

TALC: A Simple C Language Extension For Improved Performance and Code Maintainability

Jeff Keasler

Terry Jones

Dan Quinlan

Lawrence Livermore National Laboratory¹
P.O. Box 808
Livermore, CA 94550, USA
{keasler, trj, dquinlan} @llnl.gov

Abstract.

In this paper, we present TALC -- a small language extension for C and C++ suitable for applications that traverse common data structures such as large meshes or cubes. We make three contributions in this paper. First, we motivate the need for a new C/C++ extension focused on addressing emerging problem areas in performance and code maintainability. Second, we define the language extension and illustrate how it is employed in C. Third, we show the utility of such an extension by providing comparison code snippets that demonstrate advantages in both software maintainability and performance. Performance benefits of the extension are provided for several experiments resulting in up to 200% speedups over more conventional methods to achieve the same algorithm.

1. Introduction

This is a rewarding time for those practitioners who develop scientific applications. Modern computer architectures are more powerful than ever. The state of the art in mesh-based and grid-based computing now permits more accuracy and resolution than ever. Indeed, computer modeling is joining theory and empirical trial as a third component in physical sciences research.

The same advances that have extended the capabilities of scientific applications have also generated new and significant challenges. Even though today's computers have more potential than ever before, realizing that potential is becoming increasingly difficult due to new performance issues and the same old software engineering issues. To address these issues, we have developed TALC, a new C language extension focused on performance and software engineering problems. The name TALC is an acronym for Topologically Aware Layout C. The TALC user writes important data structure accesses, such as array accesses in a loop, into a higher-level syntax. Then, TALC automatically converts the higher-level syntax to optimized C for the underlying architecture and compiler, based on guidance provided by a separate 'schema' file described later. The optimized C improves performance by improving the resulting executable's cache hits and prefetch

¹ This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

2 Jeff Keasler, Terry Jones, and Dan Quinlan

effectiveness. In this way, the higher-level TALC code allows performance portability for existing and future architectures. The current version of TALC is designed for certain data structures common to mesh-based scientific applications or graphics/visualization applications (but the concept could be extended to a more general data layout optimization strategy).

The outline of our paper is as follows. In section 2, we provide the motivation for our work. In section 3, we describe the resulting design of TALC. Section 4 provides a description of ROSE, a C-scoping compiler infrastructure used in our implementation of TALC. Section 5 discusses our results, and Section 6 describes related work. Finally we provide our conclusions with Section 7.

2. Motivation

2.1 Performance Considerations

TALC was designed to improve application performance in several key aspects. Stall cycles have become an important factor in determining an applications performance. While little concern was given to stall cycles at the advent of many of today's most prevalent languages including C and C++, the dizzying speed of computer architecture innovation is now challenging many long held beliefs. For instance, previously conventional wisdom held that 'multiply is slow, but load and store is fast'. But today's conventional wisdom should be updated to a new view that recognizes the "*Memory wall*". [1] Now Load and store is slow, but multiply is fast. Modern microprocessors can take 200 clocks to access Dynamic Random Access Memory (DRAM), but floating-point multiplies may take only one clock cycle when pipelined. Contemporary architectures experience as much as 50% stall cycles for repetitive data-centric tasks as measured by "typical applications" such as the SPEC2000 CPU benchmark 181.mef. [2].

Moore's law, the observation that 'the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, doubling approximately every two years,' has given rise to a renaissance of architectural diversity. Today's supercomputer center is faced with a vast array of architectural choices where only one or two existed in yesteryear. With each choice, whether it be multi-core, Graphics GPU assisted core, added FPU core, added Vector unit core, and so on, one fact remains constant: the need for instructions and data is growing at a much faster rate than that of memory bandwidth. The use of multi-core and other advanced processor technologies are expected to expand over the next few years.

This is having a pronounced effect on application performance as seen when looking at the period between 1986 and 2006 using integer SPEC 2006 programs. [3] Performance improved by 52% per year between 1986 and 2002. However since 2002, performance has improved less than 20% per year. By 2006, processors will be a factor of three slower than if progress had continued at 52% per year. [4]

When combined, these trends suggest that a key component to current and future performance will be a programmer's ability to arrange the most

accessed data structures such that they are efficient with respect to stalls. Furthermore, the rapidly changing architectural landscape gives rise to the need for *performance portability*, the desirable trait that allows optimizations to carry-over from one computer architecture or compiler to another. Portability is essential, since many programs outlast their original platforms. In the supercomputing arena, a computer has a typical useful lifetime of 5 years, while many-decades-old applications codes are still in daily use. [5]

2.2 Software Engineering Considerations

Another major goal for TALC is to improve software maintainability. Scientific applications frequently evolve over many years through the efforts of small to medium size software teams. Given the complexity of the applications, it is not surprising that these applications are often several hundred thousand lines of code or more. Just as performance considerations are important, software engineering considerations must be given a near equal priority. TALC attempts to improve software by providing a stronger type system that can be checked at compile-time.

C and C++ are often criticized for being cryptic. C is able to adequately describe *how* an algorithm is to be realized in the low-level machine oriented domain, whereas its ability to describe the high-level *what* is to be computed is obscured. [6] On the other hand, C++ uses objects to describe *what* high level problem is being solved, but *how* that problem is being solved can be lost in a sea of method calls, each responsible for a small part of the task. TALC provides a happy medium between these two extremes. It allows for highly readable code that focuses on the *what* while at the same time providing an understanding of the *how*.

There is no simple way to determine the highest performance data structure across all possible architectures at the beginning of a software project. Once a data structure strategy is chosen for a large application, any speculative attempt to convert to another data structure for performance improvement will likely be time-consuming and introduce errors. Unfortunately, this limitation is at odds with the ongoing trend in the industry. Fueled by Moore's law, system architectures continue to change over time. To take full advantage of multi-core and other advanced processor technologies, a large amount of code will likely need to be rewritten, especially in the High Performance Computing (HPC) arena. TALC mitigates the effort of these rewrites by moving the data layout problem to a higher level so that an important class of layout choices can be implemented in minutes rather than weeks or months. It also introduces a notation that improves architectural portability.

3. The Design of TALC

TALC was designed to target applications using finite difference, finite volume, and finite element methods on structured or unstructured meshes. It also applies to many signal and image processing algorithms. A further important set of applications that can benefit from TALC are those that include multi-dimensional arrays of data.

TALC may be viewed as a centralized way to select the generation of array

4 Jeff Keasler, Terry Jones, and Dan Quinlan

or struct memory layouts for array data, with the optional capability to automatically generate subscript indices within the context of loops. The TALC extension can be incrementally introduced into existing C/C++ code. TALC is superficially similar to a vector language, but is not a vector language as described in most of the literature. [7,8]

3.1 The Data Layout Problem

Programmers have three basic choices for organizing arrays of data, as shown in Figure 1. The performance of each choice can vary greatly as code is ported from machine to machine and compiler to compiler. The execution of array statements involves inefficiencies stemming from several sources and the problem has been well documented by many researchers [9,10,11]. Our approach to performance within TALC is to use source-to-source transformation so that memory layout decisions can be globally applied at compile time.

As an example of the data layout problem, consider the performance implications of struct-like data layouts. There are many machines where a struct-like clustering is the most efficient due to hardware factors such as a reduced register pressure and enhanced memory streaming. On the other hand, other processors have hardware features or compiler optimizations that perform better with array-like layouts, for example the SSE instructions on the x86 architecture. You would like to be able to run in both environments while compiling from a static code base. This *can* be achieved with C++, but the resulting code is not necessarily easy to read or maintain.

Sample Data Structures Commonly Found In Mesh-Based Algorithms	
Array-Like double x[10000]; double y[10000]; double z[10000];	
Struct-Like struct coord { double x, y, z; } mesh[10000];	
Clustered-Struct struct coord { double x, y; } mesh[10000]; double z[10000];	

Figure 1. Sample Data Layouts

3.2 Topological Grouping of Data

It is desirable to incorporate high-level concepts from the problem domain into the data structures. This is one of the key ideas behind object oriented programming. An example of this might be to group data having similar

topological characteristics. By topology, we mean not only a similar physical topology, but also similar usage patterns.

Consider, for instance, coordinate components. Each coordinate component array is associated with a specific point in space, and each component array has similar descriptive character and the same length. Such arrays are referred to as *topologically equivalent*.

At times, it is useful to extend the concept of topological scope to encompass a wider class of arrays. Consider coordinate components and velocity components. There will often be a velocity associated with every coordinate and a coordinate associated with every velocity. In this case the coordinates and velocities are referred to as *topologically similar*.

In contrast, particle coordinates and mesh node coordinates may have similar coordinate component data, but particles often move independently from mesh coordinates. Furthermore, particles will probably be created and destroyed more often. For these reasons, one typically does not group particle coordinates with mesh coordinates. Such data items are referred to as *topologically dissimilar*, in spite of having the same underlying form of array data.

Grouping data by topological characteristics provides a powerful tool for manipulating data, especially where subsets are used heavily. TALC uses topological grouping as a way to control data layouts in memory and as a way to improve readability and correctness of code.

3.3 How Schemas address the Data Layout Problem and the Topological Grouping Problem

The TALC user begins by creating a single schema file for their whole program that describes the topological grouping of array data in the code. Topological grouping can be nested in hierarchies to indicate a subset relationship. Within each topological scope, the interleave of array data can be controlled by the ordering of declarations.

Figure 2 illustrates three different sample schemas. Topological scopes are delimited by the **View** keyword. Array data is declared with the **Field** keyword. When a name appears by itself after a Field keyword, it indicates an array-like representation. When multiple names appear after a Field keyword, it represents a struct-like array representation.

The schema describes relationships between arrays and the layout of those arrays in memory. It does not allocate memory. Instead, it cooperates with a memory allocation API to impose rules on how allocations will occur. The compiler can then use the schema to generate structs or arrays as appropriate at compile time. Since the groupings can be hierarchical, subtle relationships among groups of arrays can easily be captured and exploited.

6 Jeff Keasler, Terry Jones, and Dan Quinlan

Simple Schemas in TALC																							
Array-Like double x[10000]; double y[10000]; double z[10000];	<table border="1"> <tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>...</td></tr> <tr><td>y</td><td>y</td><td>y</td><td>y</td><td>y</td><td>y</td><td>...</td></tr> <tr><td>z</td><td>z</td><td>z</td><td>z</td><td>z</td><td>z</td><td>...</td></tr> </table>	x	x	x	x	x	x	...	y	y	y	y	y	y	...	z	z	z	z	z	z	...	View coord Field x Field y Field z View
x	x	x	x	x	x	...																	
y	y	y	y	y	y	...																	
z	z	z	z	z	z	...																	
Struct-Like struct coord { double x, y, z ; } mesh[10000];	<table border="1"> <tr><td>x</td><td>y</td><td>z</td><td>x</td><td>y</td><td>z</td><td>...</td></tr> <tr><td>x</td><td>y</td><td>z</td><td>x</td><td>y</td><td>z</td><td>...</td></tr> <tr><td>x</td><td>y</td><td>z</td><td>x</td><td>y</td><td>z</td><td>...</td></tr> </table>	x	y	z	x	y	z	...	x	y	z	x	y	z	...	x	y	z	x	y	z	...	View coord Field x, y, z View
x	y	z	x	y	z	...																	
x	y	z	x	y	z	...																	
x	y	z	x	y	z	...																	
Clustered-Struct struct coord { double x, y ; } mesh[10000]; double z[10000];	<table border="1"> <tr><td>x</td><td>y</td><td>x</td><td>y</td><td>x</td><td>y</td><td>...</td></tr> <tr><td>x</td><td>y</td><td>x</td><td>y</td><td>x</td><td>y</td><td>...</td></tr> <tr><td>z</td><td>z</td><td>z</td><td>z</td><td>z</td><td>z</td><td>...</td></tr> </table>	x	y	x	y	x	y	...	x	y	x	y	x	y	...	z	z	z	z	z	z	...	View coord Field x, y Field z View
x	y	x	y	x	y	...																	
x	y	x	y	x	y	...																	
z	z	z	z	z	z	...																	

Figure 2. The rightmost column illustrates simple schemas for Array-like, Struct-like, and Clustered-Struct access

Using combinations of array-style, struct-style, and clustered-struct style data layouts, the TALC user defines a schema for the data structures to be managed by TALC. Schemas may include arbitrary levels of nesting as shown in Figure 3. At present, the TALC user must manually generate schemas.

A Typical Schema in TALC
View element Field deltz Field dxx dyy dzz dxy dxz dyz Field sxx syy Field txy txz tyz Field v vnew View material Field delts Field newSxx newSyy newSzz newTxy newTxz newTyz View View

Figure 3. Sample Vector Schema Incorporating all context of Array-like, Struct-like, and Clustered-Struct access

3.4 Writing TALC Code

Once a TALC schema has been created, code fragments that access arrays can be converted to use the TALC extension. Figure 4 below illustrates sample TALC code that could work with the schema presented in Figure 3; this sample illustrates accessing mesh elements within a loop. Such code

TALC: A Simple C Language Extension for Performance and Code Maintainability 7

snippets are common in unstructured mesh based codes. Note that variables `dxx`, `dyy`, `sxx`, `syy`, `newSxx`, `newSyy`, and so on appear in both the schema and the TALC code in Figure 4. For such variables, the TALC compiler will determine a data layout for the destination architecture and compiler as specified by the schema. The resulting data layout could take many forms and gives the TALC user added flexibility to pursue various goals.

The process of generating the resulting standard C code is accomplished by the TALC compiler without additional annotations to source code. The choice of *which* access pattern (e.g. array-like, clustered-struct like, etc.) is currently specified manually; it is based from the schema (which must be manually generated at present).

Sample Code in TALC syntax

Compiler will use schema to select from among
Array-like, Struct-like, Clustered-Struct

```
double quarterDelta = 0.25 * deltaTime;

while(material) {
  real8 szz = - sxx - syy ;

  deltz += quarterDelta * (vnew + v) *
  ( dxx * (sxx + newSxx) +  dyy * (syy + newSyy) +
  dzz * (szz + newSzz) +
  2.*dxy * (txy + newTxy) + 2.*dxz * (txz + newTxz) +
  2.*dyz * (tyz + newTyz) );

  delts += quarterDelta * (vnew + v) *
  ( dxx * sxx +  dyy * syy +  dzz * szz +
  2.*dxy * txy + 2.*dxz * txz + 2.*dyz * tyz );
}
```

Figure 4. Sample Access Patterns In A Loop using TALC

The single TALC code snippet in Figure 4, in combination with appropriate schemas, can generate any of the resulting code snippets shown in Figure 5 below. The result of the TALC compiler is standard C code with all instances of the schema variables changed to their appropriate form. In Figure 4, note that all mesh indices have been stripped from all array operations in the loop, and the for loop has been changed to a while loop over an *Indexset* named material. IndexSets are a TALC construct needed to define how loops are traversed. Indexsets are allocated to correspond to a schema View. Indexsets behave much like “regions” in the ZPL [12] language, but Indexsets can be structured or unstructured.

8 Jeff Keasler, Terry Jones, and Dan Quinlan

Array-Like Access	<table border="1" style="font-size: 0.8em; border-collapse: collapse;"> <tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>...</td></tr> <tr><td>y</td><td>y</td><td>y</td><td>y</td><td>y</td><td>y</td><td>...</td></tr> <tr><td>z</td><td>z</td><td>z</td><td>z</td><td>z</td><td>z</td><td>...</td></tr> </table>	x	x	x	x	x	x	...	y	y	y	y	y	y	...	z	z	z	z	z	z	...
x	x	x	x	x	x	...																
y	y	y	y	y	y	...																
z	z	z	z	z	z	...																
<pre> real8 quarterDelta = 0.25 * deltaTime; for (int i = 0 ; i < material_length ; i++){ int index = material_map[i]; real8 szz = - sxx[index] - syy[index] ; deltz[index] += quarterDelta * (vnew[index] + v[index]) * (dxx[index] * (sxx[index] + newSxx[i]) + dyy[index] * (syy[index] + newSyy[i]) + dzz[index] * (szz + newSzz[i]) + 2.*dxy[index] * (txy[index] + newTxy[i]) + 2.*dxz[index] * (txz[index] + newTxz[i]) + 2.*dyz[index] * (tyz[index] + newTyz[i])) ; delts[i] += quarterDelta * (vnew[index] + v[index]) * (dxx[index] * sxx[index] + dyy[index] * syy[index] + dzz[index] * szz + 2.*dxy[index] * txy[index] + 2.*dxz[index] * txz[index] + 2.*dyz[index] * tyz[index]) ; } </pre>																						

Figure 5a. Sample Access Patterns In A Loop

Struct-Like Access	<table border="1" style="font-size: 0.8em; border-collapse: collapse;"> <tr><td>x</td><td>y</td><td>z</td><td>x</td><td>y</td><td>z</td><td>...</td></tr> <tr><td>x</td><td>y</td><td>z</td><td>x</td><td>y</td><td>z</td><td>...</td></tr> <tr><td>x</td><td>y</td><td>z</td><td>x</td><td>y</td><td>z</td><td>...</td></tr> </table>	x	y	z	x	y	z	...	x	y	z	x	y	z	...	x	y	z	x	y	z	...
x	y	z	x	y	z	...																
x	y	z	x	y	z	...																
x	y	z	x	y	z	...																
<pre> for (int i = 0 ; i < material_length ; i++){ int index = material_map[i]; real8 szz = - elem[index].sxx - elem[index].syy ; elem[index].deltz += quarterDelta * (elem[index].vnew + elem[index].v) * (elem[index].dxx * (elem[index].sxx + materialElem[i].newSxx) + elem[index].dyy * (elem[index].syy + materialElem[i].newSyy) + elem[index].dzz * (szz + materialElem[i].newSzz) + 2.*elem[index].dxy * (elem[index].txy + materialElem[i].newTxy) + 2.*elem[index].dxz * (elem[index].txz + materialElem[i].newTxz) + 2.*elem[index].dyz * (elem[index].tyz + materialElem[i].newTyz)) ; materialElem[i].delts += quarterDelta * (elem[index].vnew + elem[index].v) * (elem[index].dxx * elem[index].sxx + elem[index].dyy * elem[index].syy + elem[index].dzz * szz + 2.*elem[index].dxy * elem[index].txy + 2.*elem[index].dxz * elem[index].txz + 2.*elem[index].dyz * elem[index].tyz) ; } </pre>																						

Figure 5b. The same code as 5a, but with a different data structure (Struct-like).

Clustered-Struct Access	<table border="1" style="font-size: 0.8em; border-collapse: collapse;"> <tr><td>x</td><td>y</td><td>x</td><td>y</td><td>x</td><td>y</td><td>...</td></tr> <tr><td>x</td><td>y</td><td>x</td><td>y</td><td>x</td><td>y</td><td>...</td></tr> <tr><td>z</td><td>z</td><td>z</td><td>z</td><td>z</td><td>z</td><td>...</td></tr> </table>	x	y	x	y	x	y	...	x	y	x	y	x	y	...	z	z	z	z	z	z	...
x	y	x	y	x	y	...																
x	y	x	y	x	y	...																
z	z	z	z	z	z	...																
<pre> for (int i = 0 ; i < material_length ; i++){ int index = material_map[i]; real8 szz = - elem[index].sxx - elem[index].syy ; deltz[index] += quarterDelta * (volume[index].vnew + volume[index].v) * (deform[index].dxx * (stress[index].sxx + materialStress[i].newSxx) + deform[index].dyy * (stress[index].syy + materialStress[i].newSyy) + deform[index].dzz * (szz + materialStress[i].newSzz) + 2.*deform[index].dxy * (stress[index].txy + materialStress[i].newTxy) + 2.*deform[index].dxz * (stress[index].txz + materialStress[i].newTxz) + 2.*deform[index].dyz * (stress[index].tyz + materialStress[i].newTyz)) ; delts[i] += quarterDelta * (volume[index].vnew + volume[index].v) * (deform[index].dxx * stress[index].sxx + deform[index].dyy * stress[index].syy + deform[index].dzz * szz + 2.*deform[index].dxy * stress[index].txy + 2.*deform[index].dxz * stress[index].txz + 2.*deform[index].dyz * stress[index].tyz) ; } </pre>																						

Figure 5c. The same code as 5a, but with a clustered-struct data structure.

Note that there are no subscripts in the TALC code shown in Figure 4 since array indices can be automatically generated from the schema. A complete list of the automatic array index transformations is included in Figure 6.

Sample TALC Transformations (performed inside while loop)		
Vector variable used inside a while block	TALC Transformation in a for statement	Comments
var	var[index] or struct[index].var	autogenerate index for correct memory interleave and topological subset, base on schema and compiler option
var[xxx]	var[xxx] or struct[xxx].var	override - do what user asks for
var(offset)	var[index+offset] or struct[index+offset].var	for stencil calculations

Figure 6. The language extension

3.5 Summary For Converting Code to TALC

There are four steps involved in a general strategy for converting mesh-related algorithms from C or C++ to TALC, namely:

- First, a schema is created to identify topological relationships among Fields in the original C source code.
- Second, **for** loops over index variables are changed to **while** loops over IndexSets.
- Third, the original C source code is modified by stripping indices off of arrays in loops wherever appropriate.
- Fourth, the memory allocations for arrays of interest must use the TALC memory allocator. Many scientific codes have centralized their allocation policy (wrapped malloc, database, etc.) which may simplify this step.

The current TALC compiler framework will instantiate source code for a given schema, or directly produce a '.o' file.

4. Implementing a TALC Prototype

The economics and maturation of new language and compiler designs make it particularly difficult for highly specialized languages to appear and be accepted by developers of large scale applications. Though significant aspects of our approach are language independent, our research work has targeted the optimization of array constructs in C/C++. The framework developed to support this research, ROSE [13], allows us to express optimizations based on an abstract C++ grammar, eliminating the syntactical idiosyncrasies of C++ in the specification of a transformation.

4.1 The ROSE Framework

ROSE is a compiler infrastructure for building source-to-source translators (compilers that take in source code, implement programmable rewrites to the Abstract Syntax Tree (AST) and then regenerate source code). ROSE addresses Fortran 2003, C, and C++ and provides a high level AST representation for custom analysis and transformation of large scale

10 Jeff Keasler, Terry Jones, and Dan Quinlan

Department of Energy (DOE) applications. An explicit goal of ROSE has been to significantly lower the barrier to research work having impact on DOE scale applications. ROSE forms the basis for a wide range of external and internal research projects.

The design of the Intermediate Representation (IR) is object-oriented and defines IR nodes for each language construct of Fortran 2003, C, C++ and shared approximately 85% of the IR nodes between all three supported languages. ROSE supports all the numerous details of reproducing the generated code indistinguishable from the original input code because this is important to our user base (details include: formatting, comments, CPP directives, etc.). The interface to the IR used in ROSE is based on SAGE [14] and is similar to few other object-oriented IRs which attempt to preserve the source level of detail. While we have specific goals for this work within research on the general optimization of high-level abstractions, ROSE is a general infrastructure designed as a library for building source-based tools.

ROSE is designed to use multiple internal language front-ends and parsers. For C and C++ we use the EDG front-end. [15] For Fortran 2003 we use the Los Alamos Open Fortran Parser (OFP). [16] Older versions of Fortran, specifically Fortran 95, Fortran 90, Fortran 77, and Fortran 66 are also supported. Special attention has been paid to handling fixed and free format codes and generating fixed for free format output. For C, ROSE supports the full C89 and C99 versions, plus most of the gnu extensions. For C++, ROSE supports the full C++ language and permits analysis and transformation of all instantiated templates.

For all languages, ROSE provides full type evaluation and semantic analysis. The C and C++ work has been tested on a number of different million line applications within DOE. The Fortran support is the most recent work, and consequently is not as robust yet.

Support in ROSE includes: 1) the internal loop optimizers that can be leveraged in building optimization tools. 2) attribute grammar based AST traversals to make it easy to define custom program analysis and transformation passes. 3) shared memory and distributed memory parallel attribute evaluation for writing parallel program analysis (when program analysis performance is critical). 4) an extensible tool, Compass, for defining and evaluating properties on the AST (used as a basis for security analysis research). 5) Additional predefined optimizations (inlining, Partial Redundancy Elimination (PRE), etc.) and useful transformations to support external research (outlining). 6) Many more features are included in ROSE, which has been an ongoing project for many years.

4.2 How ROSE Ensures Conformance for TALC

Because TALC is implemented using ROSE, it derives its robustness and conformance to the C language from ROSE (which derives its conformance to C via the EDG front-end). Because the front-end is not modified within the TALC work, there is no formal language extension. More precisely, TALC represents an extension as custom compiler support for a domain-specific abstraction. It is however, superficially indistinguishable from a true language extension. Additionally as a domain-specific abstraction optimized via custom transformations written using ROSE as a source-to-source translator, the work is significantly more portable than a formal language extension.

5. Results

To evaluate the performance capabilities of TALC, we evaluated several common mesh access patterns on three different architectures with several resulting data layouts as generated by TALC.

5.1 Measurements Taken With TALC

First, we measured the wallclock time needed to traverse a domain of 12000 elements containing two sparse material subsets of 8000 and 4000 elements. The 8000 element subset is traversed using the algorithm shown in Figure 4. Normalized time is shown in the Y-axis with (1.0) being the worst performing data layout for the given architecture: a value of .5 represents a 2x speedup over a value of 1.0. Each architecture is normalized independently. Figure 7 shows the results of this test. Note that the Itanium2 runs at about the same speed no matter how we code the algorithm. Note that Array-like is fastest for the Itanium2, while struct-like is the fastest for the Power5. Also, the spread between the times on the power5 is large.

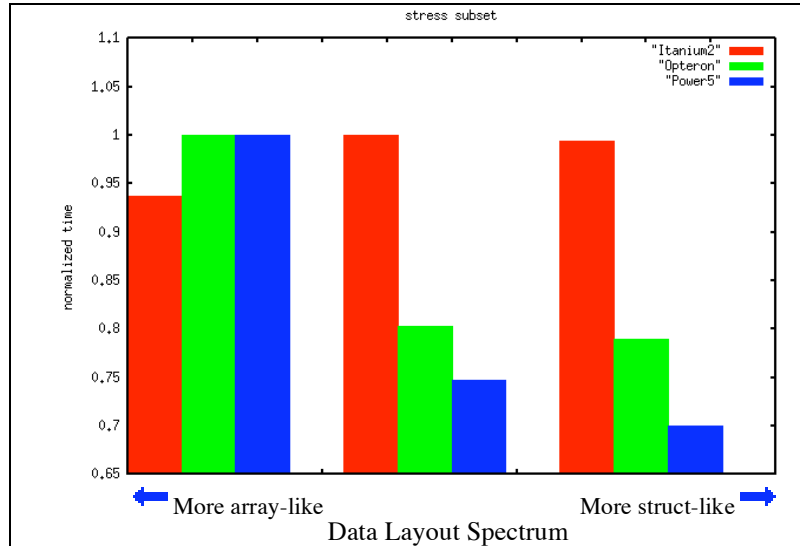


Figure 7. Mesh traversal performance for three data layouts on three architectures

In Table 1, we show the L1 cache hit ratio for each data layout in Figure 7 on the Opteron processor as measured by hardware counters using PERFlib. [17] Looking at the L1 cache hit ratio, one would assume that the Array-like layout is the most efficient. However if you sum the cache hits and misses for each layout, you find that the Struct-like access has the lowest number of memory accesses and thus the lowest number of stalls.

Memory Interleave	Hit Count	Miss Count	Hit Ratio
Array-like	3955732080	286239697	93.3%
Intermediate	2842569424	281404535	91.0%
Struct-like	2769568352	273753504	91.1%

Table 1. L1 cache statistics for the stress problem on the Opteron processor.

12 Jeff Keasler, Terry Jones, and Dan Quinlan

The reason for the lower memory access is that base-offset addressing can be used on a single pointer with structs, but when individual arrays are used, each array pointer must be given its own register. Since there are a limited number of registers on the x86 architecture, register pressure forces individual array pointers to be swapped to memory as the algorithm progresses. The lesson learned here is that raw cache hit numbers can be misleading when optimizing for performance.

For our second test, we calculated volumes for hexahedral elements in a 420K element mesh. Each element touches eight nodes, and each node has three coordinate components. As with the previous performance results, normalized time is shown in the Y-axis with (1.0) being the worst performing data layout for the given architecture. A lower value represents a faster runtime. Our results presented in Figure 8 show that putting x, y, z in a struct (e.g. class Point) is the slowest possible choice overall, even though most people use a coord struct in their code!

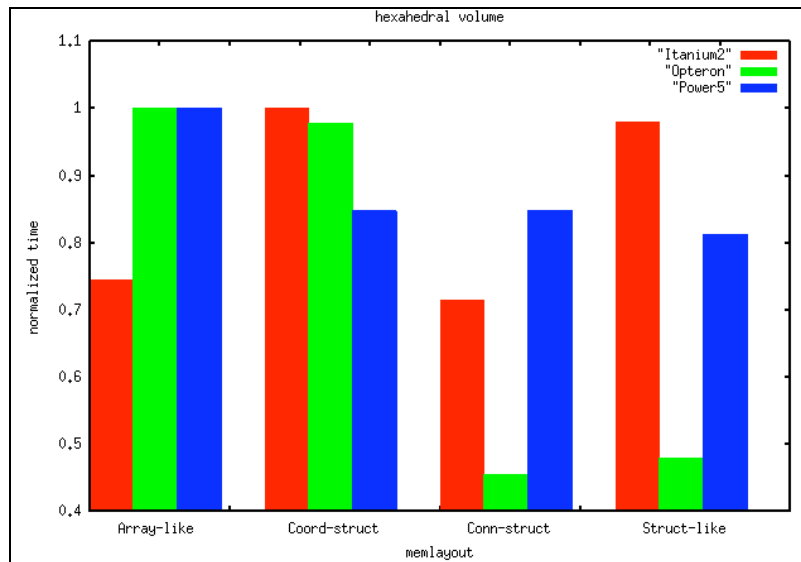


Figure 8. Brick volume calculation performance for four data layouts on three architectures.

Finally, we measured normalized times for a Jacobi Iteration across a two-dimensional mesh. Such access patterns can be employed as iterative solutions to a Poisson problem. For our results, each interior element is updated based on its neighbors to the north, south, east, and west. These results are shown in Figure 9 below.

Speedups exceeding 2x are a huge win to these mesh-based applications. While effective cache utilization is a key element, our observed gains go past cache hit ratios. As shown, a high cache hit ratio is not necessarily a good measure of a well written, high performance implementation of an algorithm.

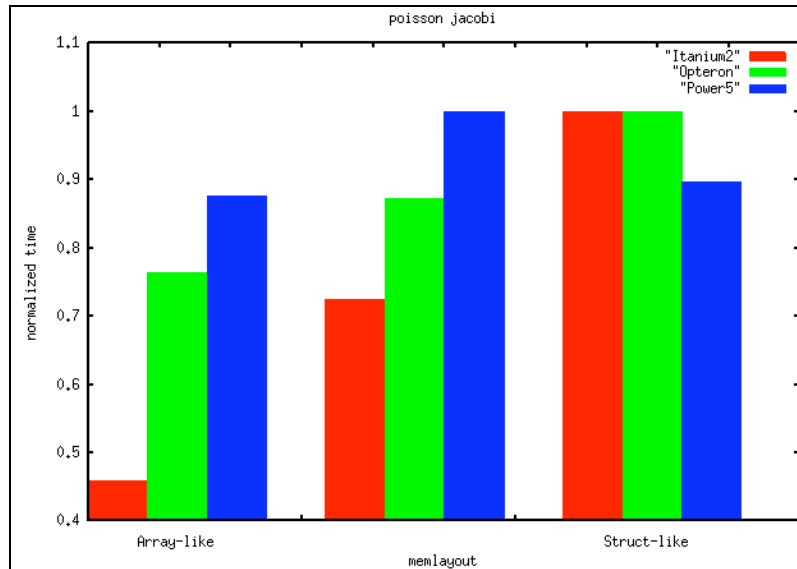


Figure 9. Jacobi Iteration performance for three data layouts on three architectures.

The following interpretations can be drawn from these results. First, data structure choices are system architecture, compiler, and problem domain dependent. Second, data structure choices can result in a 2x performance difference on a given machine. Third, choosing a data structure that is best for one machine can be the worst for another machine.

There is no simple way to determine the best data structure across all possible architectures at the beginning of a software project. Once a data structure strategy has been selected, it will likely be time-consuming and error-prone to convert to another. Since system architectures change over time, the need for an automated solution such as TALC becomes a valuable tool for those interested in performance portability.

5.2 TALC Benefits

As discussed in Sections 3.4 and 5.1, TALC provides performance portability for mesh-based applications across a diversity of system architectures. TALC centralizes the traversal policy permitting cache blocking and data movement on a variety of architectural models (multi-core, NUMA, GPGPU). By allowing the user to choose the best data layout, TALC enables up to 2x performance improvements.

TALC offers several important features that improve code readability and correctness. Removal of explicit subscripting can reduce or eliminate indexing errors. The resulting coding of loops enhances the readability of equations. The TALC schema enables the compiler to provide stronger type checking operations to enforce topological constraints. For instance, if an element centered pressure is assigned to a node centered coordinate, the compiler can immediately catch this nonsensical assignment. The use of Indexsets also allows better bounds checking on arrays than could be done with raw mallocs. Finally, the schema can simplify the refactoring of data structures when new

algorithms or physics packages are introduced, if for instance a variable defined over the mesh were to be moved to a variable defined only over a material subset.

The ability for TALC to handle these types of changes without requiring corresponding code changes is a major advantage. A large hydrodynamics code recently went through a hand-refactoring process that could have been done by TALC. Even though the code was hundreds of thousands of lines, the code team found that changing the memory interleave of some arrays resulted in a 42-100% speedup of the hydrodynamics depending on the problem being solved and the machine being used. The hydrodynamics portion can be the dominant portion of the runtime for many physics applications, so doubling the performance with just a change of data structures is impressive. Part of the performance gain probably came from the compiler recognizing extra optimizations that could be applied (same compiler flags), and the rest came from different cache latency characteristics.

Finally, the advantage of grouping arrays topologically is that they can often be nested in inheritance hierarchies. For example, one array class could contain array data common to all the nodes of a mesh, while another class could contain extra array data pertaining to a subset of the nodes. Indexsets can be used to map array indices in the subset to corresponding indices in the larger mesh.

6. Related Work

An important topic covered by this paper is data organization, which is separate from the topic of data layout. An excellent introduction to the benefits of data organization can be found in the paper, “Collection Level Polymorphism: A Path to High Performance C++ Applications” by Luke [18]. Another excellent data organization scheme is described in the paper, “Janus – a C++ Template Library for Parallel Dynamic Mesh applications” by Gerlach, et al . [19]

TALC tries to map the concepts presented in these two papers to a form that has the look and feel of standard C. In doing so, the introduction of an Indexset object was needed. An Indexset is much like a ZPL [12] Region, however an Indexset in TALC can be structured or unstructured. Also unlike ZPL, TALC does not try to be a language in and of itself, but merely extends the concept of how subscripting operations should work in the context of the C and C++ languages.

The TALC project is currently exploring an extension to heterogeneous programming environments by leveraging the RapidMind [20] platform. Other work in heterogeneous programming environments can be found in the paper “HMPP: A Hybrid Multi-core Parallel Programming Environment” by Romain Dolbeau, et al. [21]

7. Conclusions and Future Work

TALC is an extension to C that improves data layout and code maintainability for applications that traverse common data structures such as large meshes or cubes.

The use of TALC provides many benefits for mesh-based projects. These applications frequently encompass hundreds of thousands of lines of code and are therefore prime candidates for software maintainability improvements. Furthermore, such applications are frequently very CPU intensive and are also prime candidates for performance improvements.

To achieve these benefits, TALC uses topological grouping as a way to control data layouts in memory and as a way to improve readability and correctness. We have implemented TALC using the ROSE compiler infrastructure.

Our results demonstrate the minimal changes necessary to rewrite existing mesh-based loop into TALC as well as the potential for performance gains and software maintainability improvements. Performance portability is likely to become a necessary part of programming in the near future. TALC transformations provide a unified way of running effectively on a diversity of system architectures while achieving up to 2x improvements in runtime performance.

References

1. Wulf, W. A. and McKee, S. A. 1995. Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23, 1 (Mar. 1995), 20-24. DOI= <http://doi.acm.org/10.1145/216585.216588>
2. Gurumani, S. T. and Milenkovic, A. 2004. Execution characteristics of SPEC CPU2000 benchmarks: Intel C++ vs. Microsoft VC++. In Proceedings of the 42nd Annual Southeast Regional Conference (Huntsville, Alabama, April 02 - 03, 2004). ACM-SE 42. ACM, New York, NY, 261-266. DOI= <http://doi.acm.org/10.1145/986537.986599>
3. Spec Benchmark Organization: <http://www.spec.org>
4. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec 2006. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>
5. National Research Council. 2005. Getting Up To Speed: The Future of Supercomputing”, National Research Council, The National Academies Press. Washington D.C.
6. Ian Joyner, C++? A Critique of C++. 1992. <http://www.literateprogramming.com/c++critique.pdf>
7. Perrott, R. H. 1979. A Language for Array and Vector Processors. ACM Trans. Program. Lang. Syst. 1, 2 (Oct. 1979), 177-195. DOI= <http://doi.acm.org/10.1145/357073.357075>
8. Santavy, M. and Labute, P., SVL: The Scientific Vector Language, 1997. http://www.chemcomp.com/Journal_of_CCG/Features/svl.htm
9. Bassetti, F., Davis, K., Quinlan, D. A Comparison of Performance-enhancing Strategies for Parallel Numerical Object-Oriented Frameworks In Proceedings of the first International Scientific

16 **Jeff Keasler, Terry Jones, and Dan Quinlan**

- Computing in Object-Oriented Parallel Environments (ISCOPE) Conference, Marina del Rey, California, Dec, 1997
10. Karmesin, et al. Array Design and Expression Evaluation in POOMA II. In Proceeding of the Second International Symposium, ISCOPE 98, Santa Fe, NM December 1998
 11. Bassetti, F., Davis, K., Quinlan, D. Optimizing Transformations of Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures In Proceedings of the ISCOPE' Conference, Santa Fe, New Mexico, Dec 13-16 1998
 12. Snyder, L. 2007. The design and development of ZPL. In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (San Diego, California, June 09 - 10, 2007). HOPL III. ACM, New York, NY, 8-1-8-37. DOI=<http://doi.acm.org/10.1145/1238844.1238852>. See also <http://www.cs.washington.edu/research/zpl/home/index.html>
 13. D. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. LLNL Technical Report UCRL-ID-136515 (Nov. 1999). <https://e-reports-ext.llnl.gov/pdf/237284.pdf>
 14. B. Francois et. al. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In Proceedings of the Second Annual Object- Oriented Numerics conference, 1994.
 15. Edison Design Group <http://www.edg.com>
 16. Open FORTRAN Parser, <http://fortran-parser.sourceforge.net/>
 17. [Jeff Brown. PERFlib. Los Alamos National Laboratory.](#)
 18. Edward Luke. Collection Level Polymorphism: A Path To High Performance C++. Proceedings of The Fourth Annual Object-Oriented Numerics Conference (OONSCI '96), Mississippi State University. <http://citeseer.ist.psu.edu/15741.html>
 19. Gerlach, J., Sato, M., and Ishikawa, Y. 1998. Janus: A C++ Template Library for Parallel Dynamic Mesh Applications. In Proceedings of the Second international Symposium on Computing in Object-Oriented Parallel Environments (December 08 - 11, 1998). D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds. Lecture Notes In Computer Science, vol. 1505. Springer-Verlag, London, 215-222.
 20. McCool, M. D. and D'Amora, B. 2006. Programming using RapidMind on the Cell BE. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (Tampa, Florida, November 11 - 17, 2006). SC '06. ACM, New York, NY, 222. DOI=<http://doi.acm.org/10.1145/1188455.1188686>. See also <http://www.rapidmind.com>
 21. Dolbeau, Romain, Bihan, Stéphane, and Bodin, François. HMPP™: A Hybrid Multi-core Parallel Programming Environment. Proceedings of First Workshop on General Purpose Processing on Graphics Processing Units, Boston, MA. (Oct. 2007). : <http://www.caps-entreprise.com/en/documentation/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.pdf>