globally distributed.

Rule (3) covers the creation of the innermost body. It simply replaces the empty generator by the assignment associated to it.

Since we use nested `create` operations to spawn the threads for the data-parallel computation, we have to synchronize on the results on each level. However, as all threads of a WITH-loop are independent, it suffices to synchronize on

whole families of threads. We implement this barrier synchronisation by inserting appropriate `sync` statements after each `create` operation in rule (2).

7 Conclusions

This paper presents our first results from investigating the suitability of the SVP architecture as a compilation target for high-level data-parallel languages in general and SaC in particular.

Although the architecture at first glance seems to be a perfect match for the needs of the data-parallel paradigm, our initial experiments reveal that there is still a considerable design space when it comes to actually creating code for data-parallel operations. Choices in concurrency granularity and placement policies can have significant impact on the overall runtime behaviour. When chosen appropriately, the architecture can exhibit excellent dynamic scheduling behaviour even in the presence of stark workload imbalances. However, when chosen poorly, runtime speedups can be rather limited.

The compilation scheme proposed in this paper seems to constitute a reasonable compromise between general applicability and runtime performance. Further experiments indicate that some optimisations such as "unrolling" of small `create` operations or code reorganisation to increase data locality can yield further improvements. Quantitative results to this effect, however, remain future work.