LA-UR 98-1966

CONF-981063--

Approved for public release; distribution is unlimited

Title: Temporal Locality Optimizations for Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures

Author(s): Federico Bassetti
Kei Davis
Dan Quinlan

Submitted to: Parallel and Distributed Computing Systems
October 28-31, 1998
Las Vegas, Nevada

Los Alamos
National Laboratory

MASTER

DISCLAIMER

Temporal Locality Optimizations for Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures

Federico Bassetti, Kei Davis, and Dan Quinlan
Scientific Computing Group CIC-19
Computing, Information, and Communications Division
Los Alamos NM, 87545, USA
Tel: (505) 667-7492, Fax: (505) 667-1126
{fede,kei,dquinlan}@lanl.gov

Keywords: parallel object-oriented frameworks, cache utilization, performance analysis.

Abstract

High-performance scientific computing relies increasingly on high-level large-scale object-oriented software frameworks to manage both algorithmic complexity and the complexities of parallelism: distributed data management, process management, inter-process communication, and load balancing. This encapsulation of data management, together with the prescribed semantics of a typical fundamental component of such object-oriented frameworks—a parallel or serial array-class library—provides an opportunity for increasingly sophisticated compile-time optimization techniques. This paper describes a technique for introducing cache blocking suitable for certain classes of numerical algorithms, demonstrates and analyzes the resulting performance gains, and indicates how this optimizating transformation is being automated.

1 Introduction

Current ambitions and future plans for scientific applications, in part stimulated by the Accelerated Scientific Computing Initiative (ASCI), practically mandate the use of higher-level approaches to software development, particularly more powerful organizational and programming tools and paradigms for managing algorithmic complexity, making parallelism largely transparent, and more recently, providing methods for code optimization that could not be reasonably expected of a conventional compiler.

An increasingly popular approach is the use of C++ object-oriented software *frameworks* or hierarchies of extensible libraries. The use of such frameworks has greatly simplified (in fact, made practicable) the development of complex serial and parallel scientific applications at Los Alamos National Laboratory (LANL) and elsewhere. Examples from LANL include Overture [3] which supports complex geometries, adaptive mesh refinement (AMR), and overlapping grid computations in addition to more basic single logically rectangular grid computations. Concerns about performance, particularly relative for FORTRAN 77, are the single greatest impediment to widespread acceptance of such frameworks.

Our past and current work has demonstrated three broad areas where potential performance (relative to theoretical machine capabilities) is lost: language implementation issues (which we

address for C++ elsewhere [2]); communication issues (which may be algorithmic, or communication library problems); and perhaps most important, with the trend toward ever-deeper memory hierarchies and the widening differences in processor and main-memory bandwidth, poor cache utilization. This lattermost issue is the subject of this paper. Our ultimate goal is to produce FORTRAN 77 performance (or better, in a sense described later) from the computationally intensive components of such C++ frameworks, though the transformation we describe is language independent.

This paper presents a transformation for improving cache utilization, and demonstrates it application in Overture. The transformation improves *temporal locality* in the execution of stencil operations; we refer to the transformation as *temporal blocking*. Though relatively complex in its implementation for general array objects, temporal blocking can be fully automated.

2 Array Classes

In scientific computing arrays are the fundamental data structure, and as such compilers attempt a large number of optimizations for their manipulation. For the same reason, array class libraries are ubiquitous fundamental components of object-oriented frameworks. Examples include A++/P++ [7] in Overture, *valarray* in the C++ standard library [8], Template Numerical ToolKit (TNT) [6], and the GNU Scientific Software Library (GNUSSL) [4]. Typically, in an application using a framework such as these, the bulk of the computation is performed by the underlying array class library; such array computations are the focus of this work.

The target of transformation is the A++/P++ array class library which provides both serial and parallel array implementations. The temporal blocking transform is a serial transformation but applies to the parallel case with slight modification. Transformation of A++/P++ array statements is practicable because, by definition, they have no hidden or implicit loop dependence (and thus can easily have their execution distributed over multiple processors). Indeed, it is common to design array classes so that optimization is reasonably straightforward—this is clearly stated, for example, for *valarray*. Such statements vectorize well, but our focus is on cache-based architectures because they are increasingly common in both large- and small-scale parallel machines.

An example of an A++/P++ array statement is

```
for (int n=0; n != N; n++)  // Outer iteration
  A(I) = ( A(I-1,J) + A(I+1,J) + A(I,J-1) + A(I,J+1) ) * 0.25;
```

The statement may represent either a serial or parallel implementation of Jacobi relaxation. In the parallel case the array data represented by `A` is distributed in some way across multiple processors and communication (to update the ghost boundary points along the edges of the partitioned data) is performed by the "=" operator. The equivalent (serial) C code is

```
for (int n=0; n < N; n++)  // Outer iteration
  {
   for (int j=1; j < SIZE_Y-1; j++)
      for (int i=1; i < SIZE_X-1; i++)
         a_new[j][i] = ( a[j][i-1] + a[j][i+1] + a[j-1][i] + a[j+1][i] ) * 0.25;

   for (int j=1; j < SIZE_Y-1; j++)
      for (int i=1; i < SIZE_X-1; i++)
         a[j][i] = a_new[j][i];
  }
```

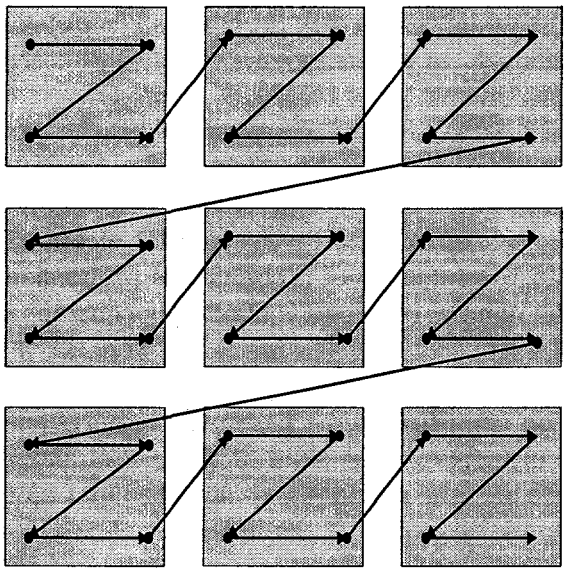

Figure 1: Compiler blocking, depicting the order of traversal. Only one iteration over each block is performed per iteration over the whole array data.

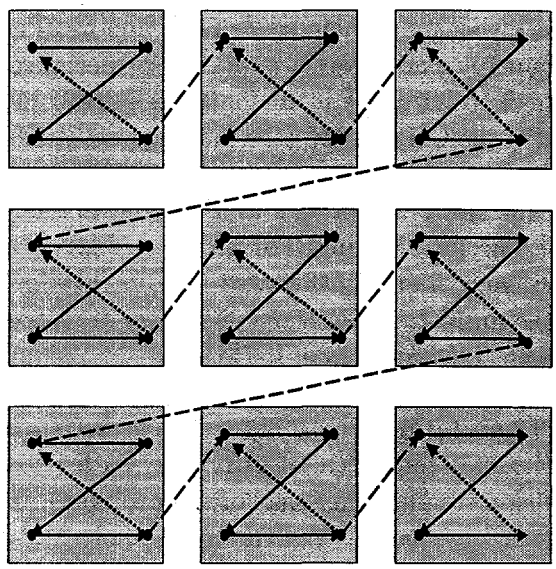

Figure 2: Temporal Blocking, depicting order of traversal, in which several interations over each block are performed while cache-resident.

3 Cache blocking

We distinguish two kinds of cache blocking: blocking done by a compiler (also called *tiling*) which we will refer to as *compiler blocking* (Figure 1), and our more effective technique (Figure 2). Blocking is also often introduced by hand, but it is usually not substantially different than that introduced by a compiler. For both, the context in which the transformation may be applied is in sweeping over an array, typically in a simple regular pattern of access visiting each element using a stencil operator. Such operations are a common part of numerical applications, including more sophisticated numerical algorithms (e.g. multigrid methods). What we describe is independent of any particular stencil operator, though the technique becomes more complex for higher-order operators because of increased stencil radius. Temporal blocking is also applicable to a loop over a sequence of statements.

On the basis of performance data and analysis of cache usage, we argue that the transformation is worthwhile, that a compiler could not reasonably be expected to perform such a transformation, but that the transformation is nonetheless practically automatable.

4 The Temporal Blocking Transformation

The problem with loops multiply traverse an array (as in the given code fragment) is that when the array is larger than the cache, the data cycles repeately through the cache. This is common in numerical applications. In the case of e.g. stencil operations, compilers can't do the kinds of optimizations we propose because of the dependence between outer iterations. A compiler may still perform blocking, but to a lesser effect.

Our transformation achieves greater temporal locality by performing multiple iterations on a block while it is in cache. Where a hierarchy of caches are present the data may fit in the L2 cache,

but we are particularly interesting in the cases where the L2 cache is similarly flushed and reloaded from either main memory or from non-local memory (as would be the case for DSM). The presence of multiple cache levels requires no special consideration, as, in our experience, optimization with respect to L1 cache is all that is required.

On the Origin 2000 (the machine for which we present data), the cost of accessing L1 cache is one clock cycle; 10 clock cycles for L2, 80 clock cycles for main memory, and for non-local memory 120 clock cycles plus network and cache-coherency overhead. These figures motivate the targetting the L1 cache for blocking. For problem sizes exceeding the size of L2 (usually the case for meaningful problems), a straightforward implementation gives rise to a number of cache misses proportional to the number of iterations; with our tranformation the number of misses is effectively constant for a moderate numbers of iterations.

For simplicity, we describe only one-dimensional blocking (or decomposition) of a two dimensional problem. The temporal blocking transformation comprises three parts and requires some additional storage (a *transitory array* for storing the elements on the sides of the blocks, one dimension smaller than that of the problem). These parts are: the special handling of the first block, the uniform handling of the interior blocks, and the special handling of the last block.

We describe the processing of the interior blocks and then comment on the special handling of the first and last blocks. The details of the transformation for the parallel case, which is more complex, is omitted. Choosing an appropriate block size is described later. The processing on each interior block is as follows:

1. Apply the stencil to each point on the edge of the cache block, but for the left edge of the stencil use the data stored previously in the transitory array. Note that the special handling of the first cache block initializes the transitory array.

2. Apply the unmodified stencil in the interior of the range of elements within the block over which the stencil is of the cache block (the rest of the cache block).[1]

3. Save the values on the edge of the cache block into the transitory array. These are used in the subsequent cache block's processing.

4. Copy the new solution to an `Old_Solution` array (this preserves the semantics of loop independence).

In the case of the processing of the first cache block the first step is not required, and in the case of the processing of the last cache block the third step is not required. Complete example code may be found at http://www.c3.lanl.gov/rose/transformations/pdcs98.html. The interaction of the stencil, array, and transitory array is depicted in Figure 12.

5 Performance Analysis

Before discussing experimental performance data we give an analysis of the problem from the point of view of performance. This is further elaborated with the presentation of the performance data.

As mentioned, memory latency is one of the primary performance bottlenecks in today's architectures, and blocking seeks to improve performance by hiding memory latency. The effectiveness of this technique depends on the amount of memory reuse. Reuse is categorized as *spatial* or *temporal*. Spatial reuse or spatial locality is present when a memory reference is related to some other

[1] For simplicity the precise effect of a particular stencil width is omitted.

memory references by a regular access pattern. In most scientific codes spatial locality is implicit in the design of the algorithm. The class of stencil computations represents a wide range of codes that have potential to exploit spatial locality. The pattern of access of memory is regular, and the references are to memory locations close together.

Temporal reuse or temporal locality is present when a memory reference to the same location occurs more than once during the execution. Again, the class of stencil computations has also potential for exploiting temporal locality in the form of memory-element reuse. In fact, across multiple stencil applications a subset of elements can be reused. In modern architectures the possibility of reuse allows a reduction of access to a lower level of the memory hierarchy. In particular, cache-based architectures have a small amount of memory that can be accessed at very high speed; the limitation is the size of that memory. Thus, the possibility of reusing data in cache is an opportunity for improving performance.

The objective of blocking, as presented in the previous section, is to exploit the potential for reuse present in a code. In scientific computing, which is the target of this work, the blocking performed by a compiler is capable of exploiting both spatial and temporal locality. From the previous sections should emerge the fact that the amount of locality, temporal locality in particular, can be significantly increased.

First we consider compiler blocking. It is possible to estimate the number of generated cache misses for a given code. For Jacobi relaxation there are two stages: the first iteration, and all subsequent iterations. In the first iteration the elements of the array are accessed in blocks (tiles) as depicted in Figure 1. Each time there is a transition between blocks the new block is loaded into cache, pushing out the previous block, so generating cache misses. Misses of this sort are called *compulsory* since they must occur for *any* pattern of access. Thus the number of misses for the first iteration is of the order of the problem size. There are also what are known as *conflict misses*, but we can assume that the conflict misses are a small fraction of the total, given that most of today's caches are associative, and that the block size can be chosen to allow free space in the cache to accomodate them. Once the first iteration is completed the process restarts and first block needs to be loaded again. The misses in the second stage can be classified as *capacity misses* since they are a result of the problem size exceeding the cache size. Thus for compiler blocking the number of misses is proportional to the problem size multiplied by the number of iterations. For the same reasons the actual execution time will be affected by the same two quantities, though the factor might be smaller than their product (the effective miss penalty can be reduced on some architectures).

The temporal blocking presented in this paper has been conceived to explicitly take advantage of the great potential of temporal locality in certain classes of codes, particularly stencil codes. The temporal blocking implementation of the Jacobi code is roughly depicted in Figure 2. Just as for compiler blocking there are two stages, with the first iteration having exactly the same behavior: a number of compulsory misses of the order of the problem size. With compiler blocking the multiple iterations are over the entire array, while with temporal blocking they are performed on a per block basis. Thus, once a block has been completed only a small fraction of that block needs to be used by the successive block. Most of the elements that are replaced in the transition between blocks are never needed again for the remainder of the computation. Temporal blocking decouples the capacity misses from the compulsory misses, making the first negligible and making the total number of misses largely independent of the number of iterations.

Unfortunately, temporal blocking does not yield an improvement in performance linear in the number of iterations, as might be hoped; there are some bounds that limit achievable speed-up: the number of iterations is limited by the block size which is directly dependent on the actual cache size. There are solutions to correctly handle cases where a larger block is needed, but they lead to

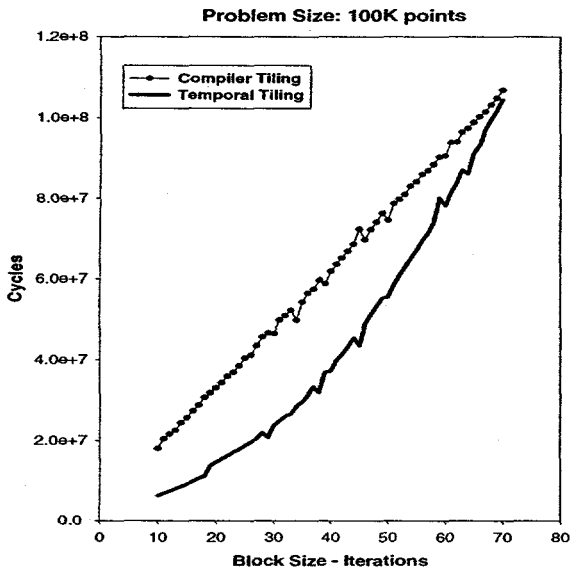

Figure 3: Caption goes here.

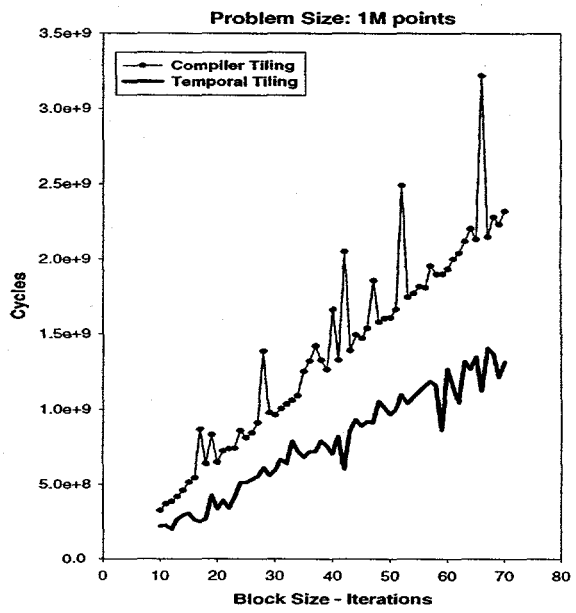

Figure 4: Caption goes here.

additional costs.

6 Performance Results

An instance of Jacobi relaxation is used to demonstratethe temporal tiling technique. Performance data were collected on the Silicon Graphics Inc. Origin 2000 which is based on the MIPS R10000 processor with 32K of primary data cache and 4M of secondary unified cache. The performance results were collected using on-processor hardware performance monitor counters; the programs were compiled at optimization level 3 using the MIPSpro C++ compiler.

Figures 3 and 4 contrast the performance of compiler blocking and temporal blocking in terms of CPU cycles. The block size as well as the number of Jacobi iterations varies on the x-axis. Figure 3 shows results for a problem size that fits in the secondary cache, while Figure 4 shows results for a problem size that does not.

Figure 3 shows that the temporal blocking version is twice as fast as the compiler blocking version until the block size exceeds the size of primary cache, beyond which temporal blocking and compiler blocking generate a similar number of cache misses. Figures 5 and 6 confirm our hypotheses: the graph describing the primary misses has the same proportion of the overall execution time. The miss count for the secondary cache shows that the problem fits in cache and the misses are only the compulsory ones. The spikes can be attributed to anomalies of the hardware counters that cannot capture results absolutely free of errors.

When the problem size exceeds the size of the secondary cache the improvement induced by temporal blocking is apparent at every point (eventually a similar scenario as the one seen for the smaller problem size if the block size exceedes the size of the secondary cache; however, having a block size larger than cache size is a poor choice for achieving good performance). The same factor of improvement in the cycle count appears in the count of primary and secondary misses, shown in Figures 7 and 8.

In all the experiments for which results are given in Figures 3 through 8 the block size is the

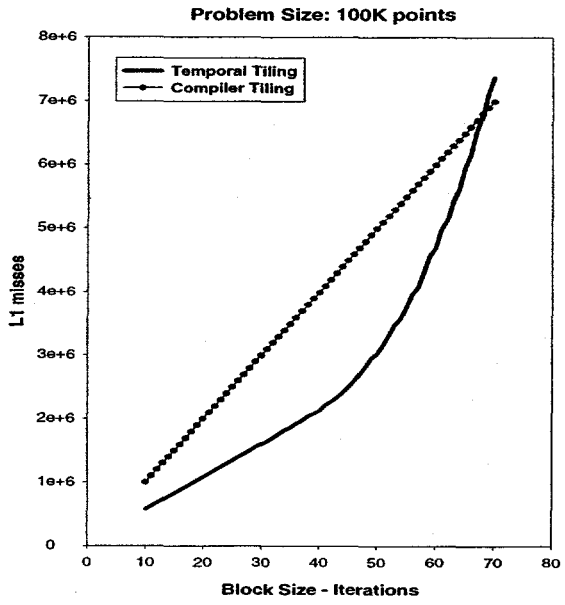

Figure 5: Caption goes here.

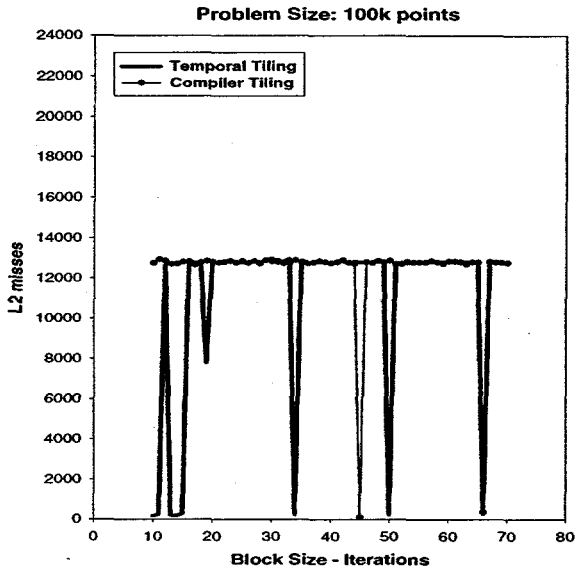

Figure 6: Caption goes here.

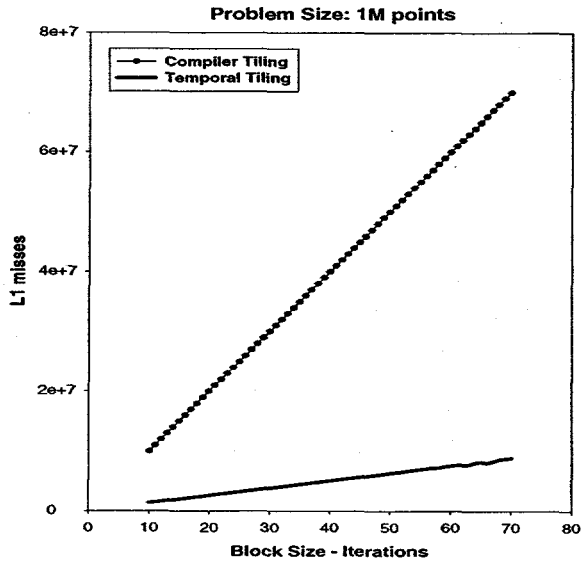

Figure 7: Caption goes here.

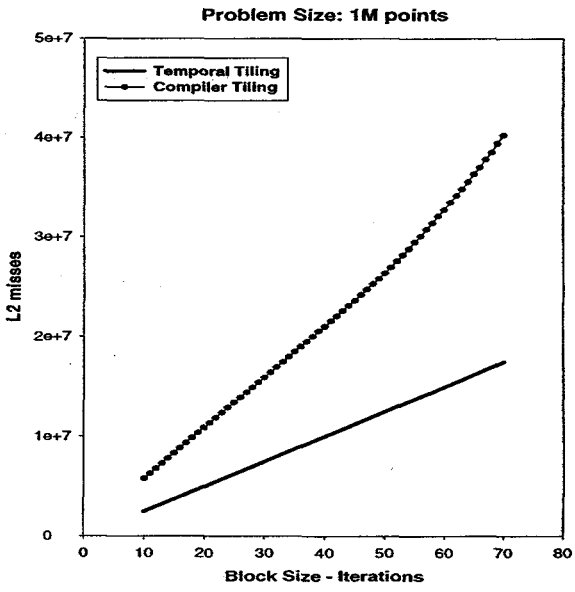

Figure 8: Caption goes here.

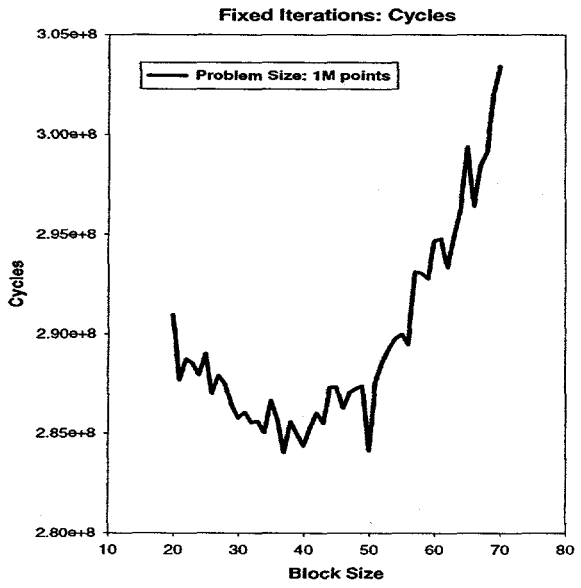

Figure 9: Caption goes here.

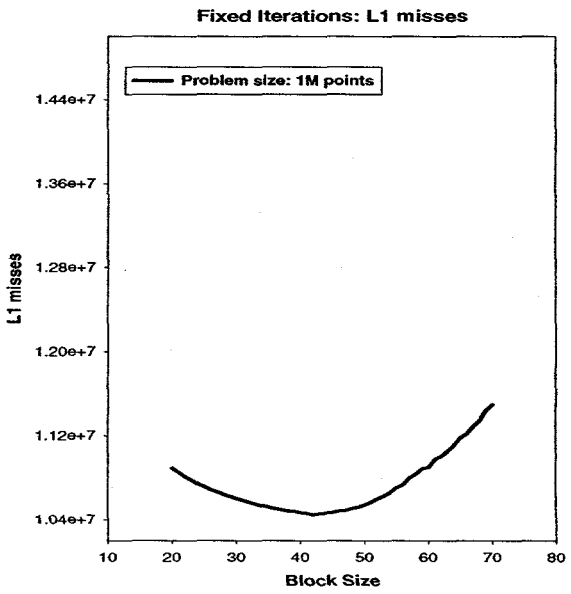

Figure 10: Caption goes here.

same as the number of iterations. In this implementation of temporal blocking the number of possible iterations is limited by the block size. However, in most cases the number of iterations is dictated by the numerical algorithm. The choice then becomes that of the best block size given a fixed number of iterations. For a Jacobi problem a good rule of thumb is to have the number of elements in the transitory array be a small fraction of the number of elements that could fit in a block that fits in primary cache. Figures 9 and 10 show results obtained by fixing the number of iterations and varying the block size for a problem size larger than the secondary cache. The first observation is that the number of primary cache misses determines the behaviour of the cycle count, shown in Figure 9. The second observation is that for greater than a characteristic block size the cache gets close to saturation and the number of conflict misses increases. The results suggest that the best performance is obtained when the number of iterations is half of the block size, which agrees with the given rule of thumb. Overall, however, variations in block size have little impact on performance so long as the block fits in cache.

It seems clear that blocking for primary cache gives better performance than blocking for secondary cache, and it appears that blocking for secondary cache comes as a consequence of blocking for primary cache. In Figure 11 blocking for primary cache and blocking for secondary cache are compared for a problem size that doesn't fit in secondary cache. This shows that blocking for secondary cache is significantly worse then blocking for primary cache, but still better then the compiler blocking. This can be explained with reference to cache misses. There is no difference in the number of misses for the secondary cache between blocking for primary cache and blocking for secondary cache, but the number of primary cache misses is significantly higher in the case of blocking for secondary cache.

7 Limits and Improvement of Temporal Blocking

As demonstrated, temporal blocking can give a factor of two improvement in performance over compiler blocking. What has not yet been shown is the dependency of performance improvement

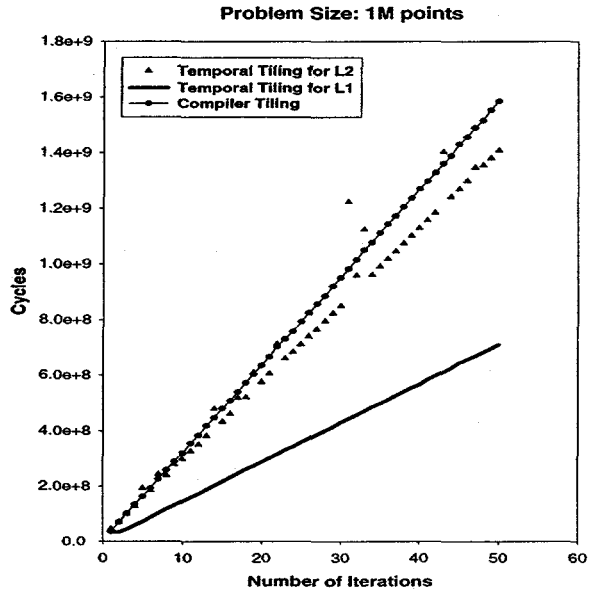

Figure 11: Caption goes here.

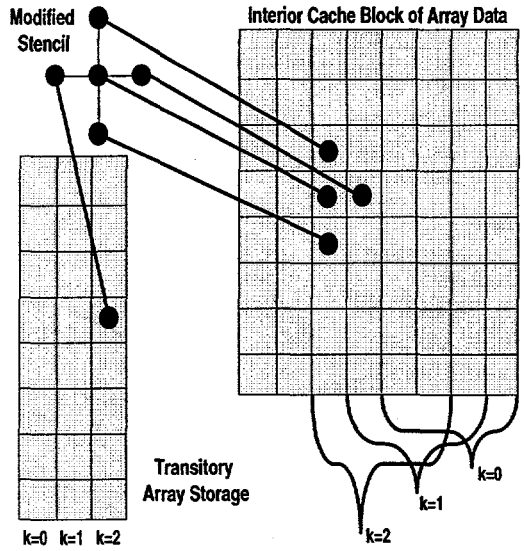

Figure 12: Caption goes here.

on the number of iterations for a fixed block size. The goal has been to reduce the number of capacity misses to a negligible value. Figures 13 and 14 show the result of varying the number of iterations for a fixed block size for a problem that doesn't fit in secondary cache. The figures show that achieved performance improvement is not as good as predicted—performance improves with the number of interations, but never exceeds a factor of two. At this stage the ideal performance is a crude estimate; in future this estimate will be improved. Figure 14 makes clear that the improvement factor is determined by the reduction in the number of misses. The figure shows that the achieved miss behavior is not relatively constant, but depends on the number of interations. The problem appears to be an artifact of the test code, which assumes that the transitory array it is always resident in cache, which is ideally correct. The assumption is based on an implementation that ensures that there is always enough space in cache for the transitory array and a subset of the other arrays. The test code currently ignores the fact that permanent residency in cache for the transitory array cannot be guaranteed just by ensuring that there is always enough cache space for all the subsets of the arrays. Different subsets of the other arrays would map to the same locations as does the transitory array, resulting in a significant increase in the number of conflict misses. Various solutions are being evaluated under the assumption that there is still room for improvement before reaching some physical architecture-dependent limitations.

8 Automating the Transformation

An optimizing transformation is generally only of academic interest if it is not deployed and used. In the context of array classes, it does not appear possible to provide this sort of optimization within the library because the applicability of the optimization is context dependent—the library can't know how its objects are being used. Two mechanisms for automating such optimizations are briefly discussed: the use of *expression templates*, which seems too limited; and a source-to-source tranformational system (a pre-processor), which is currently under development.

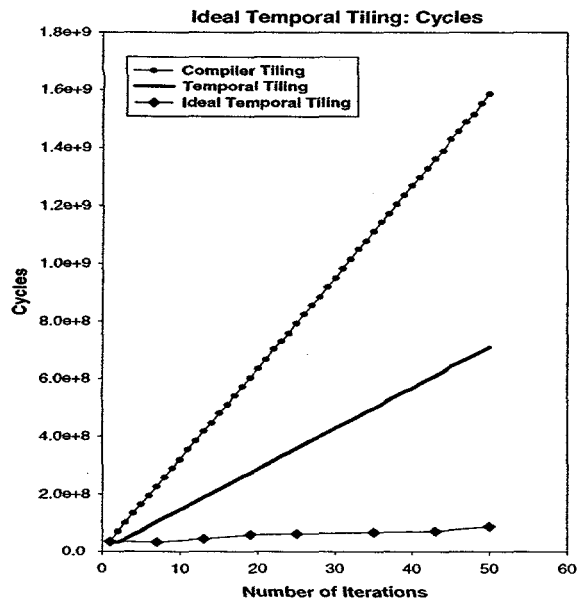

Figure 13: Caption goes here.

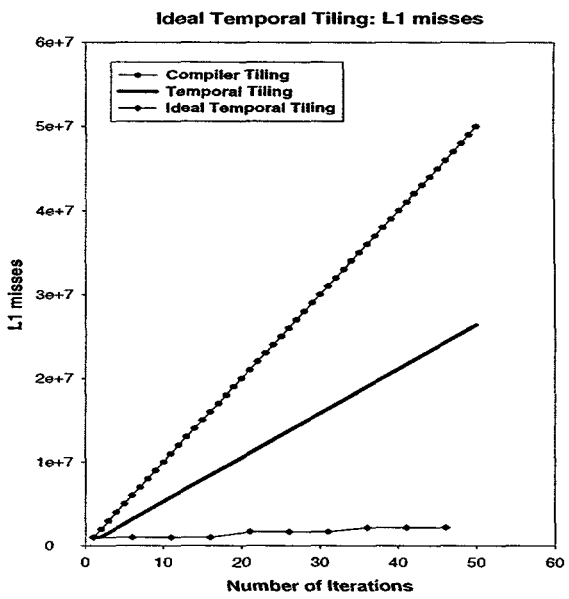

Figure 14: Caption goes here.

8.1 Expression Templates

The C++ class template mechanism has given rise to a programming idiom known as *expression templates*[9]. In essence the technique is sophisticated macro expansion, based on the C++ type system. In restricted scenarios it can be used to considerably improve performance of e.g. array operators defined as overloaded binary operators. As a methodology for improving performance it has two limitations: first, the nature of the expansion is such that it hides information required by a compiler to perform the optimizations that would be applied to equivalent 'hand written' code (this is described at length elsewhere [1]). Second, it is context-independent, so that, for example, there is no practicable means of detecting that an array statement is the body of a `for` loop.[2]

8.2 The ROSE Optimizing Preprocessor

The ROSE preprocessor is a mechanism for (C++) source-to-source transformation, specifically targetted at optimizing the use of statements manipulated array class objects. It is based on the Sage II C++ source code restructuring tools and provides a distinct (and optional) step in the compilation process. It recognizes the use of the A++/P++ array class objects, and is 'hard-wired' with (later parameterized by) the A++/P++ array class semantics, so obviating the need for difficult or impossible program analysis. There are in principle no limits on the types of transformations that can be performed using this mechanism.

Figure 15 shows the programming model associated with the development of an application using the language/compiler mechanism. It shows the addition of the framework/library as a tool to simplify the application and isolate and reuse code common to multiple applications. This figure also shows the use of the optimizing preprocessor as a mechanism to contain the semantics of the framework and process the application to perform transformations based on the context of how the framework's abstractions are used together within the application. Also shown is the

[2]The C++ class template mechanism has been shown to be Turing complete, however, using it for general-purpose programming is difficult and utterly impractical.

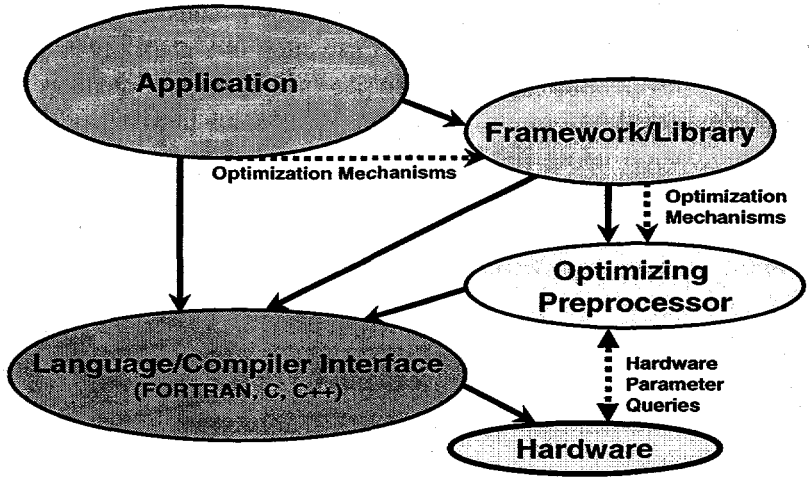

Figure 15: The OVERTURE/ROSE programming model.

communication of the optimizing preprocessor to obtain hardware specific information (cache size, communication latency data, etc.) so that the transformation can be made a highly tuned to the hardware as possible.

9 Conclusions

Previous work has focused on the optimization of the array class libraries themselves, and the use of techniques such as expression templates to provide better performance than the usual overloaded binary operators. We posit that such approaches are inadequate, that desirable optimizations exist that cannot be implemented by such methods, and cannot reasonably be expected to be implemented by a compiler. One such optimization for cache architectures has been detailed and demonstrated.

A significant part of the utility of this transformation is in its use to optimize array class statements (a particularly simple syntax for the user which hides the parallelism, distribution, and communication issues) and in the delivery of the transformation through the use of a preprocessing mechanism.

The specific transformation we introduce addresses the use of array statements or collections of array statements within loop structures, thus it is really a family of transformations. For simplicity, only the case of a single array in a single loop has been described. Specifically, we evaluate the case of a stencil operation in a `for` loop. We examine the performance using the C++ compiler, but generate only "C" code in the associated transformation. We demonstrate that the temporal blocking transform is two times faster than the standard implementation. It can be considered that this work introduces an attempt to get the best possible performance on a specific type of code fragment, albeit a common one in scientific computing.

The temporal blocking transformation is language independent, although we provide no mechanism to automate the transformation outside of the Overture object-oriented framework. Such

work could conceivably be used within FORTRAN 90 array syntax equally well.

Finally, the use of object-oriented frameworks is a powerful tool, but limited in use by the performance being less than that of FORTRAN 77, we expect that work such as this to change this situation. We expect that in the future one will use such object-oriented frameworks because they represent *both* a higher-level, simpler, and more productive way to develop large-scale applications *and* a higher performance development strategy. We expect higher performance because the representation of the application using the higher level abstractions permits the use of new tools (such as the ROSE optimizing preprocessor) that can introduce more sophisticated transformation (because of their more restricted semantics) than compilers could introduce (because of the broader semantics that the complete language represents).

10 Acknowledgements

Thanks are due to Madhav Marathe for suggesting this approach, which has been suggested in recent literature [5].

References

[1] Federico Bassetti, Kei Davis, and Dan Quinlan. A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks. In Ishikawa et al., editor, *International Scientific Computing in Object-Oriented Parallel Environments, ISCOPE 97*, volume 1343 of *LNCS*. Springer, 1997.

[2] Federico Bassetti, Kei Davis, and Dan Quinlan. Toward fortran 77 performance from object-oriented scientific frameworks. In *Proceedings of the High Performance Computing Conference (HPC'98)*, 1998.

[3] David Brown, Geoff Chesshire, William Henshaw, and Dan Quinlan. Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments. In *Proceedings of the SIAM Parallel Conference*, Minneapolis, MN, March 1997.

[4] Gnu scientific software library. http:// KachinaTech.com.

[5] Charles E. Leiserson, Satish Rao, and Sivan Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *Journal of Computer and System Sciences*, 1997.

[6] Roldan Pozo. Template numerical toolkit. http:// math.nist.gov/ tnt/.

[7] Dan Quinlan and Rebecca Parsons. A++/p++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.

[8] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3 edition, 1997.

[9] Todd Veldhuizen. Expression templates. Technical Report 5, C++ Report 7, June 1995.