Configurable data structure layout for memory hierarchies

Máté Karácsony
kmate@elte.hu

Máté Tejfel
matej@elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

Developments of the last decades resulted in an increasing gap between processor and memory speeds. Therefore in high-performance computations the main bottleneck is memory access. A practical solution to this problem is to implement memory hierarchies utilizing multiple levels of memories with different capacities and access profiles. For embedded and low-power systems it is more beneficial to apply programmable scratchpad memories instead hardware controlled caches. The overall performance of scratchpad-aware software heavily depends on how the data is distributed and accessed in different memory layers. Choosing the data layout for optimal performance is a set of non-trivial design decisions. This paper introduces configurations over data structures to enable deferring these decisions, by making it easier to change the data layout of existing programs.

1 Introduction

Even the smallest computer systems have multi-layer memory hierarchies nowadays. One purpose of these hierarchies is to reduce the effects of the speed difference between processing units and storage devices [Car02]. Quick execution of data load and store operations is critical, while they could stall the execution until the requested memory operation is done. For this reason, many architectures are using hardware controlled cache memories between registers and the main memory. These caches provide quicker access to copies of data, however, their implementation usually requires sophisticated algorithms and circuitry. In these systems, the programmers have no explicit control over the contents of the caches.

In real-time and embedded systems, where the unpredictability of hardware controlled caching is not acceptable, or physically feasible, it is common to use software controlled, explicitly programmable scratchpad memories [Pan97]. These are also preferred over complicated caches to reduce energy consumption [Ban02]. Scratchpad memories are usually very fast compared to the main memory, but they are also very limited in capacity. Deciding which data should be kept in these memories is crucial for optimal performance. There were several attempts to aid optimal mapping of data to scratchpad memories either statically [Ver02] or dynamically [Ver04], the validation of these choices is usually possible only by profiling the software. Therefore further optimizations or feature extensions often lead to changes in these design decisions.

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

Scratchpad-aware systems are usually programmed using low level languages. In these languages, the distribution of data in different memory layers is reflected by the data structure definitions and the operations on those structures [Nie97]. Changes in data layout will be reflected in the source code as well, often causing significant non-trivial modifications. To make these modifications easier, we define configurations over data structures. Configurations enable fine-grained description of data layout of an application. They were first introduced as built-in language constructs in *Miller* [Nem13], a domain specific language for multi-core router programming. By reusing the key ideas behind these constructs, it is also possible to synthesize data structure definitions and accessing code through implementations consisting of C macros and C++ templates. Configurations could also be used to control and parametrize source-to-source transformations.

The organization of this paper is the following. The next subsection presents related work. Section 2 introduces and describes configurations. Different implementations of configurations are detailed in section 3. This is followed by a comparison of the implementation techniques in section 4 . Finally, section 5 concludes the paper.

1.1 Related work

Miller is a domain-specific language focused on the development of performance-critical applications for scratchpad-aware architectures. As it was presented earlier by other members of our research group in [Nem13], it supports stackless programming using execution units called *bubbles* instead of regular functions. Therefore the control flow is based on continuation passing instead of call stacks. The other specialty of this language is the built-in support for configurable data layout, that is presented in this paper. *Miller* was implemented in Haskell, first as an embedded language. Later a custom C-like syntax was developed with a standalone compiler. The compiler supports a general MIPS32 architecture, and one of its proprietary derivatives. The generated output is optimized assembly text. The applied optimizations include peephole and prefetch optimizations. The implementation of prefetch optimization is based on memory access profile descriptions.

There are data structure transformations that increase performance both on hardware-cached and scratchpad-aware architectures. In [Pra14], several such transformations are presented over the Low Level Virtual Machine (LLVM) framework. Structure peeling and splitting is intended to improve performance by the separation of "hot" and "cold" fields of structures, that is, the frequently and infrequently used parts of the data. These transformations could also need structure field reordering. Other presented techniques for improving cache locality is struct-array copy, instance interleaving and array remapping. The execution of these transformations is planned to be automatic, based on profiling data and static code analysis. However, currently no implementation is available yet for these techniques in the public LLVM code base.

Data layout transformations are very common to support vectorization. [Šin16] shows a data layout inference technique that identifies layout transformations that convert data into a format favorable for SIMD instructions. The presented inference algorithm is based on a type system for layout transformations, and enables the auto-vectorization of programs.

Scout [Krz11] also supports automatic vectorization. It is a source-to-source transformation tool based on the *Clang* compiler infrastructure. *Scout* supports several different loop transformations to achieve optimal conditions for applying SIMD instructions. It supports several widely used SIMD instruction sets like SSE or AVX, but it is also extensible.

2 Configurations

The memory hierarchy of the target architecture must be taken into account to achieve optimal performance on scratchpad-aware systems. The data layout of an application could change for several reasons, for example in order to always store the most frequently accessed data in the fastest memory. When a new system is under development, usually not all factors affecting performance are known in advance. Fine-tuning of the application requires consecutive measurements and changes in the memory mapping of data. Requirement changes could also have the same effect. A more drastic reason is a change in the underlying memory hierarchy, for instance when an existing application is ported to a different target architecture. In this case, new capacity and speed relationships must be considered for performance.

These changes could produce two different kinds of transformations in the data layout. When the space available for a given type of objects is decreased, either because the capacity of the memory is decreased, or the number of objects is increased, it could be necessary to place the less frequently used data members into a different memory layer. The original data fields will be transformed into pointers. The indirection introduced here will increase the access time of these members, but the more frequently accessed data parts are still in the faster memories. This transformation is called *data outsourcing.*

In the opposite case, when the amount of free space available in faster memories increases, indirections that are pointing to data in different layers could be eliminated. This enables the placement of less frequently used data members into faster memories, thus increases the average memory access speed. Complementary to the previous technique, this transformation is called *data inlining.*

The role of configurations is to describe the data layout of types and objects across the memory hierarchy. Therefore changes in configurations enable the description of the previous transformations.

2.1 Architecture-independent memory model

As there are many scratchpad-aware architectures with very different memory hierarchies, we decided to describe data layout configurations in an architecture-independent way over a simplified memory model. In practice, the architecture-specific information could be added to the model easily when implementing configurations with a given technique, as it will be detailed in various subsections of Section 3.

By forgetting the architecture-dependent memory information, the mapping of data layouts to concrete memory layers is lost. The only significant property is whether the value of a data member is directly available as an integral part of the structure, or only indirectly through a pointer. The latter case makes it possible to store a part of an object in a different memory than the object itself.

2.2 Description of configurations

As most of the scratchpad-aware systems are programmed in low level languages, especially in some dialect of C, we decided to derive configurations from the type constructions of the C language. Enumeration types, function pointer types and type aliases have nothing to do with data layout in general, thus the interesting ones are pointer, array and record (struct and union) constructions. In the following, let $\mathcal{T}$ be the set of all types, and $\mathcal{T}_B \subset \mathcal{T}$ is the set of configuration base types, that is, all scalar and record types. $\mathcal{T}_B$ does not contain any pointer or array types.

According to the transformations described in the beginning of the section, only data outsourcing could be applied on a given scalar element of type $\tau \in \mathcal{T}_B$. Using the C language notation $*$ for pointer types, this transformation could be described as $\tau \mapsto \tau*$ between the types. Let the source of this transformation, that is, the primitive configuration corresponding to a directly contained scalar data member, denoted by $\mathit{scalar} \in \mathcal{C}$, where $\mathcal{C}$ is the set of all configurations. Furthermore, denote the previous transformation that introduces an indirection level with a pointer type constructor as $\mathit{ptr} : \mathcal{C} \to \mathcal{C}$. These two constructors generate the following configurations:

$$\mathit{scalar}, \mathit{ptr}(\mathit{scalar}), \mathit{ptr}(\mathit{ptr}(\mathit{scalar})), \ldots, \mathit{ptr}(\cdots(\mathit{ptr}(\mathit{scalar}))\cdots)$$

Let us introduce the type generator function $\mathcal{G} : \mathcal{C} \times \mathcal{T}_B \to \mathcal{T}$, that generates a configured data type from a configuration and a base type. Using a base type τ, these previous configurations are generating the following types:

$$\begin{aligned}
\mathcal{G}(\mathit{scalar}, \tau) &= \tau \\
\mathcal{G}(\mathit{ptr}(\mathit{scalar}), \tau) &= \tau* \\
&\cdots \\
\mathcal{G}(\mathit{ptr}(\cdots(\mathit{ptr}(\mathit{scalar}))\cdots), \tau) &= \tau * \cdots *
\end{aligned}$$

Layout configuration of arrays requires the configuration of two properties. First, the skeleton of the array needs to be accessed to index a specific element, and second, the access of the selected element itself is needed to be configured. These are both possible by introducing a third configuration constructor, $\mathit{array} : \mathcal{C} \to \mathcal{C}$. The parameter of this constructor specifies how the elements are configured at each index, relative to the skeleton. The configuration of the array skeleton could be specified by applying more configuration operations around this constructor. Figure 1 shows the application of this constructor in various cases, using [ ] array type constructor on application of *array* in $\mathcal{G}$, with the notation of the C language.

All fields of record types could be configured in the similar way as independent objects, so there is no need to define a separate configuration constructor for this case. However, we can define a *compound configuration* for

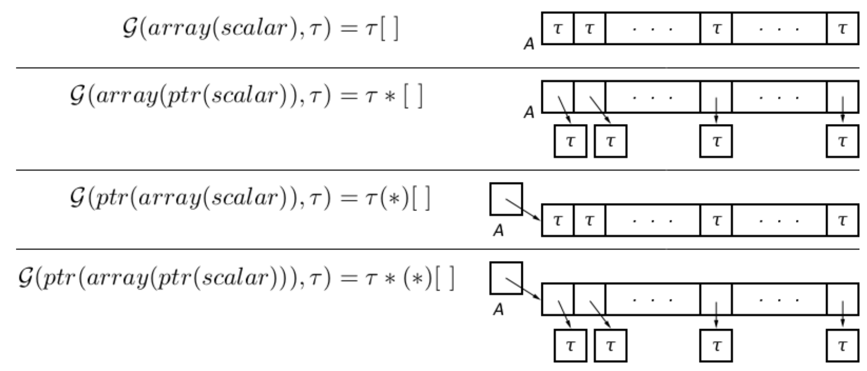
$\mathcal{G}(array(scalar), \tau) = \tau[\,]$
$\mathcal{G}(array(ptr(scalar)), \tau) = \tau * [\,]$
$\mathcal{G}(ptr(array(scalar)), \tau) = \tau(*)[\,]$
$\mathcal{G}(ptr(array(ptr(scalar))), \tau) = \tau * (*)[\,]$
A
τ

Figure 1: Examples of different configurations for array A of base type τ

a record. Let $c_{rec} = \langle\ f_1 = c_1\ ;\ \cdots\ ;\ f_n = c_n\ \rangle$ denote a compound configuration for a record *rec* with fields $f_1, \ldots, f_N$, and $c_1, \ldots, c_n \in \mathcal{C}$. As configuration operations will not be defined over compound configurations, $c_{rec} \notin \mathcal{C}$.

A configuration is *valid* for an object, when it contains as many *array* constructors applied as the dimensionality of the object requires. For scalar values, it is zero, for one dimensional arrays it is one, and so on. The innermost constructor of a valid configuration is always *scalar*. Denote the *dimension* or *array rank* of a valid configuration with $\mathcal{D} : \mathcal{C} \to \mathbb{N}$, that gives the number of *array* constructors applied in it. Important to note, that $\mathcal{T}_B$ does not contain array types. It implies that all array dimensions must be visible in configurations.

The *minimal configuration* of an object is a *valid* configuration that contains no *ptr* constructors. It is easy to see that *data inlining* could not be applied on a minimal configuration, as it would involve the removal of a *ptr* constructor. It means when an object has a minimal configuration, it is only possible to store the whole object in the same memory layer.

Two configurations are *compatible*, when they only differ in the application of *ptr* constructors. This requirement is fulfilled for configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ exactly when $\mathcal{D}(\mathcal{C}_1) \equiv \mathcal{D}(\mathcal{C}_2)$. Compatible configurations could be transformed freely into each other by adding or removing *ptr* constructors – therefore by applying data outsourcing or inlining. It is obvious that $valid(c_1, o) \wedge compatible(c_1, c_2) \implies valid(c_2, o)$, where $c_1, c_2 \in \mathcal{C}$ and o is an arbitrary configurable object.

Define the *extension function* $E_c : \mathcal{C} \to \mathcal{C}$ corresponding to a configuration $c \in \mathcal{C}$ by replacing the innermost *scalar* constructor of c with a variable. For example, let $c = ptr(array(ptr(scalar)))$. Then replacing the innermost scalar with a new variable x in c gives the extension function $E_c(x) = ptr(array(ptr(x)))$.

Consider configurations c_1 and c_2 as *structurally equivalent*, also denoted as $c_1 \equiv\ c_2$ when c_1 and c_2 was created using the same constructors in the same order. The *length* of a given configuration is $L : \mathcal{C} \to \mathbb{N}$ that gives the number of constructor functions used to create it. Of course, $L(c_1) = L(c_2)$ is implied by $c_1 \equiv\ c_2$. $L(c)$ could also be denoted as L_c.

Note that all of the configuration constructors are extensible with arbitrary information. For example, the *scalar* and *ptr* constructors could be extended with references to memory layers. This enables to inject architecture-specific information into a configuration.

2.3 Operations over configured objects

As the previous subsection presented, function $\mathcal{G}$ can be used to generate a configured data type $\tau_c = \mathcal{G}(c, \tau) \in \mathcal{T}$ where $c \in \mathcal{C}$ and $\tau \in \mathcal{T}_B$. To access data in objects defined with type τ_c, we need operations that could follow the changes of their data layout configuration.

First, an operation is needed to declare an object with type τ_c. Let $declare : \mathcal{C} \times \mathcal{T}_B \times\ Id \to\ Decl$, where Id is the set of valid identifiers, and $Decl$ denotes the set of all valid declarations of a given programming language. It is easy to see that *declare* can be built trivially when an implementation of $\mathcal{G}$ is available.

Since it is allowed to modify a configuration by adding or removing *ptr* constructors at any time, the memory management of configured objects could be changed drastically. As a consequence, explicit memory allocation and deallocation operations are needed at the beginning and at the end of the lifetime of configurable objects. For instance, take the following declaration in the C language: $declare(int, scalar, "x") \Rightarrow$`int x`. By default, `x` will have an automatic storage, that means its lifetime will be bound to the surrounding block. There is no need for explicit allocation or deallocation of `x` in this case. However, if we apply data outsourcing to the configuration, which is completely valid in this case, it gives $declare(int, ptr(scalar), "x") \Rightarrow$`int* x`. To store any value at the memory location pointed by `x`, we need to implement explicit memory management. It implies that explicit $allocate : \mathcal{C} \times\ Id \to\ Stmts$ and $deallocate : \mathcal{C} \times\ Id \to\ Stmts$ operations must be used every time to initialize and finalize storage for configured objects, where $Stmts$ is a list of statements in a given programming language. Following the previous example, $allocate(scalar, "x") \Rightarrow$`{ }` and $deallocate(scalar, "x") \Rightarrow$`{ }`, where `{ }` is an empty list of statements, but $allocate(ptr(scalar), "x") \Rightarrow$`{ x = malloc(sizeof(x)); }` and $deallocate(scalar, "x") \Rightarrow$`{ free(x); }`. These are the most complex operators, as they must be able to derive the allocation and deallocation statements for any objects with valid configurations, including multi-dimensional arrays with several layers of indirection.

Accessing the value of a configured object could also change with the configuration. It is evident that all pointers must be dereferenced to access a value. Fortunately, configurations contain all the information needed to achieve this. Let $valueOf : \mathcal{C} \times\ Expr \to\ Expr$ be an operation over a configuration and an object that generates an expression $\in\ Expr$ the set of all expressions in a given language, to access a configured object. To continue our previous example, $valueOf$ can be implemented for the C language to give $valueOf(scalar, x) \Rightarrow$`x`, and $valueOf(ptr(scalar), x) \Rightarrow$`*x`.

Similarly, when the base type of an object $\tau \in \mathcal{T}_B$ is a record type, and a field of this record object is accessed, the object must be dereferenced first. Assume we have a projection function $\rho : Expr \times\ Id \to\ Expr$ that selects a given member from an object. Let $memberOf : \mathcal{C} \times\ Expr \times\ Id \to\ Expr$, and $memberOf(c, o, m) ::= \rho(valueOf(c, o), m) \in\ Expr$, where $c \in \mathcal{C}$, $o \in\ Expr$ and $m \in\ Id$. In an example, when we have an object `x` of a record type that has a field named `y`, then the following could be implemented for the C language: $memberOf(scalar, x, "y") \Rightarrow$`x.y`, and $memberOf(ptr(scalar), x, "y") \Rightarrow$`x->y`, that is equivalent with $\rho(valueOf(ptr(scalar), x), "y") \Rightarrow$`(*x).y`, given that $\rho(o, m) \Rightarrow (o).m$. Note that when field `y` is also a configured data type, then it must be accessed with the appropriate operation, for instance using $valueOf(c_y, memberOf(c_x, x, "y"))$, where $c_x, c_y \in \mathcal{C}$ and $x \in\ Expr$ with a record base type having a field named `y`.

The last operation needed for configured objects is array indexing. An array configuration has a dual role: it describes not only how to access the elements, but it also describes the array skeleton itself. These could be done in the following steps. First, the configuration needs to be split into two parts. Let $split : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$ be a function that returns two configurations, $split(c_{arr}) = (c_{skel}, c_{item})$ when $\mathcal{D}(c_{arr}) \geq\ 1$ is satisfied. Require that $\mathcal{D}(c_{skel}) \equiv\ 0$ and $\mathcal{D}(c_{item}) \equiv \mathcal{D}(c_{arr}) - 1$. The first requirement implies that c_{skel} contains only a $scalar$ constructor and an arbitrary number of ptr constructors. Take the extension function $E_{c_{skel}} : \mathcal{C} \to \mathcal{C}$ of c_{skel} by replacing the innermost $scalar$ constructor with a variable. Then the following structural equivalence must be satisfied between the previous configurations: $c_{arr} \equiv\ E_{c_{skel}}(array(c_{item}))$. The first part of the configuration, c_{skel} is needed to dereference the array object itself before indexing. It could be done easily with the previous $valueOf$ operation. After selecting the required element using an index function $\nu : Expr \times \mathbb{N} \to\ Expr$, configuration c_{item} must be considered similarly to dereference the element data. Therefore operation $valueAt : \mathcal{C} \times\ Expr \times \mathbb{N} \to\ Expr$ could be defined as follows: $valueAt(c_{arr}, arr, idx) ::= valueOf(c_{item}, \nu(valueOf(c_{skel}, arr), idx)) \in\ Expr$, where $split(c_{arr}) = (c_{skel}, c_{item})$. Configuration c_{item} could also be referred as the *item configuration* of c_{arr}. For example, assume we have an array `x` with configuration $c_x = ptr(array(ptr(scalar)))$. Then the value of $\nu(x, 3) \Rightarrow$`x[3]` as in C language, could be accessed as $valueAt(c_x, x, 3) \equiv\ valueOf(c_{x-elem}, \nu(valueOf(c_{x-skel}, x), 3)) \Rightarrow$`*((*x)[3])` in C. Splitting c_x gives $c_{x-skel} \equiv\ ptr(scalar)$ with extension function $E_{c_{x-skel}}(c) = ptr(c)$ and also $c_{x-elem} \equiv\ ptr(scalar)$.

3 Implementation techniques

This section details techniques that we used to implement configurations for data type layout. Four solutions will be evaluated: the integration into a domain-specific language; implementation with C preprocessor macros and C++ template metaprogramming; and finally source-to-source transformations. The following subsections describe these implementations independently, while a comparison could be found in Section 4. As most of their implementation details are out of the scope of this paper, the complete source code for the C and C++ libraries could be found at `https://github.com/kmate/configurable-data-layout`.

Table 1: Summary of constructors, properties and operations defined for configurations. $\mathcal{C}$ is the set of all configurations, $\mathcal{T}$ and $\mathcal{T}_B$ are the set of types and base types, *Decl*, *Expr* and *Stmts* are declarations, expressions and statement lists of a programming language.

Sets	Description
$\mathcal{C}$	Configurations
$\mathcal{T}_B$	Base types without configuration
$\mathcal{T}$	Configured types
Constructors	Description
$scalar : \mathcal{C}$	Scalar configuration without indirection
$ptr : \mathcal{C} \to \mathcal{C}$	Introduce indirection to the parameter configuration
$array : \mathcal{C} \to \mathcal{C}$	Use parameter as an element configuration of an array object
Properties and utilities	Description
$\equiv : \mathcal{C} \times \mathcal{C} \to \mathbb{L}$	Structural equivalence
$L : \mathcal{C} \to \mathbb{N}$	Length ($L_c \equiv L(c)$)
$\mathcal{D} : \mathcal{C} \to \mathbb{N}$	Dimension or array rank ($D_c \equiv D(c)$)
$E : \mathcal{C} \to (\mathcal{C} \to \mathcal{C})$	Extension function ($E_c \equiv E(c)$)
$\mathcal{G} : \mathcal{C} \times \mathcal{T}_B \to \mathcal{T}$	Type generator function
$split : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$ where $\mathcal{D}(c_{arr}) \geq 1$	Splitting and item configuration c_{item}, where $split(c_{arr}) = (c_{skel}, c_{item})$
Operations	Description
$declare : \mathcal{C} \times \mathcal{T}_B \times Id \to Decl$	Declare an object with a configured data type
$allocate : \mathcal{C} \times Id \to Stmts$	Memory allocation for configured object
$deallocate : \mathcal{C} \times Id \to Stmts$	Memory deallocation for configured object
$valueOf : \mathcal{C} \times Expr \to Expr$	Value dereference
$memberOf : \mathcal{C} \times Expr \times Id \to Expr$	Record member dereference
$valueAt : \mathcal{C} \times Expr \times \mathbb{N} \to Expr$	Array indexing

3.1 Domain-specific Language: *Miller*

First we developed and implemented data layout configurations in a domain-specific language named *Miller*. *Miller* was designed for efficient multi-core router programming over complex multi-layer, scratchpad-aware memory hierarchies. The language has a C-like syntax and type system. It supports various fixed-width integer scalars, enumeration types, type aliases, bit sets, records, and arrays. Pointers are treated specially in the language: they are split into two separate types. Sharing of data is expressed with *references*. They are very similar to ordinary C pointers with two exceptions: it is not possible to do pointer arithmetic on references, and they cannot be used for expressing optionality. For the second use case, there is a different *optional* type in the language, that can be used in a special case construction to deal with the absence of a value. Reference types are introduced with a `ref` keyword, while option types are marked with a question mark. Arrays and records have similar syntax as in C. For instance, in the following listing `bucket` is a reference to an array of 32 optional values with type `elem`. It is able to store a single line out of 256 possible such entries of `table`.

```
struct elem
{
    int32 key;
    int32 value;
}
elem in;
elem?[32][256] table;
ref elem?[32] bucket;
```

Miller has a language construction to provide information about the memory hierarchy. The `memory` keyword introduces a new layer with the given name and size. It is also needed to provide an *access profile*, that can be used by the prefetch optimizer of the compiler. In the following example it is omitted, as this information is irrelevant for configurations. The listing below defines two memory layers named `dram` and `spad`, with 256 and

8 megabytes of total size.

```
memory dram (256MB) { /* access profile omitted */ }
memory spad (8MB)   { /* access profile omitted */ }
```

Once the memory types are given, it is possible to define locations where objects can be stored. There are two different kinds of location definitions: registers and locations in one of the memory layers.

```
register r1(4B);
register r2(4B);
// ...
location table_loc(256KB) on spad;
location in_loc(8B) on dram;
```

After memory types and locations are given, configurations can be provided for each object independently from the algorithms working with them. In fact, there are two types of configurations in *Miller*. There are *type configurations* to determine how objects of a given type should be stored in the hierarchy, and there are *object configurations* for specifying location and type configuration of individual objects.

```
typeconfig elem_conf {
    key;
    value -> dram;
};
```

`elem_conf` shows a possible configuration for record type `elem`. Field `key` has an empty configuration. It means that all data referred by `key` will be stored in the same memory as the containing object, so it represents a *scalar* configuration constructor. This is the default behavior, so this entry is optional. The `value` field has a type configuration built with operator `->`. It introduces an indirection into the configuration: the containing object will only have a pointer to the value stored in the `dram` memory. Hence the configuration of `value` is $ptr(scalar)$. Note that for algorithms working with the `value` field this indirection is invisible. This guarantees that the configuration could easily be changed independently from other parts of the code.

An object configuration consists of a location and a type configuration. In the following listing some of the irrelevant syntactical parts are omitted – the ones that are only required to unambiguously map the names in configurations to the object declarations.

```
in : in_loc;
table : table_loc elem_conf [][] ?> dram;
bucket : r1 :> spad elem_conf [] ?> dram;
```

The simplest object configuration only specifies a location for the given object. The configuration for `in` could be described with a single *scalar* constructor, along with the meta-information about the location. As `table` refers to a two-dimensional array, its configuration must be constructed using two *array* constructors. Because the elements of the array are optional, and this is implemented with pointers under the hood, it is possible to store the elements of this two-dimensional array in another layer. The `?>` operator configures an optional type to store the values in the memory given on its right hand side. The configuration of `table` can be described as $array(array(ptr(scalar)))$. The outer *array* constructors have associated meta-information about the location `table_loc`, and the *ptr* constructor refers to `dram`. Similarly to `?>`, operator `:>` can be used to configure a pointer, but under a reference type instead of an optional. It means that `bucket` will be stored as a pointer in register `r1` to an array of pointers on `spad`. The pointers in the array are referencing values stored on the `dram`. Both of the last two configurations refer to `elem_conf`, to specify how the structure fields are stored.

Operations defined for configurations in Section 2.3 are built-in into the *Miller* compiler. As configurations can also be stored separately, it is possible to have multiple different configurations for the same program. The configurations are read from source files, and connected implicitly and unambiguously with the types and objects found in the program by their qualified names. The user does not have to explicitly use different operations on objects with configured data types, as all objects are configurable in the language. Of course, there is a validation phase in the compiler, that checks the configurations of each type and object before compilation. Type configurations are allowed to be under-specified: when no configuration is provided, a default configuration takes places that stores everything in the same memory as its container object. The details of the target architecture were adjustable through memory and register configuration to a given extent, but still some information was built into the compiler, especially about how the data access code should be generated for particular layers.

Table 2: Summary of type configuration operators available in *Miller* and their corresponding configuration constructors. c_1 and c_2 are also type configuration elements themselves.

Type configuration element	Configuration constructor	Description
$\langle \textit{Empty configuration} \rangle$	$\langle \textit{None} \rangle$	Used to terminate configuration chains, or explicitly state default scalar configuration
$\langle \textit{Type configuration name} \rangle$	*scalar*	Specifies storage of data members
c_1 `:>` $\langle \textit{Memory name} \rangle$ c_2	*ptr*	Configures storage for a reference type
c_1 `?>` $\langle \textit{Memory name} \rangle$ c_2	*ptr*	Configures storage for an optional type
c_1 `->` $\langle \textit{Memory name} \rangle$ c_2	*ptr*	Introduces indirection without change in type
c_1 `[]` c_2	*array*	Configures storage for array elements

3.2 C Preprocessor Metaprogramming

Even though *Miller* implemented configurations perfectly – as it was explicitly designed for them –, it is always hard to introduce a new domain specific language instead a well-known industrial standard. This lead us to experiment with the embedding of configurations into C, that is already widely used on scratchpad-aware architectures.

The most convenient option was to implement configurations and their operations as a preprocessor macro library. Despite the high expressiveness of preprocessor macros, it is very hard to build and debug really complex libraries. To ease the development, we have chosen an implementation strategy based on modeling the macros with simple purely functional programs in Haskell [Kar14]. Finally, these definitions were translated back to the preprocessor macro language, using the Boost Preprocessor Library [Bpl].

In this system, the configurations are represented internally with a parenthesized sequence of tokens instead a chain of constructor applications. It means that configurations in the form $C_1(C_2(C_3))$, where each C_i is a constructor of configurations, is expressed as $(T_1)(T_2)(T_3)$, where each T_i is the corresponding token to constructor C_i. The exact definition of these tokens is not important, only the fact that each different constructor has a different, unique token that makes it possible to pattern match on the value during processing. Although the internal representation is not a chain of constructor applications, it is possible to define builder macros that support chained application.

```
#define SCALAR          (SCALAR)
#define PTR(tokens)     (PTR)tokens
#define ARRAY(tokens)   (ARRAY)tokens
```

As macro expansions are not recursive, tokens on the right hand side will not be re-expanded after substitution. For example, the configuration `PTR(ARRAY(PTR(SCALAR)))` will build the token sequence `(PTR)(ARRAY)(PTR)(SCALAR)`. This sequence can be effectively processed with a fold-like construction, up to a predefined maximum length (that is currently 64 using the Boost Preprocessor Library). The library defines the following macros corresponding to properties of configurations. Macro `ARRAY_RANK` calculates $\mathcal{D}_c$, the number of array dimensions for a given $c \in \mathcal{C}$. `ITEM_CONFIG` returns the item configuration of a given configuration. It is used to implement *split*. Utility `HAS_PTR_ITEM` gives a macro representation of a boolean value. When the configuration passed to `HAS_PTR_ITEM` has a `(PTR)` item it returns true, and false otherwise. This utility is useful when implementing memory allocation and deallocation operations, as it determines whether it is needed to generate dynamic memory allocation code or not.

The macro library is implemented with two different interfaces. The first version requires the configurations to be passed as an argument for every operation. The second version uses a lookup mechanism to retrieve configurations for objects by their name. This works as the following. The user has to declare their configurations as macros according to a naming scheme. Then the `GET_OBJECT_CONFIG` utility macro could retrieve the appropriate value for a given object identifier:

```
#define CONFIGURE__input PTR(SCALAR)
#define CONFIGURE__table ARRAY(PTR(ARRAY(SCALAR)))
```

The above snippet defines configurations for two objects. They can be retrieved by expanding `GET_OBJECT_CONFIG(input)` and `GET_OBJECT_CONFIG(table)`. The naming scheme is also configurable in the library. In the second version of the interface, this utility macro is called automatically by the operations, so they only take an object identifier instead of a configuration. In the following, this interface will be presented.

Each operation on objects from configured data types is implemented with a specific macro. Thus for the *declare* operation there is a corresponding macro named `DECLARE`, for *valueOf* there is `VALUE_OF` and so on. An interesting difference, that comes from the characteristics of the C language, is that the declaration of arrays need a different treatment. This is because they contain the length of each array dimension. This is implemented with a `DECLARE_ARRAY` variadic macro. Unfortunately, it is problematic to handle the zero parameter-case of variadic arguments. This is the main reason for this solution.

```c
struct input_t { uint32_t a, b; };
DECLARE(struct input_t, input);      // struct input_t * input;
DECLARE_ARRAY(char, table, 8, 32);   // char ((*( table[8]) ) [32]);
```

The comments after each macro invocation are showing their expansion results. Under normal circumstances, the result of the preprocessing phase is not directly visible to the user. It is not formatted by the compiler, and the macro library does not try to eliminate unnecessary parentheses. The configurations defined earlier are used to assemble the final declarations. Of course, it is not enough to declare these objects before using them. As the configurations could contain *ptr* constructors, explicit allocation and deallocation is needed at the beginning and at the end of their lifetime. This could be achieved as the following.

```c
ALLOCATE(input);  // do { input = malloc(sizeof(*(input))); } while(0);
ALLOCATE(table);
// do { for(int i0 = 0; i0 < sizeof(table) / sizeof((table)[0]); ++i0) {
//   (table)[i0] = malloc(sizeof(*((table)[i0])));
// } } while(0);

// ...

DEALLOCATE(table);
// do { for(int i0 = 0; i0 < sizeof(table) / sizeof((table)[0]); ++i0) {
//   free((table)[i0]);
// } } while(0);
DEALLOCATE(input);  // do { free(input); } while(0);
```

The expansions in comments clearly show the complexity of the generated code for allocation and deallocation. Bodies of `do-while` blocks are only running once, and the loop structure will be optimized out by the compiler. They are only needed to suppress the semicolon at the end of the macro invocation when multiple statements are generated.

Of course, the data stored in the configured objects should be accessed only with the appropriate operations, instead of directly operating on them. This ensures the correct access event when the configuration is changed between compilations.

```c
strcpy(VALUE_AT(table, 0), "Zero");
// strcpy(*((table))[0], "Zero");
// ...
scanf("%d␣%d", &MEMBER_OF(input, a), &MEMBER_OF(input, b));
// scanf("%d %d", &(*(input)).a, &(*(input)).b);
uint32_t key = (MEMBER_OF(input, a) + MEMBER_OF(input, b)) % 8;
// uint32_t key = ((*(input)).a + (*(input)).b) % 8;
printf("%s\n", VALUE_AT(table, key));
// printf("%s\n", *((table))[key]);
```

Note that address-of operator `&` can safely be applied on the result of `MEMBER_OF`, as it will alway result in a value selected from a structure. It is clearly visible, that the macro library does not try to simplify the syntax tree, as it generates `(*(input)).a` instead of the equivalent `input->a`. It is also worth to note that fields of `struct input_t` are not configurable. In that case, the value access of the fields must apply an additional `VALUE_OF` operator around `MEMBER_OF`.

Unfortunately, the case of configurable records fields is even more complicated. As there is no compile time reflection information behind the macro system, it is impossible to automatically determine the fields of a structure. However, when the structure declaration itself is built up with macros as a preprocessor data structure, there is a hope to share its field information. Although, the current version of this library follows a simpler implementation. The following example shows how to use named configurations with structure fields.

```
#define CONFIGURE__t0 PTR(SCALAR)
#define CONFIGURE_MEMBER__test_t__x PTR(SCALAR)
#define CONFIGURE_MEMBER__test_t__y PTR(ARRAY(SCALAR))

typedef struct
{
    int id;
    DECLARE_MEMBER(test_t, bool, x);
    DECLARE_MEMBER_ARRAY(test_t, char, y, 8);
} test_t;

DECLARE(test_t, t0);
ALLOCATE(t0);
ALLOCATE_MEMBER(test_t, VALUE_OF(t0), x);
ALLOCATE_MEMBER(test_t, VALUE_OF(t0), y);

MEMBER_OF(t0, id) = 0;
VALUE_OF_MEMBER(test_t, VALUE_OF(t0), x) = true;
strcpy(&VALUE_OF_MEMBER(test_t, VALUE_OF(t0), y), "test");
printf("t0:␣{␣%d,␣%d,␣\"%s\"␣}\n",
       MEMBER_OF(t0, id),
       VALUE_OF_MEMBER(test_t, VALUE_OF(t0), x),
       VALUE_OF_MEMBER(test_t, VALUE_OF(t0), y));

DEALLOCATE_MEMBER(test_t, VALUE_OF(t0), y);
DEALLOCATE_MEMBER(test_t, VALUE_OF(t0), x);
DEALLOCATE(t0);
```

This listing shows how the complexity of the macro system explodes when there are configured record fields. In case of named configurations, different macros needed to be used for the operations on members, because they need to use a different name-lookup algorithm. This is why all `*_MEMBER` macro receives the name of the encapsulating struct, and the name of the actual member. This information is used to assemble a macro name where the required configuration can be looked up. When the user chooses the alternative interface, where configurations are directly passed to operation macros, there is no need for defining separate `*_MEMBER` macros, but the original ones could be used everywhere. It is also important to see, that the encapsulating struct object must be resolved to its value with its own configuration before any member operations could be applied to it.

While it is possible to implement configurations in the C preprocessing system, it is not very practical. The configuration changes are propagated easily, once the code is appropriately prepared, but unfortunately that requires a huge effort, and is highly error-prone.

Architecture specialization could be done by extending the configuration builder macros with additional parameters. For instance, `PTR` can be extended to hold additional value to identify the target memory of a pointer. In the implementation of operations, this information can be extracted and used for example to generate different allocation or dereference code for different pointers.

3.3 C++ Template Metaprogramming

While the previous subsection showed that it is possible to implement configurations in the preprocessing phase, there is still space for improvement. Some of the scratchpad-aware systems also have C++ compilers available. It is a known result that template metaprograms are Turing-complete in absence of instantiation limits [Vel03]. Their integration in the C++ language, along with other language features like operator overloading, provide a good basis for implementing configurations. The library presented in this section is implemented using the Boost MPL Library [Abr04].

As template metaprogramming also has its roots in functional programming, like preprocessor metaprogramming, the representation of configurations follows a similar strategy: configurations are stored as type-level lists of configuration tokens, where tokens themselves are types. Most of the operations are processing these type level lists with folds and pattern matching on the tokens.

```
typedef struct {} T_SCALAR;
typedef struct {} T_PTR;
typedef struct {} T_ARRAY;

struct SCALAR : mpl::vector<impl::T_SCALAR> {};

template<typename Tokens>
struct PTR : mpl::push_front<typename Tokens::type, impl::T_PTR> {};

template<typename Tokens>
struct ARRAY : mpl::push_front<typename Tokens::type, impl::T_ARRAY> {};
```

Builder constructions for these list are provided too, thus the user can easily define named configurations as type aliases:

```
typedef PTR<SCALAR> input_config;
typedef ARRAY<PTR<ARRAY<SCALAR>>> table_config;
```

These configurations are essentially the same as we used in the previous subsection. An interesting aspect of this technique is that the configuration chain is built up from nested template applications. Declarations of configurable objects is done in the following way:

```
configured<struct input_t, input_config, true> input;
configured_array<char, table_config, false, 8, 32> table;
```

Declarations for scalar and array objects are different, like it was with preprocessor macros. However, the reason for this is different here: it is to improve type safety. Array-like configured objects support indexing, while others do not. Both of the types are parametrized with a base type, a configuration, and a boolean value, that indicates whether automatic allocation and deallocation should be done for these objects. Configured arrays also need to provide sizes for each dimensions, this makes `configured_array` a variadic template. While `input` will be automatically allocated and deallocated, this has to be done explicitly for `table` (hence the `false` parameter value above):

```
table.allocate();
// ...
table.deallocate();
```

As the above listing shows, allocation and deallocation operations become instance methods of configured objects in this system. This allows the implicit sharing of the underlying configuration. Other configuration operations are also implemented with instance methods, and some of them with operator overloading:

```
strcpy(table[0], "Zero");
// ...
scanf("%d␣%d", &input->a, &input->b);
uint32_t key = (input->a + input->b) % 8;
printf("%s\n", table[key]());
```

The *valueAt* operation is simply implemented with the `operator[]` of `configured_array` objects. Also, operation *memberOf* is mapped to `operator->` of `configured` objects. The *valueOf* operation is implemented with overloaded cast operators, and constructors. When it is not possible to infer the required type cast, like in case of a `printf` parameter, explicit execution of *valueOf* could be requested by using the object as a functor – that is, calling `operator()`.

Similarly to the preprocessing library, utilities for configurations are also provided with templates: `array_rank`, `item_config` and `has_ptr_item` are defined to produce the same result as their macro counterparts, but now as type-level values.

To understand how the library works, take a look at the hierarchy and the internal structure of configurable objects.

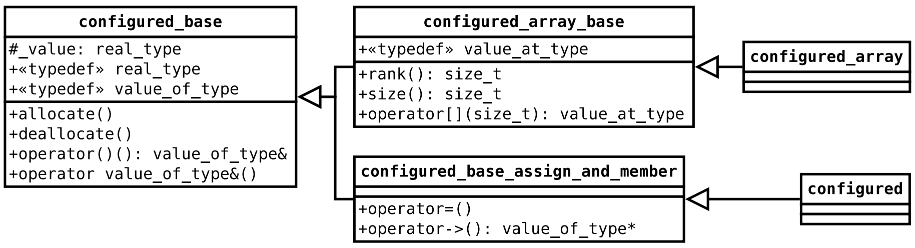

Figure 2: Class hierarchy of configurable object templates

The data wrapped by a configurable object is stored in a member named `_value`. Its type is derived using a template metaprogram implementation of generator function $\mathcal{G}$. For instance, the type of this member could be defined as `typedef struct input_t *real_type` for `input`, and `typedef char (*real_type[])[]` for `table`. The type casting operator and the functor implementation returns an object of `value_of_type`. This type is derived from the configuration after removing its outer *ptr* constructors, if any. This ensures that these operations implement the *valueOf* operation, by dereferencing the outer layer of pointers in the representation. Result of array indexing is `value_at_type`, that is a type derived from the *item type* of the corresponding configuration. For multi-dimensional arrays, indexing an outer dimension will return another object of type `configured_array`, with decreased rank and shorter configuration. Utility methods `rank` and `size` are returning the number of dimensions and the length of the outer dimension, respectively.

```
typedef PTR<SCALAR> CFG_t0;
typedef PTR<PTR<SCALAR>> CFG_tx;
typedef PTR<ARRAY<SCALAR>> CFG_ty;

typedef struct test_t
{
    int id;
    configured<bool, CFG_tx> x;
    configured_array<char, CFG_ty, true, 8> y;
} test_t;

configured<test_t, CFG_t0> t0;

t0->id = 0;
t0->x  = true;
strcpy(t0->y, "test");
printf("t0:␣{␣%d,␣%d,␣\"%s\"␣}\n", t0->id, t0->x(), t0->y());
```

Configuration of record fields is also very easy with this library. The user can reuse the `configured` and `configured_array` templates to declare arbitrary fields of records. When the automatic allocation and deallocation is turned on, the lifetime of the fields will be bound to the containing objects. As this is the default setting, the declaration of field `x` omits this parameter entirely.

Transforming existing software to use configurations only needs a little effort: the declaration of configurable objects should be replaced. Adaptation to operations is simple, thanks to operator overloading. Type checking will ensure that each access of the configured objects is correct.

Architecture specialization could be done similarly as in the previous technique. Every configuration builder template can be extended with meta-information about the memory hierarchy, and this can be processed in the implementation of the operations.

3.4 Source-to-source Transformations

The previous two implementations are intrusive: they require the modification of existing sources when applied. Using the C macro library requires to replace all declarations and operations where configurable objects involved. Working with the C++ template solution needs much less effort, but still, configurable objects have to be treated differently from ordinary data types.

This subsection shows a technique where applying configurations requires no preliminary changes on the source code. The idea is to use automated source-to-source transformations, that are parametrized with configurations. In this paper we focus on the description and validation of configurations, and not on the execution of transformations themselves. Although the presented method could be applied to any appropriate programming language, to make the results comparable with the previous sections, we decided to work on C source code.

The role of configurations is to describe the data layout of types and objects across the memory hierarchy. This information is actually represented in the existing source code as types. The key to generate configurations from C types, is to implement the inverse of generator function $\mathcal{G}$. Denote this function with $\mathcal{G}_C^{-1} : \mathcal{T} \to \mathcal{C} \times \mathcal{T}_B$, where $\mathcal{T}$ is the set of all possible C types, $\mathcal{C}$ is the set of configurations over a representation, and $\mathcal{T}_B$ is the set of C base types. The latter set contains all scalar, enumeration and record types (introduced either with a `struct` or `union` keyword). Type aliases should be fully resolved before applying $\mathcal{G}_C^{-1}$. This is a partial function, as C function pointer types are not supported by configurations.

$$\mathcal{G}_C^{-1}(\tau) = \begin{cases} scalar, & \text{when } \tau \in \mathcal{T}_B \\ ptr(\mathcal{G}_C^{-1}(\tau')), & \text{when } \tau = \tau'* \\ array(\mathcal{G}_C^{-1}(\tau')), & \text{when } \tau = \tau'[\,] \end{cases}$$

Once this function is available, the question is, how should these configurations be stored and presented for the C programmer, and how the changes will be given as instructions to the transformation tool. The solution is to create a *configuration skeleton* for the source files. The skeleton file will have the same structure as the input file, and is also a valid C source file, but contains no statements at all. To create a skeleton, process the input file as the following. Keep all type declarations, global objects and functions. For functions, only keep their local variable configurations in their outermost blocks. Drop all function parameters, and change return types to `void`. The preservation of the structure is needed to uniquely identify each type and object declarations in the source file later. As inner blocks of function bodies are dropped, the identification will not be available for local variables declared there, but the practice shows that being configurable is not a concern for short-living deeply nested locals. What remains in the skeleton file, is essentially a set of configurations, if we apply $\mathcal{G}_C^{-1}$ on the types in the remaining declarations. The user can rewrite the types in the skeleton file, and use it as a parameter to the transformation tool.

The transformation starts with an analyzer phase. Both an input file and a configuration file (possibly a modified skeleton) is parsed using a C parser, and a semantic analysis is executed to create a type-annotated abstract syntax tree of the sources. Function $\mathcal{G}_C^{-1}$ is applied on each type found in the configuration file. At the same time, the structurally corresponding declaration is looked up in the input file for each declaration in the configuration file. This makes it possible to apply $\mathcal{G}_C^{-1}$ also on the types found in the input declarations, and compare the old and the new configurations.

Each of the compared configurations must be *compatible*, that is, they must only differ in the number of *ptr* constructors applied. This restricts the changes to execute only data outsourcing or data inlining transformations. Base types corresponding to the two configurations must also be the same. As changes restricted to compatible configurations, there is no way to enforce an invalid configuration for an object. When there is no corresponding declaration found in the input file for an element in the configuration file, or the compatibility check fails, the transformation could signal an error and terminate.

For each of the compatible configurations, the differences can be collected. These differences will control the transformations executed on the input syntax tree. According to the old configuration, that is read from the input file, it could be checked that the objects are only used according to the operations of configured objects. When any non-matching access pattern is found, it can not be guaranteed that the transformation will be successful. This restriction is needed because C enables pointer arithmetics and other special ways to work with data. The modified code for the operations can only be derived correctly when it is ensured that in the input code the given object is used like an object configured previously with its old configuration.

Unfortunately, correct source-to-source transformations of C programs is highly non-trivial. Preservation of preprocessing structure, comments and formatting is very difficult. Type aliases in configured types are need to be handled with care. Consider the following:

```
// input file
typedef int speed;
typedef int* place;
speed vertical_speed;
// modified configuration file
speed *vertical_speed;
```

The old and new configurations of `vertical_speed` are *scalar* and *ptr*(*scalar*), which are compatible. Application of $\mathcal{G}_C^{-1}$ resolves all type aliases to produce the configurations and examine the base types, thus giving `int *vertical_speed` when re-synthesized with $\mathcal{G}$. It is incorrect, as picking another alias, like `place vertical_speed` would also be. To transform the declaration correctly, the new type should be written exactly in the same way as it was in the configuration file. Detection of incorrect access patterns and synthesizing the transformed code with the minimal amount of changes is also a challenging problem. For these reasons, we only implemented a few special cases in a tool based on the *Clang* compiler infrastructure.

4 Comparison of implementation techniques

This section contains a comparison of the previously presented configuration implementation techniques. The comparison is made based on the following 6 + 1 aspects.

Reconfiguration

Shows how much code is needed to be modified manually by the user when a configuration changes. For the next example, take a data outsourcing on the elements of an array, thus the configuration changes from *array*(*scalar*) to *array*(*ptr*(*scalar*)).

Miller: `elem_config []` ⇒ `elem_config [] -> dram`

C macros: `ARRAY(SCALAR)` ⇒ `ARRAY(PTR(SCALAR))`

C++ templates: `ARRAY<SCALAR>` ⇒ `ARRAY<PTR<SCALAR>>`

Source transformations: `elem_type arr[];` ⇒ `elem_type *arr[];`

Storage of configurations

Describes where and how the configurations are stored and what is their relationships to other parts of the code.

Miller Configurations are an integral part of the language. They can be stored in the same files as in the corresponding program code, or in separate files. In that case, configurations can be included into the program with a standard C preprocessor command. Configurations can be named and referenced later. Objects and configurations are loosely-coupled.

C macros The user can make her own configuration definitions as macros. This enables to store them either together with other parts of the code or separately. However, configuration of record fields is much less advanced as they can not belong together structurally as in the previous case.

C++ templates This solution has almost the same properties as C macros from this perspective. The only difference is that user-defined configurations can be defined as type aliases instead of macros. This helps to group related configurations together.

Source transformations Configurations are stored in separate files. They are valid C files, to ease their handling and processing both for the users and the tools. The structure of configurations follow the structure of the original code. Objects and configurations are tightly-coupled, as the configuration file must follow the structure of the original code. However, it is usually not a problem, as configurations can be deleted after the desired transformations are executed.

Syntax of configurations

Miller introduces a custom syntax for describing memory layers, locations and configurations, that the user must learn in the beginning.

C macros and **C++ templates** use almost the same syntax. The macro library uses a chain of macro invocations, while the other uses nested template applications. The low number of configuration constructors makes it easy to master it in minutes.

Source transformations use configuration files where each configuration is described as a standard C declaration. This is the most natural and straightforward way to express the changes in types, as it uses the same syntax to describe the transformation as the transformed language itself.

Amount of syntactic noise

What is the price of configurability in terms of readability and maintainability of code.

Miller has built-in construction to support configurations. Basically, every object is configured in the language, and the default operations are working accordingly. Configurability "comes for free" in this manner.

C macros add huge syntactic noise. Every declaration gets replaced by a syntactically more complicated one. This is also true for all operations done with these objects.

C++ templates add less noise than macros, thank to their better integration into the language and to operator overloading. However, non-trivial changes should be done when turning a non-configurable object into a configured one.

Source transformations have no syntactic noise at all, if the transformation engine is implemented well and could preserve all the appropriate comments and formatting of the original source.

Re-usability of configurations

How easy it is to re-use an existing configuration for a new piece of code.

Miller, **C macros** and **C++ templates** provide the ability to give names to configurations. They can be referenced later in any other configurations or get assigned to newly defined objects.

Source transformations use separate configurations for each declaration explicitly. However, as configuration files are C source files, it could be possible to use preprocessor macros to produce the configurations. Using single configuration file to transform multiple input files is also feasible.

Applicability to existing code base

Difficulties that arising when turning non-configured objects to configured ones.

Miller is a domain-specific language that was designed with memory hierarchies and configurability in mind from the very beginning. Every object and type expressed in *Miller* is configurable.

C macros may require a huge amount of highly error-prone, non-trivial modifications of the original code to turn a single object into a configured one.

C++ templates reduce the drawbacks of macros, by using operator overloading and instance methods. The type checker also ensures that the programmer rewrites them correctly.

Source transformations were designed especially to be applicable on existing code bases, so their cost is zero from this perspective.

Other benefits and drawbacks

Miller provides built-in configurations, and gives outstanding support for programming complex memory hierarchies. However, it is always hard to introduce a new language when there is an industrially accepted standard or a huge existing code base.

C macros are simple enough for demonstration and prototyping purposes, but are very impractical to use in real software.

C++ templates are completely acceptable, when the target platform provides a C++ compiler – which is not always the case. When the number of configurable variables is small, and the library is introduced at the beginning of the development, it could be a practical solution. Type safety of templates helps a lot compared to macros, but the template instantiation errors could be very long and confusing.

Source transformations are trying to combine the benefits of the built-in constructions with an already known language. However, implementing a transformation tool that handles all cases automatically and correctly, is a great challenge for difficult to refactor languages like C.

5 Conclusion

We introduced configurations to ease the change of data layout over different memory layers. The programmer only needs to change these configurations of the data structures, and the required highly non-trivial modifications will be propagated automatically over the algorithms. This makes data layout changes less error-prone and more efficient. This is especially important for performance tuning of applications on scratchpad-aware architectures. Four different implementation techniques were shown: one with built-in constructions of a domain-specific language, one with a C preprocessor library, a solution based on C++ template library, and a source-to-source transformation tool. The comparison shows that in practice, as it is expected, the C preprocessor metaprograms are not really usable. However under specific circumstances a C++ template metaprograms could implement configurable objects adequately. The real solution is a domain-specific language like *Miller*, or a sophisticated source-to-source transformation tool. The former is unfortunately difficult to introduce in industrial scenarios, while the latter is a great challenge to implement correctly for languages with a rich syntactic structure and complex semantics like C.

Acknowledgements

We wish to thank the current and former members of the research group at ELTE Software Technology Lab, including Gergely Dévai, Boldizsár Németh, Zoltán Csörnyei, Zoltán Kelemen and Dániel Leskó. Their useful and constructive work on this project made our publication possible.

References

[Abr04] David Abrahams and Aleksey Gurtovoy. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. *Addison Wesley Professional*, 2004.

[Ban02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. *Proceedings of the tenth international symposium on Hardware/software codesign.* ACM, 2002. 73–78

[Bpl] The Boost Library Preprocessor Subset for C/C++.
`http://www.boost.org/doc/libs/1_60_0/libs/preprocessor/doc/index.html`

[Car02] Carlos Carvalho. The gap between processor and memory speeds. In *Proceedings of IEEE International Conference on Control and Automation*, 2002.

[Kar14] Máté Karácsony. Modeling C preprocessor metaprograms using purely functional languages. *Proceedings of the 9th International Conference on Applied Informatics*, 2014. Vol. 2. 85–92.

[Krz11] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn and Wolfgang E Nagel. Scout: a source-to-source transformator for SIMD-optimizations. *Euro-Par 2011: Parallel Processing Workshops*, 2011. 137–145.

[Nem13] Boldizsár Németh and Zoltán Csörnyei. Stackless programming in Miller. *Acta Universitatis Sapientiae, Informatica 5.2*, 2013. 167–183.

[Nie97] Jarek Nieplocha, Robert Harrison, and Ian Foster. Explicit management of memory hierarchy. *Advances in High Performance Computing.* Springer Netherlands, 1997. 185–199.

[Pan97] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. *Proceedings of the 1997 European conference on Design and Test. IEEE Computer Society*, 1997.

[Pra14] Prashantha NR., Vikram TV. and Vaivaswatha N. Implementing Data Layout Optimizations in LLVM Framework. *LLVM Developers' Meeting*, October 28-29, 2014. `http://llvm.org/devmtg/2014-10/#talk14`

[Šin16] Artjoms Šinkarovs and Sven-Bodo Scholz. Type-driven Data Layouts for Improved Vectorisation. *Concurrency and Computation: Practice & Experience*, 2016. 2092–2119.

[Vel03] Todd L. Veldhuizen. C++ templates are turing complete.
`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670`, 2003.

[Ver02] Manish Verma, Stefan Steinke and Peter Marwedel. Data partitioning for maximal scratchpad usage. *Proceedings of the 2003 Asia and South Pacific Design Automation Conference.* ACM, 2003. 77–83.

[Ver04] Manish Verma, Lars Wehmeyer and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis.* ACM, 2004. 104–109.