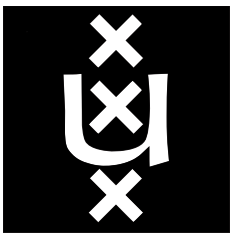
Research Internship Report

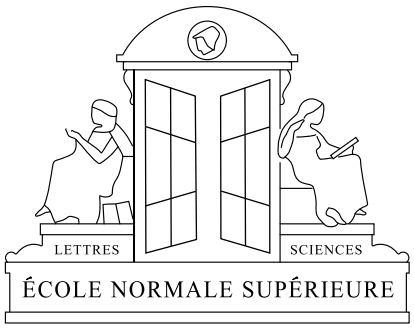
in the Computer System Architecture group of the Informatics Institute, at the University of Amsterdam, from May to August 2011.



Supervised by Clemens Grelck.

Implicit parallelization of code called from an external and already parallelized environment: from design to implementation.

Pablo Rauzy



Computer Science Department — École normale supérieure

September 15, 2011

Abstract

SAC is a purely functional array-based programming language with a strong focus on implicit parallelization. Our goal is to enable C programmers to take advantage of SAC implicit parallelization through SAC C interface. The difficulty however is that the SAC parallelization mechanism is not designed to be called from already parallelized programs. We aim at solving this problem by redesigning the SAC C interface, but with minimal changes to SAC's efficient implicit parallelization mechanism. To this end we extend the SAC compiler to introduce the concept of XT functions and virtual runtime environments. XT functions allow the implicit parallelization to take place in an already parallel context.

Contents

Thanks

I'd like to thank everyone in the Computer System Architecture research group for welcoming me as an intern in their group and for every fun and interesting conversations we had. Among them I would especially like to thanks to Clemens Grelck, my advisor, without whom this internship wouldn't have been possible.
I'd also like to thank everyone working on SAC and S-Net with whom I spend a great week in Riga (Latvia) for the SAC and S-Net 2011 Developer Camp.

1 The SAC programming language

SAC is a fifteen years old academic project whose development is spread across several institutions:

- Institute of Computer Science and Applied Mathematics of the University of Kiel, Germany ;
- Institute of Software Technology and Programming Languages of the University of Lübeck, Germany ;
- Computer Engineering Research Group of the University of Toronto, Canada ;
- Compiler Technology and Computer Architecture Research Group at the University of Hertfordshire, United Kingdom ;
- Computer Systems Architecture Group at the University of Amsterdam, Netherlands.

My internship took place in the latter.

1.1 An overview of the language

SAC (Single Assignment C) is a strict purely functional programming language whose design is focussed on the needs of numerical applications. Particular emphasis is laid on efficient support for array processing. Efficiency concerns are essentially twofold. On the one hand, efficiency in program development is to be improved by the opportunity to specify array operations on a high level of abstraction. On the other hand, efficiency in program execution, i.e. the runtime performance of programs both in time and memory consumption, is still to be achieved by sophisticated compilation schemes. Only as far as the latter succeeds, the high-level style of specifications can actually be called useful. Key to the optimization process that facilitates these runtimes is the underlying functional model which also constitutes the basis for implicit parallelization. This makes SAC ideally suited for harnessing the full potential of modern Chip Multiprocessor Architectures.

In order to facilitate the compilation to efficiently executable code, certain functional language features which are not considered essential for numerical applications, e.g. higher-order functions, polymorphism, or lazy evaluation, are not (yet) supported by SAC. These may be found in general-purpose functional languages, e.g. Haskell, Clean, Miranda, or ML.

In order to overcome the acceptance problems encountered by other functional or array based languages intended for numerical / array intensive applications,

e.g. Sisal, Nesl, Nial, APL, J, or K, particular regard is paid to ease the transition from a C / Fortran like programming environment to SAC.

In more detail, the basic language design goals of SAC are

- to provide a purely functional language with a syntax very similar to that of C in order to ease, for a large community of programmers, the transition from an imperative to a functional programming style ;
- to support multi-dimensional arrays as first class objects ;
- to allow the specification of shape- and dimension-invariant array operations ;
- to provide high-level array operations that liberate programming from tedious and error-prone specifications of starts, stops and strides for array traversals thereby improving code reusability and programming productivity, in general ;
- to incorporate a module system that allows for separate compilation, separate name spaces, and abstract data types, and, additionally, provides an interface to foreign languages in order to enable reuse of existing code ;
- to provide means for a smooth integration of states and state modifications into the functional paradigm based on uniqueness types ;
- to use the module system, the foreign language interface, and the integration of states in order to create a standard library which provides a functionality similar to that of the standard C libraries, e.g. powerful I/O facilities or mathematical functions ;
- to facilitate the compilation to host machine code which can be efficiently executed both in terms of time and space demand ;
- to *facilitate the compilation for non-sequential program execution in multiprocessor environments*.

Arrays. In SAC arrays are first class citizens of the language. All data are arrays, in particular, scalars are a special case of arrays. Arrays are represented by a data vector containing all its elements and by a shape vector which provides structural information. Its length specifies the dimensionality (or rank) of the array, and its elements define the array's extent in each dimension (see figure 1).

The type system of SAC is extended by suitable types for expressions which evaluate to arrays and for specifying prototypes of functions which take arrays as arguments or produce arrays as results. For each base types, including user defined types, an entire hierarchy of array types allows for shape specifications

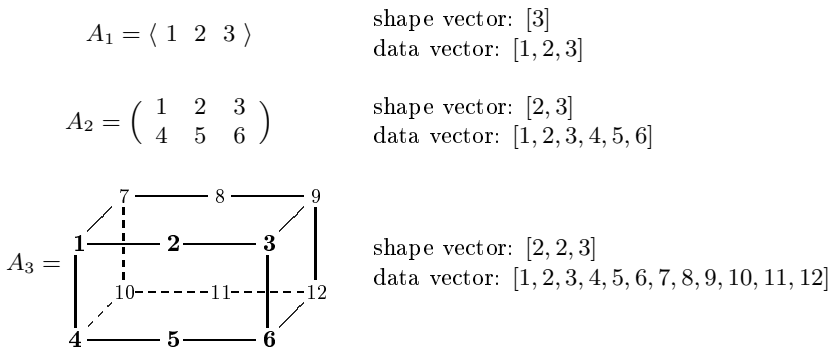


Figure 1: Example of array representation.

of varying precision. See figure 2 for an example with the `int` base type. In addition to that functions can return multiple values.

Built-in operations. SAC provides a small set of built-in functions on arrays, mainly to extract information pertaining to their structure or contents:

- `dim(array)` returns the dimensionality (or rank) of `array` ;
- `shape(array)` returns the shape vector of `array` ;
- `sel(index_vector, array)` returns the array element of `array` selected by the vector `index_vector` ;
- `reshape(new_shape_vector, array)` creates a new array whose data vector is the same as that of `array`, but whose shape is specified by `new_shape_vector`, which must be a legal shape vector with respect to the number of elements of `array` ;
- `modarray(array, index_vector, value)` generates a new array which is identical to `array` except for the element identified by `index_vector`, which is set to `value`.

Two notational abbreviations allow to write applications of the built-in functions `sel` and `modarray` in a way which is familiar from other programming languages. The expression `sel(iv, A)` is equivalent to `A[iv]`, and the assignment `A = modarray(A, iv, val);` may alternatively be written as `A[iv] = val;`.

With-loops. Array operations beyond the primitive functions may be specified by means of `with`-loops. They represent a versatile SAC-specific language construct for the definition of aggregate array operations. `with`-loops are similar

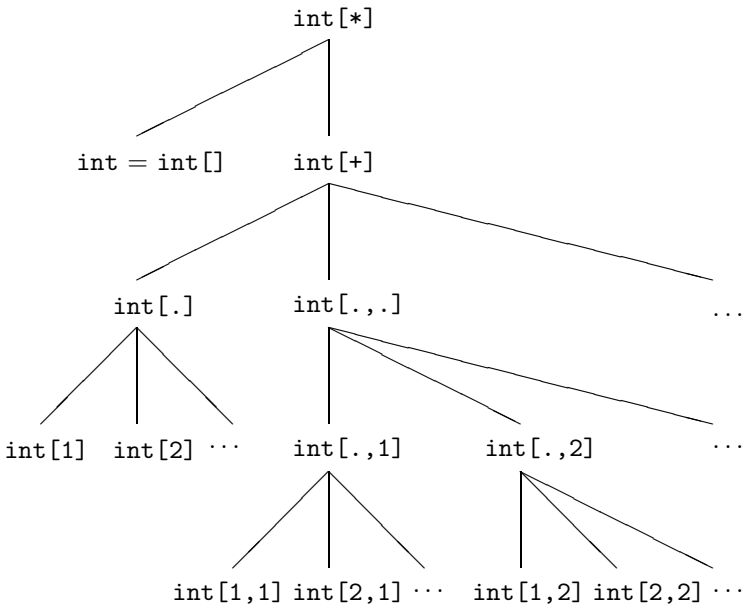


Figure 2: Array type hierarchy for the `int` base type.

to array comprehensions in other functional languages, e.g. in Clean or Haskell, and to the `for`-loops of Sisal. They either define the shape of an array to be created along with a specification of how to initialize its elements ("`genarray`" and "`modarray`" `with`-loops), or a fold operation along with a specification of how to compute the set of fold operands ("`fold`" `with`-loop). However, in contrast to similar constructs in other languages, they allow for the specification of dimension-invariant array operations, i.e., not only extents of argument or result arrays may vary in some dimensions, but even their dimensionality itself. The `with`-loop is the core operation in the SAC programming language.

For a more precise description of the kernel of the SAC programming languages along with its syntax please refer to appendix A (page 17).

1.2 SAC compilation

The SAC language is compiled to C, and the C code is then compiled to machine code using an external C compiler. The SAC compiling tools are themselves written in C. The SAC compiling process is organised in *phases* and *subphases*.

Most of the subphases correspond to a *traversal* of the abstract syntax tree (AST). The traversal system can be seen as a reimplementation of ML's `match`

... `with` construct on algebraic datatypes. For each traversal one specifies a function for each type of AST nodes (or the default function which does nothing except "traversing" is used). The result of the traversal is a modified AST that can be used by the next traversal.

The subphases composing the front-end of the compiler are reading and parsing SAC code and generating the AST using standard C tools such as Flex and Bison.

The subphases composing the back-end consists mainly of *intermediate code macros*, or ICM. ICM are a mix of C functions called by the compiler and macros written by those C functions which are responsible for printing the final C code of the compilation. This mechanism allows for a better abstraction of the code printing.

The code is actually compiled to a library, `libsac2c` and then tools using this library are created using the compiler building system which allows to specify which phases and subphases should be used and in which order. The main SAC compiler, `sac2c`, and other tools such as `sac4c` are built like this. `sac4c` creates C wrappers for SAC modules, and is of course a key part of the SAC C interface, which we're going to discuss at length.

2 Implicit parallelization

Definition. Implicit parallelism is a characteristic of a programming language that allows a compiler or interpreter to automatically exploit the parallelism inherent to the computations expressed by some of the language's constructs. A pure implicitly parallel language does not need special directives, operators or functions to enable parallel execution. SAC is a pure implicitly parallel language.

In SAC, the implicit parallelization happens with the first `with`-loop for which it's worth it. Whether it is worth is an arbitrary decision made by the compiler

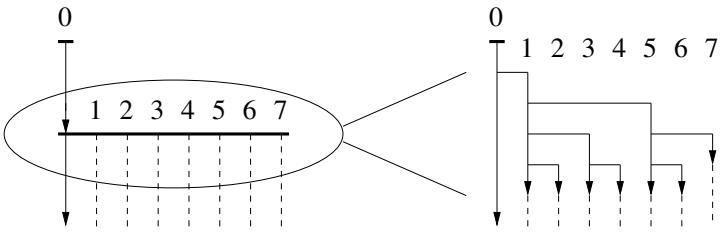


Figure 3: Organization of thread creation phase.

writer, and is decided at compile time when possible and at run time when ranks and/or dimensions are not known at compile time.

Execution model. Spawning threads takes time so instead of creating and joining them for each parallelization, SAC use a barrier mechanism. SAC runtime creates *worker threads* at the very beginning of the program execution. The worker threads creation follow a tree-structure as presented in figure 3 to be as fast as possible (see [2]). All of them immediately hit a start-barrier. The master threads execute the program sequentially and, when it encounters a `with`-loop worth parallelizing, the start-barrier is lifted. When a worker thread finishes its job it hits a stop-barrier. When the master threads finishes its part of the parallelized job it waits at its stop-barrier for the other threads to finish before proceeding to to stop-barrier synchronization. It then continues the execution of the following sequential code. See figure 4 representing this execution model graphically.

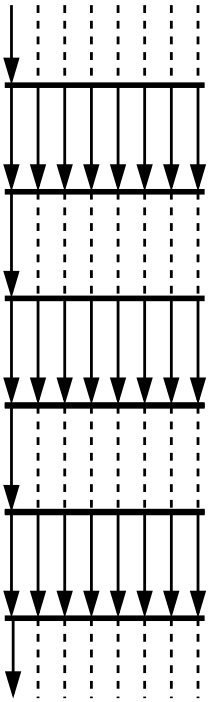


Figure 4: Execution model.

Parallelization. A closer look at what happens at parallelization time is interesting. The SAC compiler, `sac2c`, can be used to generate sequential code, or multi threaded code on different backends (pthread, CUDA graphic cards, ...) if the `-mt` argument is given. In the latter case, the compiler actually generates two versions of each function: an ST (single threaded) version and an MT version (multi threaded). SPMD (single program, multiple data) functions which wraps `with`-loop to permit further optimization by creating regions of parallel executions that stretch over several `with`-loops (see [2]).

ST and MT functions. The ST versions of functions are executed in single threaded contexts. Unlike with SEQ (sequential) functions, the code may go parallel when it encounters a `with`-loop. Once the parallelization took place, the MT versions of functions are run and the code can't go parallel again — until the parallel work is over and the code is run sequentially using the ST versions of functions. In addition to that, the ST and MT codes are aware that it runs in a single/multi threaded context. With respect to that awareness, ST and MT versions of function differ for concerns such as memory management (SAC has its own private heap manager).

SPMD **functions.** Regions of code which are parallelizable are wrapped into SPMD functions. SPMD functions are utility functions and thus their arguments concern the parallelization itself: what code (i.e. which `with`-loops) is parallelized, which variables will be read, which variables will be written to, etc. There need to be a way to transmit in the actual data the parallel computation needs and to transmit out the data it creates. This transmission is done using SPMD frames. At the end of the parallelization, a synchronisation happens and this is done using an SPMD barrier (which serves as the stop-barrier from the execution model).

SPMD **frames.** Each function has a specific SPMD frame, which corresponds to its arguments and its return values. In each SAC module, SPMD frames are implemented as a statically allocated `union` of `structs`. Each `struct` of this `union` corresponds to a function and is generated at compile time.

SPMD **barriers.** There need to be one SPMD barrier per thread, and for efficiency concerns, those are statically allocated for a predefined maximum number of threads (which can be chosen at compile time) in each SAC module. The SPMD barrier synchronization might have to collect data in the case of a `fold with`-loop. It follows a tree-structured scheme for efficiency concerns (see [2] and figure 5) and the algorithm relies on the fact that the n worker threads are numbered from 1 to n with the master thread being the number 0.

Scheduling. SAC implicit parallelization mechanism can use static as well as dynamic scheduling to distribute tasks among worker threads, thought dynamic scheduling is not fully implemented yet. A key factor for successful application of dynamic scheduling techniques is the choice of a suitable granularity, i.e. the

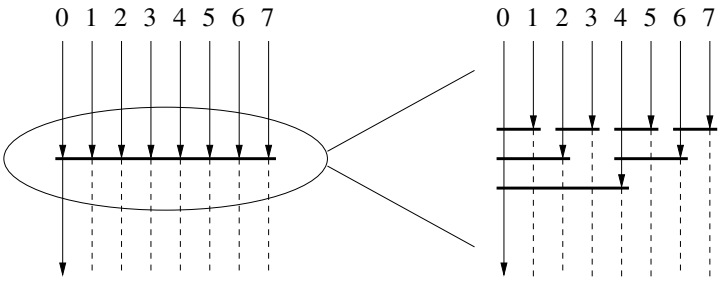


Figure 5: Organization of tree-structured stop barriers.

size of the tasks or groups of iterations as basis for the demand-driven allocation to worker threads. While a large number of small tasks results in good load balancing, the organizational overhead becomes prohibitive. Low granularity keeps the overhead within bounds, but load balancing may be insufficient. SAC scheduling schemes address this dilemma by adapting task granularity during execution. They start with rather large tasks for low overhead in the beginning and systematically reduce task sizes for good load balancing towards approaching the final synchronization barrier.

3 Redesigning the C interface

3.1 Using the right tool for the right job

SAC has been designed for high-performance computing and to take advantage of multi-core / multi-processor environments through implicit parallelization. However it is not a general purpose programming language: it is not suited for GUI development for instance. That's why we want to take advantage of SAC for the computation-intensive part of a program while using other languages (general purpose or domain specific as well) for the rest of the program code. C is the most suitable language to interface with since virtually every other programming languages can be interfaced with it.

However, SAC parallelization mechanism makes two big assumptions which may not be fullfilled when calling SAC code from an external environment. First, that we only have one parallel computation at the same time. Second, that SAC can use all the available resources (one thread per CPU core).

These assumptions are reasonable in a context where a program is entirely written in SAC. These assumptions also enable a lot of optimizations which are very welcome in a language designed for high-performance computing like SAC. However, in a context where SAC code is called from an external program may violate these assumptions. If the calling code is parallelized it already uses some of the available resources. Moreover, it could launch at the same time two or more parallel SAC computations which might in turn go parallel.

There was an existing SAC C interface which only permitted to call sequential SAC code. Adding support for parallel SAC code would theoretically not be difficult if the calling C code is guaranteed to be sequential. Real world programs require to be multi-threaded be it only to be able to respond to user input while performing another task in background. This is typically the case of virtually every program whicshah have a graphical user interface. Thus, we need to be

able to call parallelizable SAC code from an already parallelized C environment. This implies that multiple SAC computations can be launched simultaneously. Redesigning the C interface then requires to answer three questions.

First, "When and how to setup the SAC runtime environment?". This question is not specific to our requirements but its answer might be.

Second, "What to change in the the existing runtime environment?". The SAC runtime environment was not designed to account for our constraints.

Third, "How to deal with SAC code called simultaneously?". This question is specific to our setting where two concurrent threads could both call parallelizable SAC code.

3.2 Ideas and propositions

The first idea was to wrap every call from C to SAC with code responsible for setting up the SAC runtime environment with a numbers of worker threads specified via an additional argument. With this approach, the SAC parallelization mechanism could be left unchanged. Concurrent call to SAC from C would co-exist in two different environments without problems. However, efficiency concerns are important in SAC. The overhead implied by the threads creation at each call from C to SAC is too big so this approach was discarded.

The next idea was to have an external SAC daemon. The daemon would act as a server and calls from C to SAC would be requests to the server. With this approach, the runtime environment would be set up once and for all at the beginning of the program execution (or even before if the SAC daemon is permanent). The parallelization mechanism would have to be changed to be able to use only the required number of worker threads for each request. This is when the idea of having *virtual runtime environment* came in, to avoid making changes to the parallelization mechanism by rather running it as is on top of a virtualization layer. Also, the main threads of the server would have to always be listening and thus couldn't be used in parallel computations. However this approach was also discarded for performance issues: communication between the main program and the SAC daemon is too slow when input data of the SAC computation are large.

The final idea came from an evolution of this second idea. The solution to the slow communication problem is to have the SAC daemon inside the main program. Then there is then no need for a permanently running "SAC server" thread because it can be replaced with global runtime data tracking the usage of the worker threads. What was the daemon would then be a set of functions

using these runtime data to play the role of a "resource" manager. Managing the resources (i.e. the worker threads) could be done separately from the calls to SAC in order to be able to reuse ressources.

In summary, the idea is to have:

- the runtime environment (worker threads, etc.) started at the beginning of the program execution (or at least, before any call to SAC) ;
- runtime data representing the state of threads (barrier flags, etc.) ;
- a way for the C programmer to request and release resources ;
- a virtualization layer which permit to use the SAC parallelization mechanism on a subset of the worker threads with respect to a given resource.

3.3 SAC runtime initialization

The SAC runtime environment is initialized when the C program calls the newly introduced `SAC_InitRuntimeSystem` function from the `libsac` C library, which is part of SAC C interface.

This function is responsible for reading the desired number of worker threads in the `SAC_PARALLEL` envrionment variable and creating them. The worker threads creation scheme hasn't been changed and the same code is used as in pure SAC programs. Once the threads are created things change a bit. As seen in section 2, worker threads in pure SAC programs all hit a start-barrier as soon as they are created. Here the same happen but there is a different barrier for each threads so they can be controlled independently.

The function then allocate and initialize the global runtime data. The runtime data contains the total number of worker threads, the number of available (i.e. not currently owned by a resource) threads, and for each threads: a pointer to the resource owning it (which is `NULL` if the threads is available), a barrier flag, and the virtual identifier of the thread in the resource which owns it if the thread it not available.

3.4 SAC resources and virtual runtime environment

When calling SAC from C, the programmer must provide a SAC resource which will be used to create a virtual runtime environment in which the SAC code will be executed.

When the C programmer requests for a resource with n threads (supposing there are more than n available worker threads), each of the n threads is attributed

a virtual identifier from 1 to n. The thread 0 is always the calling thread (so there is actually $n+1$ threads working when the parallelization takes place). In the case were less than n available worker threads, `NULL` is returned and the C programmer has to deal with the error (by asking for less worker threads or by waiting for availability of more worker threads).

From the C programmer's perspective, a SAC resource consists of a subset of the worker threads that he can use to launch SAC computations. Internally, SAC resources contain an array which maps its threads virtual identifiers to the actual identifiers of worker threads. That's basically what a resource is.

In addition to that, for implementation purposes, SAC resources are also used by the parallelization mechanism as a placeholder for the SPMD frame, the SPMD barriers, and a pointer to the SPMD function. Those are specific to the function which is being parallelized. Thus, these SPMD stuff are set only at the parallelization time, for each parallelization, and reset once it's done.

The parallelization then happens without a problem thanks to the virtual thread identifiers and the resource-specific frame and barriers. The stop-barrier synchronisation mechanism works without changes except that it uses the virtual thread identifiers. The virtualization layer is responsible for mapping these to the actual threads using the SAC resources informations.

3.5 XT functions

Using the existing ST version of functions when calling SAC from C was not possible. First, because the parallelization mechanism has to be modified to take SAC resources into account and create the virtual runtime environments. Second, because SAC implements its own private heap manager for memory sharing among threads and it needs to take into account the fact the the program might already be parallel. The XT version of functions was introduced for those reasons. XT functions are the functions called from the C interface.

These XT versions of functions are created at the same time as ST and MT versions when compiling a SAC module for multi-threading with `sac2c`. Like ST versions of functions, XT functions AST is traversed to update the calling tree to be calling XT functions all the time until parallelization.

At the ICM level, SPMD functions called from XT functions also had to be modified to take SAC resources into account. The main change is the addition of two parameters: a pointer to a SAC resource, and the virtual thread identifier. This modification propagates to all ICM regarding SPMD frames, the synchronisation barrier.

The SAC module system does not keep track of all the functions variants because it emits its informations before their creation. For this reason, we have to partially rebuild XT versions of functions in sac4c to have what could be called their "prototype" in the AST, in order to be able to generate the C wrapper which will call the XT functions.

3.6 The new C interface

Using the new C interface requires to first initialize the SAC runtime. This is done by calling:

```
SAC_InitRuntimeSystem(argv, argc);
```

The argv and argc arguments are the one the main function receives. They are provided for compatibility with the previous way of telling the SAC runtime how many worker threads we want, which was the -mt command line flag. The number of worker threads should now be precised in the SAC_PARALLEL environment variable.

At the end of the program, or at least when the SAC C interface is not used anymore, clean programmers call:

```
SAC_FreeRuntimeSystem();
```

This call will release internal memory used by libsac and clean the SAC runtime system.

In between, the C programmer can declare pointer to SACresources and SACarg. Requesting N worker threads in SAC resource is done like this:

```
SACresources* rsc = SAC_RequestResources(N);
assert(rsc != NULL);
```

If not enough threads are available, the call will return NULL, otherwise it returns a pointer to a SACresources which can be used to call SAC functions by placing it as first argument of the wrapper function the C interface provides.

The other arguments to SAC functions must be pointers to SACargs. SACargs are representations of SAC arrays, with a shape (ranks + size of each dimension). The SACARGconvertFrom... functions are responsible for converting C arrays or scalars to SACarg. The function takes the C data as first argument, then the rank of the SAC array we want to obtain, and then the size for each of its dimensions. Here is an example of usage:

```
int *vect = functionThatReturnsAPointerToAnIntArrayOfSize100();
```

```
SACarg *theVector = SACARGconvertFromIntPointer(vect, 1, 100);
```

The C programmer can then call the SAC function. For our example, let's assume we have compiled a SAC module `Sum` which exports function `sum` which receives an array as argument and sum all its elements. As said before, the first argument must be a SAC resource, then pointers to pointers to `SACarg` for each return values, and then pointers to `SACarg` for each argument of the SAC function:

```
SACarg *theResult;
Sum__sum1(rsc, &theResult, theVector);
```

The `SACARGconvertTo...` functions are responsible for converting `SACarg` back to native C arrays or scalars. They return what would be the data vector to the SAC array.

```
int *result = SACARGconvertToIntArray(theResult);
```

The SAC resource can be reused without a problem. To release it and make its worker threads available for further resources requests, the C programmer calls:

```
SAC_ReleaseResources(rsc);
```

A small but complete example can be found in appendix B.

3.7 SAC C interface toolchain

To be able to call SAC code from C and benefit from SAC implicit parallelism, the SAC module has to be compiled with the `sac2c` compiler with the `-mt` command line argument. For a module `Foo`, this compilation will generate two library files `libFooMod.so` and `libFooTree.so`. The first one is the module itself and the second one is used by SAC module system to store information about what is in the `Foo` module.

Then, in order to generate the C wrapper functions, "`sac4c -xt Foo`" have to be called. This call will use informations about the `Foo` module from `libFooTree.so` to generate `libcwrapper.so` and a header file `cwrapper.h` which have to be included in the C program.

The C compiler can then be called to compile the C program and link it with all the libraries generated by SAC tools.

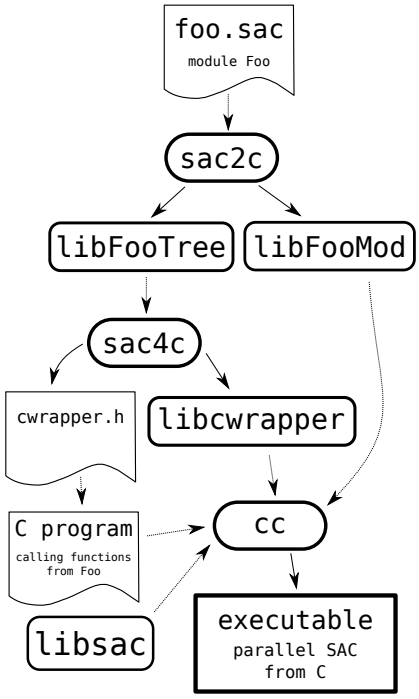


Figure 6: SAC C interface toolchain.

4 Conclusion

Our problem was calling an implicit parallel language from a multithreaded external environment, with the constraint of having to keep the existing parallelization mechanism for its efficiency, despite the fact that it has been designed assuming the whole program was coded with the same language and thus that it can use all the resources available. This work is experimental and the approach we took to solve our problem is not guaranteed to be the best one. We haven't found any other project which tries to deal with such a problem and which work under similar constraints.

While we have a successful proof of concept, things are far from perfect. For instance SAC worker threads do busy waiting while they are not in use. This was a good choice when it was made: worker threads start very fast when their start-barrier is lifted and SAC was assuming to be the only program running with one thread per core. With our new C interface SAC can't assume it's the only program running anymore. Moreover, nowadays CPU can reduce their energy consumption when they are not used so doing busy-waiting is not a good choice even when SAC is the only program running with one thread per core.

The techniques developped for the work presented here could be reused to answer other questions that are asked by SAC development: for instance how to have multiple level of parallelization?

A The SAC language kernel

The language kernel of SAC is a functional subset of C. In this context, the term "functional" refers to a rather straightforward mapping of language constructs to an applied lambda-calculus. The meaning of programs is given by this mapping and by context-free substitutions, as defined by the applied lambda-calculus. Still, this meaning is supposed to exactly coincide with that defined by the original host language's semantics, which is based on implicit control flow.

These requirements rule out global variables, pointers, and, hence, all kinds of data structures which rely on pointers. The control flow instructions `break`, `continue`, and `goto` must be dropped as well, but for the remaining language constructs found in C, transformation rules into an applied lambda-calculus can actually be defined more or less straightforwardly.

The syntax of programs is outlined here:

```
<program>    ::= [ <type-def> ]* [ <fun-def> ]* <main> [ <fun-def> ]*
<type-def>   ::= 'typdef' <type> <id>
<fun-def>    ::= <ret-types> <id> '(' [ <parameters> ] ')' <fun-block>
<ret-types>  ::= <type> [ ',' <type> ]*
<parameters> ::= <type> <id> [ ',' <type> <id> ]*
<main>       ::= 'int main ()' <fun-block>
<type>       ::= <id> | 'int' | 'float' | 'double' | 'char' | 'bool'
```

They are sequences of top-level type and function definitions with a specific function main serving as the designated starting point for program execution. Function headers are almost identical to ANSI-C, the sole difference being that supports functions with multiple return values, similar to ID or SISAL . Hence, function definitions may start with sequences of return types, separated by commas, instead of a single return type only. The set of basic types is restricted to the four scalar data types known from C, which represent integer numbers, single and double precision floating point numbers and characters. In contrast to C, SAC explicitly distinguishes between integer and boolean values. Hence, a dedicated data type bool is added to the set of basic types. Furthermore, SAC supports function overloading , i.e., functions may share the same name as long as they differ with respect to the types of their formal parameters. The ordering

of function definitions is irrelevant, i.e., functions may be applied before they are defined, thus making obsolete forward declarations of function prototypes.

```
<fun-block>    ::= '{' [ <var-dec> ]* [ <assign> ]* <return> '}'
<var-dev>      ::= <type> <id> ';'
<assign>       ::= <let> ';'
                 | 'if (' <expr> ')' <assign-block> [ 'else' <assign-block> ]
                 | 'while (' <expr> ')' <assign-block>
                 | 'do' <assign-block> 'while (' <expr> ');'
                 | 'for (' <let> ';' <expr> ';' <let> ')' <assign-block>
<let>          ::= <id> [ ',' <id> ]* <assign-op> <expr>
<assign-op>    ::= '=' | '+=' | '-=' | '*=' | '/=' | '%='
<assign-block> ::= <assign> | '{' [ <assign> ]* '}'
<return>       ::= 'return (' <expr> [ ',' <expr> ]* ');'
```

As shown in the table, a function body consists of variable declarations, assignments, and a trailing `return`-statement. Local variable declarations are optional in SAC; they may be omitted. An assignment is either a simple let-statement or a compound assignment. In the presence of functions with multiple return values, let-statements may in one conceptual step bind multiple variables to values. A compound assignment is either a conditional or one of the three loop constructs known from C. Branches of conditionals as well as loop bodies may either be single assignments or sequences of multiple assignments delimited by curly brackets. However, such local blocks may not contain additional variable declarations, i.e., there are no local scopes in other than function bodies. The `return`-statement may contain a sequence of expressions, separated by commas, rather than a single one only, to support functions with multiple return values.

The mapping of SAC function definitions to an applied lambda-calculus is rather straightforward. A sequence of let-statements is mapped to nestings of let-blocks with the trailing `return`-statement defining the goal expression of the entire term. An `if`-clause is interpreted as a functional conditional with the continuation code being copied into both branches. Last but not least, loops are considered syntactic sugar for equivalent tail-end recursive functions.

The following table gives an overview of SAC expressions, which may be constants, variable identifiers, or applications of built-in operators, primitive functions, or defined functions. SAC supports the usual arithmetic, relational, and boolean operators just as C. The usual precedence rules apply, but expressions may be grouped by round brackets.

```
<expr>   ::= <const> | <id>
```

```
          | '(' <expr> ')'
          | <mon-op> <expr>
          | <expr> <bin-op> <expr>
          | <prf> '(' [ <expr> [ ',' <expr> ]* ] ')'
          | <id> '(' [ <expr> [ ',' <expr> ]* ] ')'
<mon-op> ::= '!'
<bin-op> ::= '+' | '-' | '/' | '%'
          | '==' | '!=' | '<' | '<=' | '>' | '>='
          | '&&' | '||'
<prf>    ::= 'toi' | 'tof' | 'tod' | 'min' | 'max'
```

The primitive functions `toi`, `tof`, and `tod` allow for explicit conversion between the three numerical data types `int`, `float`, and `double`; there is no implicit conversion as in C. Additionally, SAC provides built-in minimum and maximum functions.

This relatively small and simple language kernel allows for the specification of clear and concise purely functional programs. Yet, their functional semantics, as defined by mapping them into an applied lambda-calculus, exactly coincides with their usual imperative interpretation.

```
<expr>       ::= ... | <with-expr>
<with-expr>  ::= 'with (' <generator> ')' [ '{' <assign>* '}' ] <operation>
<generator>  ::= <bound-expr> <relop> <id> <relop> <bound-expr> [ <filter> ]
<bound-expr> ::= '.' | <expr>
<relop>      ::= '<' | '<='
<filter>     ::= 'step' <expr> [ 'width' <expr> ]
<operation>  ::= 'genarray (' <expr> ',' <expr> ')'
                 | 'modarray (' <expr> ',' <expr> ',' <expr> ')'
                 | 'fold (' <id> ',' <expr> ',' <expr> ')'
                 | 'fold (' <fold-op> ',' <expr> [ ',' <expr> ] ')'
<fold-op>    ::= '+' | '*' | '&&' | '||' | 'min' | 'max'
```

The syntax of `with`-loops is specified above. A `with`-loop basically consists of two parts: a generator and an operation. The generator defines a set of index vectors along with an index variable representing elements of this set. Two expressions which must evaluate to vectors of equal length define lower and upper bounds of a range of index vectors, which may additionally be restricted by an optional filter to define grids of some stride and width.

For further details about SAC, visit its homepage[1].

B Small example using the new C interface

This appendix contains a very small but complete example usage of the new SAC C interface. We make a few assumptions:

- The `SAC_PARALLEL` environment variable is set to 16 (because we have 16 cores).
- We have a SAC module called `Foo` exposing a `bar` and a `quux` functions.
- `Foo` has been compiled with `sac4c -xt`, creating a C wrapper lib for `Foo` and the "cwrapper.h" header file.
- `SAC_InitRuntimeSystem(argc, argv);` has been called in the main function at the beginning of the program.

In a thread we can have:

```
int data[W * H], *vec;
SACresources *rcs;
SACarg *arg, *res;

/* populating the data array... */

/* we want to use only 10 cores */
rcs = SAC_RequestResources(10);

arg = SACARGconvertFromIntPointer(data, 2, W, H);

/* res = Foo::bar(arg); */
Foo__bar(rcs, &res, arg);

vec = SACARGconvertToIntArray(res);
```

And in another thread, at the same time:

```
int datum[S], *result;
SACresources *sac_resources;
SACarg *sac_datum, *sac_result;

/* populating the datum array... */
```

```
/* we can use the 6 other cores */
sac_resources = SAC_RequestResources(6);

sac_datum = SACARGconvertFromIntPointer(datum, 1, S);

/* sac_result = Foo::quux(sac_datum); */
Foo__quux(sac_resources, &sac_result, sac_datum);

result = SACARGconvertToIntArray(sac_result);
```

References

[1] SAC tutorial, `http://www.sac-home.org/`.

[2] Clemens Grelck, *Shared Memory Multiprocessor Support for Functional Array Processing in SAC*. Journal of Functional Programming 15(3), 353-401 (2005).

[3] Clemens Grelck and Sven-Bodo Scholz, *SAC – From High-Level Programming with Arrays to Efficient Parallel Execution*. Parallel Processing Letters 13(3), 401-412, (2003).

[4] Nico Marcussen-Wulff and Sven-Bodo Scholz, *On Interfacing SAC Modules with C Programs*. Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL'00), 381-386, (2000).