Run-time code generation in C++ as a foundation for domain-specific optimisation

Olav Beckmann, Alastair Houghton, Paul H J Kelly, and Michael Mellor

Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, United Kingdom
{o.beckmann,p.kelly}@imperial.ac.uk
www.doc.ic.ac.uk/{~ob3,~phjk}

Abstract. The TaskGraph Library is a C++ library for dynamic code generation, which combines specialisation with dependence analysis and loop restructuring. A TaskGraph represents a fragment of code which is constructed and manipulated at run-time, then compiled, dynamically linked and executed. TaskGraphs are initialised using macros and overloading, which forms a simplified, C-like sub-language with first-class arrays and no pointer arithmetic. Once a TaskGraph has been constructed, we can analyse its dependence structure and perform optimisations. In this paper, we present the design of the TaskGraph library, and two sample applications to demonstrate its use for runtime code specialisation and restructuring optimisation.

1 Introduction

Setting the Scene: Cross-Component Optimisation at Runtime. The work we describe in this paper is part of a wider research programme at Imperial College aimed at addressing the apparent conflict between the quality of scientific software and its performance. High-quality, easy-to-maintain scientific software is often built from abstract components which have been independently verified and optimised. Unfortunately, there is a performance penalty associated with this approach since components are deployed outside the context in which they have been optimised. Our proposal for reversing this performance penalty is based on runtime cross-component optimisation. Current research projects which implement this general approach are a library for performing runtime cross-component data placement optimisation in data-parallel programs [13], a system for optimising Java RMI calls at runtime [22] and runtime cross-component loop fusion [6].

The TaskGraph Library. The TaskGraph library is a key tool which we are developing in order to drive this research programme. The library is written in C++ and is designed to support dynamic code generation, specialisation and explicit analysis and manipulation of the generated code:

– *Dynamic Component Specialisation*
  The TaskGraph library can be used for specialising software components

according to either their parameters or other runtime context information. Later in this paper (Section 3), we show an example of specialising a generic image filtering function to the particular convolution matrix being used.
- *Runtime Dependence Analysis and Restructuring*
  The TaskGraph library uses SUIF-1 [21], the Stanford University Intermediate Format, as its internal representation for code. This makes a rich collection of dependence analysis and restructuring passes available for our use in code optimisation.
- *Runtime Generation of Component Metadata*
  Our delayed evaluation, self-optimising (DESO) library [13] for performing runtime cross-component data placement optimisation currently relies on hand-written metadata which characterises the data placement constraints of each component. We have carried out initial work aimed at generating this metadata automatically using the TaskGraph library [19].

Relationship with Earlier Work. Several earlier tools for dynamic code optimisation have been reported in the literature [5, 8]. The key characteristics which distinguish our approach are as follows:

- *Single-Language Design*
  The TaskGraph library is implemented in C++ and any TaskGraph program can be compiled as C++ using widely-available compilers. This is in contrast with approaches such as 'C [5] which rely on a special compiler for processing dynamic constructs. The TaskGraph library's support for manipulating code as data within one language was pioneered in Lisp [15].
- *Explicit Specification of Dynamic Code*
  Like 'C [5], the TaskGraph library is an imperative system in which the application programmer has to construct the code as an explicit data structure. This is in contrast with ambitious partial evaluation approaches such as DyC [8,9] which use declarative annotations of regular code to specify where specialisation should occur and which variables can be assumed constant. Offline partial evaluation systems like these rely on *binding-time analysis* (BTA) to find other, derived static variables [12].
- *Simplified C-like Sub-language*
  Dynamic code is specified with the TaskGraph library via a small sub-language which is very similar to standard C (see Section 2). This language has been implemented through extensive use of macros and C++ operator overloading and consists of a small number of special control flow constructs, as well as special types for dynamically bound variables. This means that BTA in our approach is effectively performed by the C++ type system. The language has first-class arrays, unlike C and C++, to facilitate dependence analysis.

Structure of this Paper. In Section 2, we describe how TaskGraphs are constructed. Section 3 offers a simple demonstration of run-time specialisation. Section 4 explains how the library itself is implemented. In Section 5, we use matrix

```
#include <stdio.h>
#include <TaskGraph>

using namespace tg;

int main( int argc, char *argv[] ) {
  TaskGraph T;
  int b = 1, c = 1;

  taskgraph( T ) {
    tParameter ( tVar ( int, a ) );

    a = a + c;
  }

  T.compile();
  T.execute( "a", &b, NULL );

  printf ( "b␣=␣%d\n", b );
  return 0;
}
```

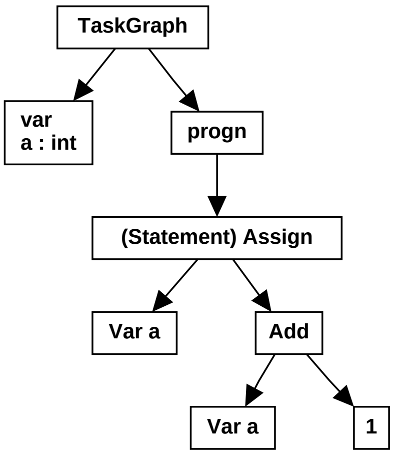


Fig. 1. *Left:* Simple Example of using the TaskGraph library. *Right:* Abstract syntax tree (AST) for the simple TaskGraph constructed by the piece of code shown on the left. The variable `c`, which has is static at TaskGraph construction time appears in the AST as a value (see Section 4).

multiplication to illustrate the use of the library's loop restructuring capabilities. In Sections 6 and 7 we discuss related and ongoing work, and Section 8 concludes.

2 The TaskGraph Library API

A TaskGraph is a data structure which holds the abstract syntax tree (AST) for a piece of dynamic code. A key feature of our approach is that the application programmer has access to and can manipulate this data structure at runtime; in particular, we provide an extensible API (sub-language) for *constructing* TaskGraphs at runtime. This API was carefully designed using macros and C++ operator overloading to look as much as possible like ordinary C.

A Simple Example. The simple C++ program shown in the left-hand part of Figure 1 is a complete example of using the TaskGraph library. When compiled with `g++`, linked against the TaskGraph library and executed, this program dynamically creates a piece of code for the statement `a = a + c`, binds the application program variable `b` as a parameter and executes the code, printing `b = 2` as the result. This very simple example illustrates both that creation of dynamic code is completely explicit in our approach and that the language for creating the AST which a TaskGraph holds looks similar to ordinary C.

```
void convolution( const int IMGSZ, const FLOAT *image, FLOAT *new_image,
                  const int CSZ /* convolution matrix size */, const FLOAT *matrix ) {
  int i, j, ci, cj;
  assert( CSZ % 2 == 1 );
  const int c_half = ( CSZ / 2 );

  // Loop iterating over image
  for( i = c_half; i < IMGSZ - c_half; ++i ) {
    for( j = c_half; j < IMGSZ - c_half; ++j ) {
      new_image[i * IMGSZ + j] = 0.0;

      // Loop to apply convolution matrix
      for( ci = - c_half; ci <= c_half; ++ci ) {
        for( cj = - c_half; cj <= c_half; ++cj ) {
          new_image[i * IMGSZ + j] +=
            image[(i+ci) * IMGSZ + j+cj] * matrix[(c_half+ci) * CSZ + c_half+cj];
        }
      }
    }
  }
  return;
}
```

Fig. 2. Generic image filtering: C++ code. Because the size as well as the entries of the convolution matrix are runtime parameters, the inner loops (for-ci and for-cj), with most likely very low trip-count, cannot be unrolled efficiently.

3 Generalised Image Filtering

We now show an example which uses a fuller range of TaskGraph constructs and which also demonstrates a real performance benefit from runtime code optimisation. A generic image convolution function, which allows the application programmer to supply an arbitrary convolution matrix could be written in C as shown in Figure 2. This function has the advantage of genericity (the interface is in principle similar to the General Linear Filter functions from the Intel Performance Libraries [11, Section 9]) but suffers from poor performance because

– The loop bounds of the inner loops over the convolution matrix are statically unknown, hence these loops, with most likely very low trip-count, cannot be unrolled efficiently.
– Failure to unroll the inner loops not only leads to unnecessarily complicated control flow, but also blocks optimisations such as vectorisation on the outer loops.

Figure 3 shows a function which constructs a TaskGraph that is specialised to the particular convolution matrix being used. The `tFor` constructs are part of the TaskGraph API and create a loop node in the AST. Note, however, that the inner `for` loops are executed as ordinary C++ at TaskGraph *construction* time, creating an assignment node in the AST for each iteration of the loop body. The effect is that the AST contains control flow nodes for the for-i and for-j loops and a loop body consisting of `CSZ * CSZ` assignment statements.

We study the performance of this example in Figure 4. The convolution matrix used was a 3×3 averaging filter, images were square arrays of single-

```
void taskgraph_convolution( TaskGraph &T, const int IMGSZ,
                            const int CSZ, const FLOAT *matrix ) {
  int ci, cj;
  assert ( CSZ % 2 == 1 );
  const int c_half = ( CSZ / 2 );

  taskgraph( T ) {
    unsigned int dims[] = {IMGSZ * IMGSZ};
    tParameter( tArray( FLOAT, tgimg, 1, dims ) );
    tParameter( tArray( FLOAT, new_tgimg, 1, dims ) );
    tVar ( int, i );
    tVar ( int, j );

    // Loop iterating over image
    tFor( i, c_half, IMGSZ - (c_half + 1) ) {
      tFor( j, c_half, IMGSZ - (c_half + 1) ) {
        new_tgimg[i * IMGSZ + j] = 0.0;

        // Loop to apply convolution matrix
        for( ci = -c_half; ci <= c_half; ++ci ) {
          for( cj = -c_half; cj <= c_half; ++cj) {
            new_tgimg[i * IMGSZ + j] +=
              tgimg[(i+ci) * IMGSZ + j+cj] * matrix[(c_half+ci) * CSZ + c_half+cj];

          }
        }
      }
    }
  }
  return;
}
```

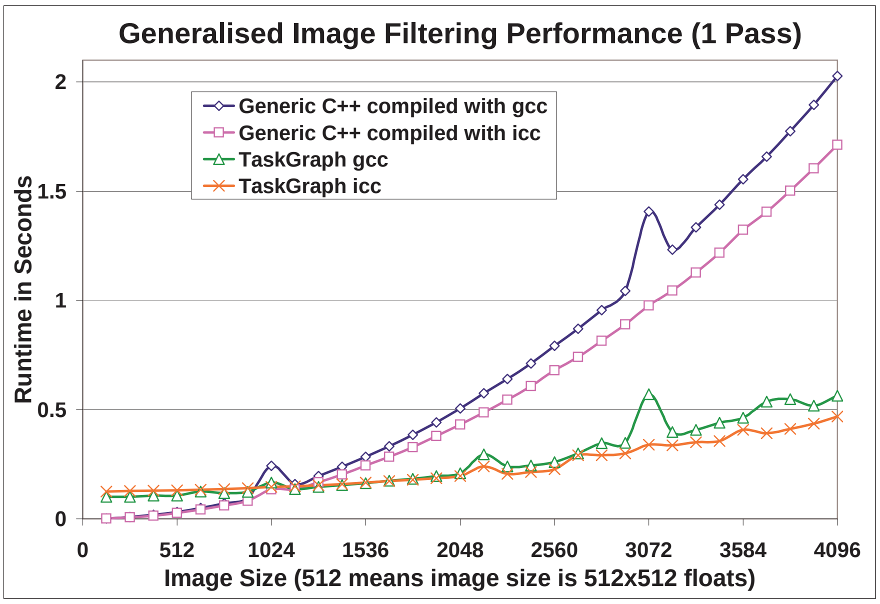
Fig. 3. Generic image filtering: function constructing the TaskGraph for a specific convolution matrix. The size as well as the entries of the convolution matrix are static at TaskGraph construction time. This facilitates complete unrolling of the inner two loops. The outer loops (for-i and for-j) are entered as control flow nodes in the AST.

precision floats ranging in size up to 4094×4096. Measurements are taken on a Pentium 4-M with 512KB L2 cache running Linux 2.4, `gcc` 3.3 and the Intel C++ compiler version 7.1. We compare the performance of the following:

– The static C++ code, compiled with `gcc` 3.3.
– The static C++ code, compiled with the Intel C++ compiler version 7.1. The `icc` compiler reports that the innermost loop (for-cj) has been vectorised[1]. Note, however, that this loop will have a dynamically determined trip-count of 3, *i.e.* the Pentium 4's 16-byte vector registers will not be filled.
– The code dynamically generated by the TaskGraph library, compiled with `gcc` 3.3. The two innermost loops are unrolled.
– The code dynamically generated by the TaskGraph library, compiled with `icc` 7.1. The two innermost loops are unrolled and the then-remaining innermost loop (the for-j loop over the image) is vectorised by `icc`.

We have deliberately measured the performance of these image filtering functions for only one pass over an image. In order to see a real speedup the overhead of

[1] The SSE2 extensions implemented on Xeon and Pentium 4 processors include 16-byte vector registers and corresponding vector instructions [10].



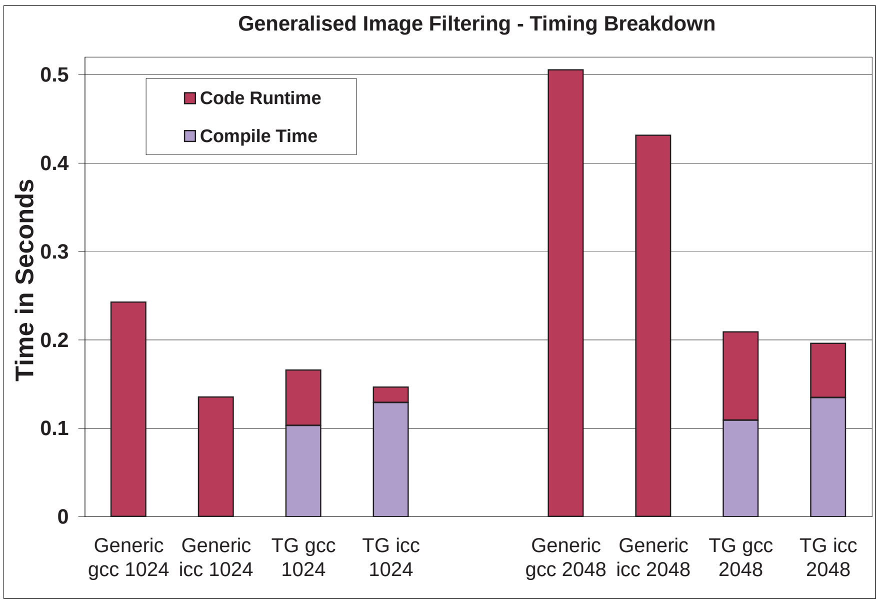


Fig. 4. Performance of image filtering example. Top: Total execution time, including runtime compilation, for one pass over image. Bottom: Breakdown of total execution time into compilation time and execution time of the actual convolution code for two specific image sizes: 1024 × 1024 (the break-even point) and 2048 × 2048.

runtime compilation therefore needs to be recovered in just a single application of the generated code. Figure 4 shows that we do indeed get an overall speedup for image sizes that are greater than 1024×1024. In the right-hand part of Figure 4, we show a breakdown of the overall execution time for two specific data sizes. This demonstrates that although we achieve a huge reduction in execution time of the actual image filtering code, the constant overhead of runtime compilation cancels out this benefit for a data size of 1024×1024. However, for larger data sizes, we achieve an overall speedup.

Note, also, that image filters such as the one in this example might be applied either more than once to the same image or to different images — in either case, we would have to pay the runtime compilation overhead only once and will get higher overall speedups.

4 How it Works

Thus far, we have given examples of how the TaskGraph library is used, as well as demonstrated that it can achieve significant performance gains. In this section we now give a brief overview of TaskGraph syntax, together with an explanation of how the library works.

TaskGraph Creation. The TaskGraph library can represent code as data — specifically, it provides TaskGraphs as data structures holding the AST for a piece of code. We can create, compile and execute different TaskGraphs independently. Statements such as the assignment `a = a + c` in line 13 of Figure 1 make use of C++ operator overloading to add nodes (in this case an assignment statement) to a TaskGraph. Figure 1 illustrates this by showing a graphical representation of the complete AST which was created by the adjacent code. Note that the variable `c` has *static* binding-time for this TaskGraph. Consequently, the AST contains its value rather than a variable reference.

The `taskgraph( T ){...}` construct (see line 308 in Figure 3) determines which AST the statements in a block are attached to. This is necessary in order to facilitate independent construction of different TaskGraphs, representing different computations.

Variables in TaskGraphs. The TaskGraph library inherits lexical scoping from C++. The `tVar(type, name)` construct (see lines 312 and 313 in Figure 3) can be used to declare a dynamic local variable. Similarly, the `tArray(type, name, no_dims, extents[])` construct can be used to declare a dynamic multidimensional array with number of dimensions `no_dims` and size in each dimension contained in the integer array `extents`. Arrays are first-class objects in the TaskGraph construction sub-language and can only be accessed inside a TaskGraph using the `[]` subscript operators. There are no pointers in the TaskGraph construction sub-language.

```
302 void taskgraph_convolution( TaskGraph &T, const int IMGSZ,
303                             const int CSZ, const FLOAT *matrix ) {
304   int ci , cj ;
305   assert ( CSZ % 2 == 1 );
306   const int c_half = ( CSZ / 2 );
307
308   taskgraph( T ) {
309     unsigned int dims[] = {IMGSZ * IMGSZ};
310     tParameter( tArray( FLOAT, tgimg, 1, dims ) );
311     tParameter( tArray( FLOAT, new_tgimg, 1, dims ) );
312     tVar ( int, i );
313     tVar ( int, j );
314
315     // Loop iterating over image
316     tFor( i, c_half, IMGSZ − (c_half + 1) ) {
317       tFor( j, c_half, IMGSZ − (c_half + 1) ) {
318         new_tgimg[ i * IMGSZ + j ] = 0.0;
319
320         // Loop to apply convolution matrix
321         for( ci = −c_half; ci <= c_half; ++ci ) {
322           for( cj = −c_half; cj <= c_half; ++cj) {
323             new_tgimg[ i * IMGSZ + j ] +=
324               tgimg[( i +ci) * IMGSZ + j +cj] * matrix[(c_half+ci) * CSZ + c_half+cj];
325           }
326         }
327       }
328     }
329   }
330   return;
331 }
```

Fig. 5. Binding-Time Analysis. TaskGraph construction code for the image filtering example from Figure 2, with all dynamic variables marked by a boxed $\boxed{\text{outline}}$.

TaskGraph Parameters. Both Figure 1 (line 11) and Figure 3 (lines 310 and 311) illustrate that any TaskGraph variable can be declared to be a TaskGraph parameter using the `tParameter()` construct. We require the application programmer to ensure that TaskGraph parameters bound at execution time do not alias each other.

Control Flow Nodes. Inside a TaskGraph construction block, `for` loops and `if` conditionals are executed at construction time. Therefore, the `for` loops on lines 321 and 322 in Figure 3 result in an unrolled inner loop. However, the TaskGraph sub-language defines some constructs for adding control-flow nodes to an AST: `tFor(var,lower,upper)` adds a loop node (see lines 316 and 317 in Figure 3). The loop bounds are *inclusive*. `tIf()` can be used to add a conditional node to the AST.

Expressions and Binding-Time Analysis. We refer to variables that are bound at TaskGraph construction time as *static* variables and those that are bound at execution time as *dynamic*. Declarative code specialisation systems such as DyC [8] use annotations that declare some variables to be static for the purpose of partial evaluation. In contrast, static binding time, *i.e.* evaluated at TaskGraph construction time is the default for the TaskGraph language. Only TaskGraph variables, including parameters, are dynamic. Internally, dynamic variables are represented by special types and the overloaded operators defined on those dynamic types define binding-time derivation rules. Thus, an expression such as `a + c` in Figure 1 where `a` is dynamic and `c` is static is derived dynamic, but the static part is evaluated at construction time and entered into the AST as a value. We illustrate this by reproducing the TaskGraph image filtering code from Figure 3 again in Figure 5; however, this time all dynamic expressions are marked by a boxed outline. Note that the convolution matrix, including its entire subscript expression in the statement on line 324, is *static*.

5 Another example: matrix multiply

In Section 3, we showed an example of how the specialisation functionality of the TaskGraph library can be used to facilitate code optimisations such as vectorisation. In this Section, we show, using matrix multiplication as an example, how we can take advantage of the use of SUIF-1 as the underlying code representation in the TaskGraph library to perform restructuring optimisations at runtime.

Figure 6 shows both the code for the standard C/C++ matrix multiply loop (ijk loop order) and the code for constructing a TaskGraph representing this loop, together with an example of how we can direct optimisations from the application program: we can interchange the for-j and for-k loops before compiling and executing the code. Further, we can perform loop tiling with a runtime-selected tile size. This last application demonstrates in particular the possibilities of using the TaskGraph library for domain-specific optimisation:

– *Optimising for a particular architecture*
  In Figure 6, we show a simple piece of code which implements a runtime search for the optimal tilesize when tiling matrix multiply. In Figure 8, we show the results of this search for both a Pentium 4-M (with 512K L2 cache) and an Athlon (with 256K L2 cache) processor. The resulting optimal tilesizes differ for most problem sizes, but they do not differ by as much as would have been expected if the optimal tilesize was based on L2 capacity. We assume that a different parameter, such as TLB span, is more significant in practice.
– *Optimising for a particular loop or working set*
  We note that the optimal tile size for matrix multiply calculated by our code shown in Figure 6 differs across problem sizes (see Figure 8). Similarly, we would expect the optimal tilesize to vary for different loop bodies and resulting working sets.

We believe that high performance achieved, with relatively straight-forward code, in our matrix multiply example (up to 2 GFLOP/s on a Pentium 4-M

```
/*
 * mm_ijk
 * Most straight-forward matrix multiply
 * Calculates C += A * B
 */
void mm_ijk( const unsigned int sz,
             const FLOAT *const A,
             const FLOAT *const B,
             FLOAT *const C ) {
  unsigned int i, j , k;

  for( i = 0; i < sz; ++i ) {
    for( j = 0; j < sz; ++j ) {
      for( k = 0; k < sz; ++k ) {
        C[i*sz+j] += A[i*sz+k] * B[k*sz+j];
      }
    }
  }
  return;
}
```

```
void TG_mm_ijk( unsigned int sz[2],
                TaskLoopIdentifier *loop,
                TaskGraph &t ) {
  taskgraph( t ) {
    tParameter(tArray(FLOAT, A, 2, sz));
    tParameter(tArray(FLOAT, B, 2, sz));
    tParameter(tArray(FLOAT, C, 2, sz));
    tVar( int, i );
    tVar( int, j );
    tVar( int, k );

    tGetId( loop [0] );
    tFor( i , 0, sz [0] - 1 ) {
      tGetId( loop [1] );
      tFor( j , 0, sz [1] - 1 ) {
        tGetId( loop [2] );
        tFor( k, 0, sz [0] - 1 ) {
          C[i ][ j] += A[i][k] * B[k][ j ];
        }
      }
    }
  }
}
```

```
for( int tsz = 4; tsz <= min(362, matsz); ++tsz ) {
  unsigned int sizes[] = { matsz, matsz };
  int trip3 [] = { tsz , tsz , tsz };
  TaskLoopIdentifier loop [3];
  TaskGraph MM;

  TG_mm_ijk( sizes, loop, MM );
  interchangeLoops( loop [1], loop [2] );  // Interchange loops
  tileLoop( 3, &loop [0], trip3 );         // Tile inner two loops
  MM.compile( TaskGraph::ICC, false );

  tt2 = time_function ();
  MM.setParameters( "A", A, "B", B, "C", C, NULL );
  MM.execute();
  tt2 = time_function() - tt2;

  time [0]   = time_to_seconds( tt2 );
  if( time[0] < best_time_icc ) {
    best_time_icc = time [0];
    best_tsz_icc  = tsz;
  }
}
```

Fig. 6. The code on the top left is the standard C++ matrix multiply (ijk loop order) code. The code on the top right constructs a TaskGraph for the standard ijk matrix multiply loop. The code underneath shows an example of using the TaskGraph representation for the ijk matrix multiply kernel, together with SUIF-1 passes for interchanging and tiling loops to search for the optimal tilesize of the interchanged and tiled kernel for a particular architecture and problem size.

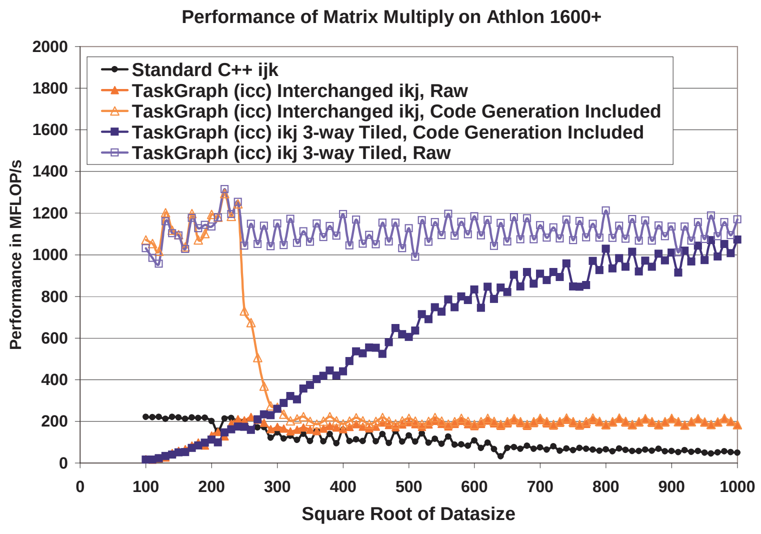
1.8 GHz) shows promising potential for our approach of performing dynamic specialisation and optimisation, based on runtime domain-specific information.

6 Related Work

In this section, we briefly discuss related work in the field of dynamic code optimisation.

Language-Based Approaches.

- *Imperative*
  Tick-C or 'C [5], a superset of ANSI C, is a language for dynamic code generation. Like the TaskGraph library, 'C is explicit and imperative in nature; however, a key difference in the underlying design is that 'C relies on a special compiler (`tcc`). Dynamic code can be specified, composed and instantiated, *i.e.* compiled, at runtime. The fact that ‘C relies on a special compiler also means that it is in some ways a more expressive and more powerful system than the TaskGraph library. For example, 'C facilitates the construction of dynamic function calls where the type and number of parameters is dynamically determined. This is not possible in the TaskGraph library. Jak [2], MetaML [20], MetaOCaml [4] and Template Haskell [18] are similar efforts, all relying on changes to the host language's syntax.
- *Declarative*
  DyC [8,9] is a dynamic compilation system which specialised selected parts of programs at runtime based on runtime information, such as values of certain data structures. DyC relies on declarative user annotations to trigger specialisation. This means that a sophisticated binding-time analysis is required which is both polyvariant (*i.e.* allowing specialisation of one piece of code for different combinations of static and dynamic variables) and program-point specific (*i.e.* allowing polyvariant specialisation to occur at arbitrary program points). The result of BTA is a set of *derived* static variables in addition to those variables which have been annotated as static. In order to reduce runtime compilation time, DyC produces, at compile-time, a *generating extension* [12] for each specialisation point. This is effectively a dedicated compiler which has been specialised to compile only the code which is being dynamically optimised. This static pre-planning of dynamic optimisation is referred as *staging*.
  Marlet *et al* [14] present a proposal for making the specialisation process itself more efficient. This is built using Tempo [3], an offline partial evaluator for C programs and also relies on an earlier proposal by Glück and Jørgensen to extend two-level binding-time analysis to multiple levels [7], *i.e.* to distinguish not just between dynamic and static variables but between multiple stages. The main contribution of Marlet *et al* is to show that multi-level specialisation can be achieved more efficiently by repeated, incremental application of a two-level specialiser.



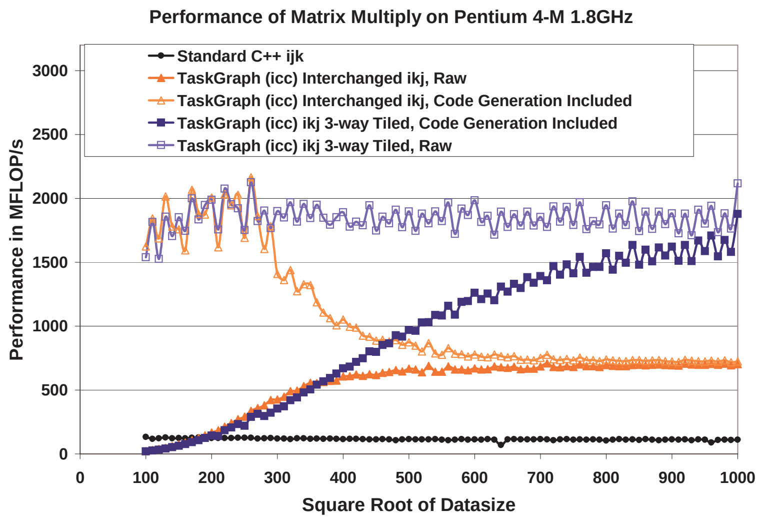


Fig. 7. Performance of matrix multiply on Athlon 1600+ with 256KB L2 cache and on Pentium 4-M 1.8 GHz with 512KB L2 cache. We show the performance of the naive C++ code (ijk loop order), the code where the we have used the TaskGraph library to interchange the inner two loops (resulting in ikj loop order) and the code where the TaskGraph library is used to interchange and 3-way tile the loops. For the tiled code, we used the TaskGraph library to search for the optimal tile size for each data point, as shown in Figure 6. For both the interchanged and tiled code, we plot one graph showing the raw performance of the generated code and one graph which shows the performance after the dynamic code generation cost has been amortised over one invocation of the generated code.

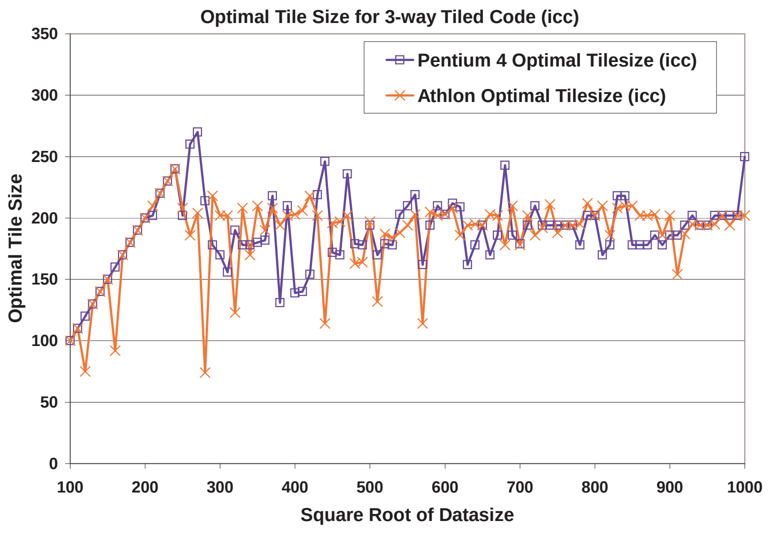


Fig. 8. Optimal tile size on Athlon and Pentium 4-M processors, for each data point from Figure 7. These results are based on a straight-forward exhaustive search implemented using the TaskGraph library's runtime code restructuring capabilities (see code in Figure 6).

Data-Flow Analysis. Our library performs runtime data flow analysis on loops operating on arrays. A possible drawback with this solution could be high runtime overheads. Sharma *et al* present deferred data-flow analysis (DDFA) [17] as a possible way of combining compile-time information with only limited runtime analysis in order to get accurate results. This technique relies on comprising the data flow information from *regions* of the control-flow graph into *summary functions*, together with a runtime *stitcher* which selects the applicable summary function, as well as computes summary function compositions at runtime.

Transparent Dynamic Optimisation of Binaries. One category of work on dynamic optimisation which contrasts with ours are approaches which do not rely on program source code but instead work in a transparent manner on running binaries.

Dynamo [1] is a transparent dynamic optimisation system, implemented purely in software, which works on an executing stream of native instructions. Dynamo interprets the instruction stream until a *hot trace* of instructions is identified. This is then optimised, placed into a code cache and executed when the starting-point is re-encountered.

These techniques also perform runtime code optimisation; however, as stated in Section 1, our objective is different: restructuring cross-component optimisation at runtime.

7 Ongoing and Future Work

We have recently evaluated the current TaskGraph library implementation in the context of some moderately large research projects [6]. This experience has led us to planning future developments of this work.

- *Automatic Generation of OpenMP Annotations*
  We would like to use the runtime dependence information which is calculated by the TaskGraph library for automatically annotating the generated code with OpenMP [16] directives for SMP parallelisation. An alternative approach would be to use a compiler for compiling the generated code that has built-in SMP parallelisation capabilities.
- *Automatic Derivation of Component Metadata*
  Our delayed evaluation, self-optimising (DESO) library of data-parallel numerical routines [13] currently relies on hand-written metadata which characterise the data placement constraints of components to perform cross-component data placement optimisation. One of the outstanding challenges which we would like to address in this work is to allow application programmers to write their own data-parallel components *without* having to understand and supply the placement-constraint metadata. We hope to generate these metadata automatically with the help of the TaskGraph library's dependence information. Some initial work on this project has been done [19].
- *Transparent Cross-Component Loop Fusion*
  In an ongoing project [6] we are using the TaskGraph library to perform cross-component loop fusion in our DESO library of data-parallel numerical routines.

8 Conclusion

The TaskGraph library combines code specialisation with runtime dependence analysis and restructuring optimisations. We believe that this combination is unique, and essential for our research agenda of restructuring cross-component optimisation, carried out at runtime with the benefit of runtime context information. Since our long-term objectives include the optimisation of large scientific codes, we decided on the exclusive use of standard C++ to facilitate integrating the TaskGraph library with existing codes.

Acknowledgements. This work was supported by the United Kingdom EPSRC-funded OSCAR project (GR/R21486).

References

1. Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00: Programming Language Design and Implementation*, pages 1–12, 2000.

2. Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer Society Press, 1998.
3. C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3), September 1998.
4. Walid Taha *et al.* MetaOCaml homepage. www.cs.rice.edu/ taha/MetaOCaml/.
5. Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: a language for high-level, efficient, and machine-independent dynamic code generation. In *POPL '96: Principles of Programming Languages*, pages 131–144, 1996.
6. Peter Fordham. Transparent run-time cross-component loop fusion. MEng Thesis, Department of Computing, Imperial College London, June 2002.
7. Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In *Perspectives of System Informatics*, number 1181 in LNCS, pages 261–272. Springer-Verlag, 1996.
8. Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, October 2000.
9. Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *PLDI '99: Programming Language Design and Implementation*, pages 293–304, 1999.
10. Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 1999–2002. Available via `developer.intel.com`.
11. Intel Corporation. *Integrated Performance Primitives for Intel Architecture. Reference Manual. Volume 2: Image and Video Processing*, 200–2001.
12. Neil D. Jones. Mix Ten Years Later. In *PEPM '95: Partial Evaluation and Semantics-Based Program Manipulation*, 1995.
13. Peter Liniker, Olav Beckmann, and Paul H. J. Kelly. Delayed evaluation self-optimising software components as a programming model. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing: Proceedings of the* 8th *International Euro-Par Conference*, number 2400 in LNCS, pages 666–673, Paderborn, Germany, August 2002.
14. Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. *ACM SIGPLAN Notices*, 34(5):281–292, May 1999. Proceedings of PLDI'99.
15. John McCarthy. History of LISP. In R L Wexelblat, editor, *The first ACM SIGPLAN Conference on History of Programming Languages*, volume 13(8) of *ACM SIGPLAN Notices*, pages 217–223, 1978.
16. OpenMP C and C++ Application Program Interface, Version 2.0, March 2002. Available via `www.openmp.org`.
17. Shamik Sharma, Anurag Acharya, and Joel Saltz. Deferred Data-Flow Analysis. Technical Report TRCS98-38, University of California, Santa Barbara, December 30, 1998.
18. Tim Sheard and Simon Peyton-Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, December 2002.
19. Mahadhevan Subramanian. A C++ library to manipulate parallel computation plans. Msc thesis, Department of Computing, Imperial College London, U.K., September 2001.
20. Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, October 2000.

21. R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, Shih-Wei Liao, Chau-Wen, Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.
22. Kwok Cheung Yeung and Paul H. J. Kelly. Optimising java rmi programs by communication restructuring. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference 2003, Rio De Janeiro, Brazil, 16–20 June 2003*, Lecture Notes in Computer Science. Springer-Verlag, June 2003.