

A Library for Explicit Dynamic Code Generation and Optimisation in C++

Olav Beckmann, Peter Fordham, Alastair Houghton and Paul Kelly
Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2BZ
United Kingdom
Email {o.beckmann,p.kelly}@ic.ac.uk

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 dependence structure and perform optimisations. In this paper, we present the design of the TaskGraph library, present several sample applications for this library which demonstrate its use for runtime code specialisation and optimisation and discuss related work and future applications and developments of the library.

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, using delayed evaluation to recover the lost context information. Current specific 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 [19] and runtime cross-component loop fusion [5].

The TaskGraph Library. The TaskGraph library, described in this paper, is a key tool which we are developing in order to drive this research programme. The library is written in C++ and can be used to perform transparent dynamic code generation and specialisation:

- *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 2.1), we show an example of specialising a generic image filtering function to the particular convolution matrix being used.
- *Runtime Dependence Analysis*
The TaskGraph library implements basic dependence analysis on loops operating on arrays. Dependence information is crucial for determining the legality of cross-component optimisations.
- *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 [18].
- *Runtime Component Fusion*
The TaskGraph library can perform runtime loop fusion. We have used this functionality in an ongoing project for performing fusion of data-parallel components at runtime [5].

The TaskGraph library was created by Alastair Houghton as part of his 4th year MEng thesis [9].

Relationship with Earlier Work. There are several earlier tools for dynamic code optimisation reported in the literature [4, 7]. 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++¹. This is in contrast with approaches such as ‘C [4] which rely on a special compiler for processing dynamic constructs. The TaskGraph library’s support for manipulating code as data was pioneered in Lisp [15].

- *Explicit Specification of Dynamic Code*

Like ‘C [4], 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 [7, 8] 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 the TaskGraph library’s API, including a detailed worked example in Section 2.1 and an overview of TaskGraph syntax, together with an explanation of how the library works in Section 2.2. In Section 3 we discuss a different example of how we have used the TaskGraph library: a RISC emulator which we specialise to derive a just-in-time (JIT) compiler. In Sections 4 and 5 we discuss related and ongoing work, and Section 6 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)

¹Currently, we have only built the TaskGraph library with GNU g++. This is partly due to the fact that we rely on dynamic library facilities as implemented in glibc.

```
1 #include <stdio.h>
2 #include <TaskGraph>
3
4 int main( int argc, char *argv[] ) {
5     TaskGraph T;
6     int b = 1, c = 1;
7
8     taskgraph( T ) {
9         parameter( int, a );
10
11         a = a + c;
12     } endtaskgraph;
13
14     T.compile();
15     T.execute( "a", &b, NULL );
16
17     printf( "b=%d\n", b );
18     return 0;
19 }
```

Figure 1. Simple Example of using the TaskGraph library.

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 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.

2.1 Example: 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 ordinary C as shown in the left-hand part of 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

```

258 void convolution( const int IMG SZ,
259                 const float *image,
260                 float *new_image,
261                 // convolution matrix size
262                 const int CSZ,
263                 const float *matrix ) {
264     int i, j, ci, cj;
265
266     assert( CSZ % 2 == 1 );
267     const int c_half = ( CSZ / 2 );
268
269     // Loop iterating over image
270     for( i = c_half; i < IMG SZ - c_half; ++i ) {
271         for( j = c_half; j < IMG SZ - c_half; ++j ) {
272             new_image[i * IMG SZ + j] = 0.0;
273
274             // Loop to apply convolution matrix
275             for( ci = - c_half; ci <= c_half; ++ci ) {
276                 for( cj = - c_half; cj <= c_half; ++cj ) {
277
278                     new_image[i * IMG SZ + j] +=
279                         image[(i+ci) * IMG SZ + j+cj] *
280                         matrix[(c_half+ci) * CSZ + c_half+cj];
281                 }
282             }
283         }
284     }
285     return;
286 }

```

```

289 TaskGraph
290 *taskgraph_convolution( const int IMG SZ,
291                        const int CSZ,
292                        const float *matrix ) {
293     int ci, cj;
294     const unsigned int dims[] = {IMG SZ, IMG SZ};
295
296     TaskGraph T;
297     taskgraph( T ) {
298         c_fixedarrayparameter(float,tgimg,2,dims);
299         c_fixedarrayparameter(float,new_tgimg,2,dims);
300
301         assert( CSZ % 2 == 1 );
302         const int c_half = ( CSZ / 2 );
303
304         // Loop iterating over image
305         loopfor( i, c_half, IMG SZ - (c_half + 1) ) {
306             loopfor( j, c_half, IMG SZ - (c_half + 1) ) {
307                 new_tgimg[i][j] = 0.0;
308
309                 // Loop to apply convolution matrix
310                 for( ci = -c_half; ci <= c_half; ++ci ) {
311                     for( cj = -c_half; cj <= c_half; ++cj ) {
312
313                         new_tgimg[i][j] +=
314                             tgimg[i+ci][j+cj] *
315                             matrix[(c_half+ci)*CSZ + c_half+cj];
316                     }
317                 }
318             } endloop;
319         } endloop;
320     } endtaskgraph;
321     return new TaskGraph( T );
322 }

```

Figure 2. Generic image filtering — C++ (left) and function constructing the TaskGraph for a specific (static) convolution matrix (right).

- The loop bounds of the inner loops over the convolution matrix are unknown, hence these loops, with most likely very low trip-count, cannot be unrolled.
- 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.

The right-hand part of Figure 2 shows a function which constructs a TaskGraph that is specialised to the particular convolution matrix being used. The `loopfor ... endloop` 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. The inner loops are therefore completely unrolled.

We study the performance of this example in Figure 3. The convolution matrix used was a 3×3 averaging filter, images were square arrays of single-precision floats ranging in size up to 4094×4096 . Measurements are taken on a Xeon 2.2GHz running Linux 2.4.27, with gcc 2.95.3 and the Intel C++ compiler version 7.0. We compare the performance of the following:

- The static C++ code, compiled with gcc 2.95.3.
- The static C++ code, compiled with the Intel C++ compiler version 7.0. The icc compiler reports that

the innermost loop (“for-cj”) has been vectorised². Note, however, that this loop will have a dynamically determined trip-count of 3, *i.e.* the Xeon’s 16-byte vector registers will not be filled.

- The code dynamically generated by the TaskGraph library, compiled with gcc. The two innermost loops are unrolled.
- The code dynamically generated by the TaskGraph library, compiled with icc. 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 runtime compilation therefore needs to be recovered in just a single application of the generated code. Figure 3 shows that we do indeed get an overall speedup for image sizes that are greater than about 1500×1500 . In the right-hand part of Figure 3, 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

²The SSE2 extensions implemented on Xeon and Pentium 4 processors include 16-byte vector registers and corresponding vector instructions [10].

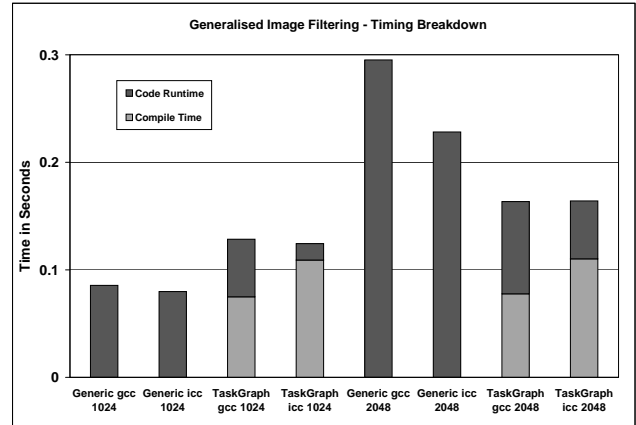
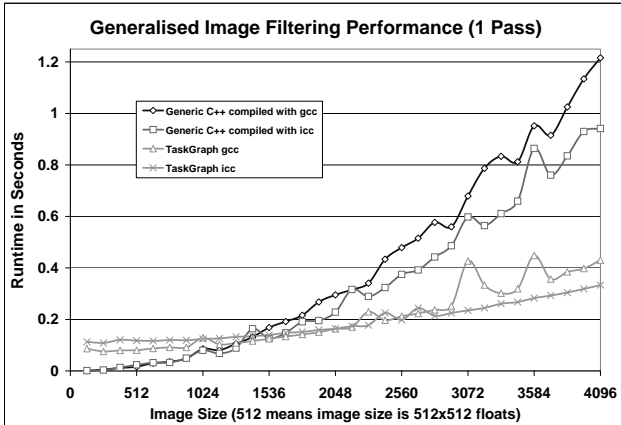


Figure 3. Performance of image filtering example. Left: Total execution time, including runtime compilation, for one pass over image. Right: Breakdown of total execution time into compilation time and execution time of the actual convolution code for two specific image sizes: 1024, which is below and 2048, which is above the break-even point for using the TaskGraph library.

the actual image filtering code, the constant overhead of runtime compilation outweighs this benefit for a relatively small data size, such as 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 have to pay the runtime compilation overhead only once and will get higher overall speedups.

2.2 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 some 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 11 of Figure 1 make use of C++ operator overloading to add nodes (in this case an assignment statement) to a TaskGraph. Figure 4 illustrates this by showing a graphical representation of the complete AST which was created by the code in Figure 1. Note that the variable c has *static* binding-time for this TaskGraph. Consequently, the AST contains its value rather than a variable reference.

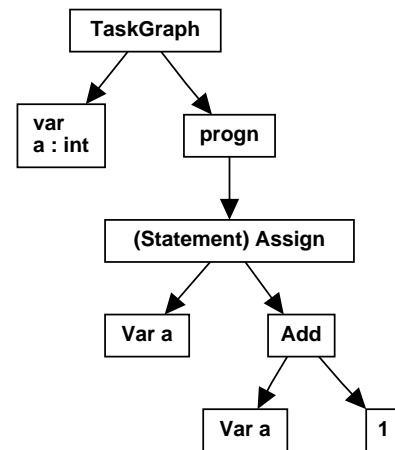


Figure 4. AST for the simple example TaskGraph of Figure 1.

The `taskgraph(T) { ... } endtaskgraph` construct determines which AST the statements in the enclosing block are attached to (in this case T). This is necessary in order to facilitate independent construction of different TaskGraphs, representing different computations³.

Local Variables in TaskGraphs. The TaskGraph library inherits lexical scoping from C++. The `loopfor` construct,

³Unfortunately, this is currently implemented by setting a global pointer variable. TaskGraph creation is therefore not thread-safe.

such as on lines 305 and 306 of Figure 2, declares a local loop control variable whose scope is just within the loop being declared. In addition to that, programmers can explicitly declare local variables within TaskGraphs using the `local(type, name)` construct. These variables are renamed when C code is generated from a TaskGraph.

TaskGraph Parameters. Both Figure 1 (line 9) and Figure 2 (lines 298 and 299) illustrate show that TaskGraphs can have parameters which are bound at TaskGraph execution time:

`parameter(type,name)` declares a basic-type parameter.

`c_arrayparameter(type,name,dims)` declares an array-parameter with a static number of dimensions and dynamic extents.

`c_fixedarrayparameter(type,name,dims,ext)` declares an array parameter with a static number of dimensions and static extents. `ext` is an array of integers of length `dims` holding the extent of the parameter in all dimensions (see This can sometimes facilitate additional optimisations).

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. Also, 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 310 and 311 of Figure 2 result in an unrolled inner loop. However, the TaskGraph sub-language defines some constructs for adding control-flow nodes to an AST:

`loopfor(var,lower,upper)` Adds a loop node. The loop bounds are *inclusive*. Notice that such a block needs to be terminated with `endloop`.

`when(cond) ...otherwise` adds a conditional node to the AST which has to be terminated by `endwhen`.

Delayed Function Calls. It is possible to insert nodes into TaskGraphs which represent functions to be called at TaskGraph execution time. However, that requires implementing a specialised class, derived from the `TaskFuncall` superclass, for each function to be called. There is currently no generic mechanism for adding a function call node to the AST.

```

296 TaskGraph T;
297 taskgraph( T ) {
298   c_fixedarrayparameter(float, tgmimg, 2, dims);
299   c_fixedarrayparameter(float, new_tgmimg, 2, dims);
300
301   assert( CSZ % 2 == 1 );
302   const int c_half = ( CSZ / 2 );
303
304   // Loop iterating over image
305   loopfor( i, c_half, IMG SZ - (c_half + 1) ) {
306     loopfor( j, c_half, IMG SZ - (c_half + 1) ) {
307       new_tgmimg[ i ][ j ] = 0.0;
308
309       // Loop to apply convolution matrix
310       for( ci = -c_half; ci <= c_half; ++ci ) {
311         for( cj = -c_half; cj <= c_half; ++cj ) {
312
313           new_tgmimg[ i ][ j ] +=
314             tgmimg[ i + ci ][ j + cj ] *
315             matrix[(c_half+ci)*CSZ + c_half+cj];
316         }
317       }
318     } endloop;
319   } endloop;
320 } endtaskgraph;

```

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

Expressions and Binding-Time Analysis. For the purpose of this discussion, we shall refer to variables that are bound at TaskGraph construction time as static variables and those that are bound at execution time as dynamic variables. Declarative code specialisation systems such as DyC [7] 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 parameters and local variables are *dynamic*. Internally, dynamic variables are represented by special types (`TaskExpr`, `TaskVariable` etc.) and the overloaded operators defined on those dynamic types define binding-time derivation rules. Thus, an expression such as `a + c` in Figures 1 and 4 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 2 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, is *static*.

TaskGraph Member Functions. The TaskGraph library implements several useful member functions.

```

1 // Define the layout of a simple 32-bit instruction
2 // word
3
4 class Instruction {
5 public:
6     unsigned opcode:5;
7     unsigned rdst:5;
8     union {
9         struct {
10            unsigned rsrc1:5;
11            unsigned rsrc2:5;
12            int shortimmediate:12;
13        } s;
14        int immediate:22;
15    };
16 };
17
18 // Define opcodes for a selection of instructions
19 #define OP_ADD 1 // rdst := rsrc1 + rsrc2
20 #define OP_BLT 6 // pc := immediate if rdst < 0
21 #define OP_RET 31 // return from interpreter;
22 // result is rdst
23
24 // interpreter
25 // Execute the bytecode, updating registers
26 // accordingly. Returns value returned by return
27 // instruction.
28 int
29 interpreter( Instruction *instructions,
30             int pc, int regs[] ) {
31     while( 1 ) {
32         Instruction inst = instructions[pc];
33         pc += 1;
34         switch( inst.opcode ) {
35             case OP_ADD:
36                 regs[inst.rdst] =
37                     regs[inst.s.rsrc1] + regs[inst.s.rsrc2];
38                 break;
39             case OP_BLT:
40                 if( regs[inst.rdst] < 0 )
41                     pc = inst.immediate;
42                 break;
43             case OP_RET:
44                 return regs[inst.rdst];
45         }
46     }
47 }

```

Figure 6. An interpretive emulator for a simple RISC instruction set.

print() writes the AST as human-readable text.

append() facilitates sequential composition of two TaskGraphs. The set of parameters of the resulting graph AST is the union of the parameter sets of the two composing parts.

optimise() performs restructuring optimisations, such as loop fusion.

setParameters() binds runtime parameters.

execute() executes the TaskGraph. If the TaskGraph has not yet been compiled, a builtin interpreter is used.

compile() writes out the TaskGraph as C++ code, compiles and dynamically links it. The particular native compiler to use can be determined with a parameter.

```

1 // specialise_interpreter
2 // Build taskgraph specialising interpreter to given
3 // pc+code. Returns result by assigning to 'result'.
4 TaskGraph *
5 specialise_interpreter( Instruction *instructions,
6                       int entrypc ) {
7     int pc = entrypc;
8     TaskGraph T;
9     taskgraph( T ) {
10        unsigned regs_size[] = {32};
11        c_fixedarrayparameter( int, regs, 1, regs_size );
12        parameter( int, newpc ); parameter( int, result );
13
14        // wrap body in loop so exit is break and branch
15        // to start is efficient (should be "while true")
16        loopfor( dummy, 0, 1000000000 ) {
17            while( true ) {
18                Instruction inst = instructions[ pc++ ];
19                switch( inst.opcode ) {
20                    case OP_ADD:
21                        regs[inst.rdst] =
22                            regs[inst.s.rsrc1] + regs[inst.s.rsrc2];
23                    continue;
24                    case OP_BLT:
25                        when( regs[inst.rdst] < 0 ) {
26                            // if the branch is back to the start
27                            // of the block, generate a 'continue'
28                            if( inst.immediate == entrypc ) {
29                                loopcontinue;
30                            } else {
31                                newpc = inst.immediate;
32                                loopbreak;
33                            }
34                        } endwhen;
35                    continue;
36                    case OP_RET:
37                        result = regs[inst.rdst];
38                        newpc = -1;
39                        loopbreak;
40                    break; // break to exit switch and return
41                } // (not reached)
42            }
43        } endloop;
44    }
45    return new TaskGraph( T );
46 }
47
48 // interpret_tg
49 // Does what 'interpreter' does, but uses cached
50 // compiled TaskGraph fragments for each basic block.
51 int interpreter_tg( Instruction *instructions,
52                  int pc, int regs[] ) {
53     // newpc and result have to be static as their
54     // addresses are bound to the cached TGs
55     static int newpc; static int result;
56     newpc = pc; // reset newpc on each entry
57     static CodeCache cache; // static so it persists
58
59     // Find an interpreter version specialised to newpc
60     // newpc = -1 means OP_RET executed and 'result' set
61     while( newpc >= 0 ) {
62         TaskGraph *T = cache.checkget( newpc );
63         if( T == NULL ) {
64             T = specialise_interpreter( instructions, newpc );
65             T->compile();
66             T->setParameters( "regs", regs, "result", &result,
67                             "newpc", &newpc, NULL );
68             cache.insert( T, newpc );
69         }
70         T->execute();
71     }
72     return result;
73 }

```

Figure 7. A RISC instruction set emulator which uses just-in-time compilation.

3 Another Example: RISC emulator

Figures 6 and 7 present another interesting example application. In Figure 6 we define the instruction encoding for a simple RISC microprocessor, and show a simple interpreter — a loop repeatedly fetches an instruction, and switches on the opcode. A subset of the instructions are shown.

A Simple Just-in-Time Compiler. In Figure 7 we use the TaskGraph library to derive a just-in-time (JIT) compiler from this interpreter. The function `specialise_interpreter` shares the structure and much of the detail of the interpreter shown in Figure 6 — but instead of interpreting the code, it builds a TaskGraph representing the work to be done.

The default behaviour for `specialise_interpreter` is to generate a TaskGraph representing a sequence of instructions, starting at `entrypc` and ending by assigning to the global variable `newpc` the address of the next instruction to be executed.

The function `interpret_tg` takes the place of `interpret`: it interprets RISC instructions starting from a specified `pc`. It maintains a cache of compiled TaskGraphs, indexed by `pc`. If there is no TaskGraph for this `pc`, it calls `specialise_interpreter` to create one, and adds it to the cache.

After it has been executed, the next `pc` is collected from `newpc`. This is then used, in turn, as an index into the cache.

For all straight-line code `specialise_interpreter` is very straightforward — the case for `OP_ADD` is identical in the interpreter. The interesting case is the conditional branch, `OP_BLT`. The `when` (line 25) constructs the run-time test. If the branch is not taken, we continue, and build the TaskGraph for the next instruction (line 35). If the branch is taken, the new program counter is given by the instruction's immediate operand. Here a tricky optimisation gains a lot of performance.

If the branch destination is the first instruction of the current block, *i.e.* `entrypc`, we generate a `continue` (line 29). In anticipation of this, the entire block of code is wrapped in a `for` loop (line 16) — so this `continue` returns control to the start.

If the branch destination is not the first instruction, we set `newpc` and generate a `break` (line 40).

3.1 Performance of the JIT

Table 1 illustrates the performance potential of this scheme. The JIT-compiled code generated by the TaskGraph library (Figure 7) runs more than three times as fast as the interpreted version (Figure 6). A simple C version

of the factorial loop shows the maximum possible performance — the JIT-generated code is still substantially slower because it updates the register vector for each RISC instruction. The factorial code results in two TaskGraphs — one for `pc = 0`, and one for the factorial loop. The `gcc` compiler is called twice at run-time, leading to a substantial overhead. The JIT-compiled code has to be executed 280,000 times to justify the cost of compilation.

Larger examples will suffer larger overheads due to taken branches which are not branches back to the head of an innermost loop. These (and function returns) lead to TaskGraph exit, cache access with the new PC, and entry to another TaskGraph. It would be interesting to consider schemes to reduce this by choosing whether to specialise following the taken or not-taken branch — in effect forming a trace cache.

4 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 [4], 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.
- *Declarative*
DyC [7,8] is a dynamic compilation system which specialises 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

| Implementation | Time per fac(10) | Approx Time per Instruction | Compilation Overhead |
|----------------|------------------|-----------------------------|----------------------|
| Interpreted | 655 ns | 15.2 ns | |
| JIT version | 189 ns | 4.4 ns | 129 ms |
| Plain C | 44.2 ns | 1.03 ns | |

Table 1. Performance of the JIT executing factorial(10), a 4-instruction loop with a 3-instruction prologue. Using gcc-2.95, running on a 2.2GHz Intel Pentium 4 Xeon

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 [6], *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.

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 [2] 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.

5 Ongoing and Future Work

We have recently evaluated the current TaskGraph library implementation in the context of some moderately large research projects [5]. This experience has led us to planning future developments of this work.

- *Using SUIF-2 as Internal Representation*
Moving internal representation of the TaskGraph library to the SUIF-2 representation [1]. This is a major re-implementation project but will facilitate the use of SUIF compiler analyses and transformation passes in optimisation.
- *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.
- *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 [18].
- *Transparent Cross-Component Loop Fusion*
In an ongoing project [5] we are using the TaskGraph library to perform cross-component loop fusion in our DESO library of data-parallel numerical routines.

6 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. Mustafa Arif carried out programming work on the TaskGraph library during the Summer of 2002. Ajay Padala implemented a prototype system for generating parallel code for SMPs from TaskGraphs as part of his 4th-year MEng thesis.

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

References

- [1] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Available online via suif.stanford.edu/suif/suif2/doc-2.2.0-4/.
- [2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00: Programming Language Design and Implementation*, pages 1–12, 2000.
- [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), Sept. 1998.
- [4] D. R. Engler, W. C. Hsieh, and M. F. 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.
- [5] P. Fordham. Transparent run-time cross-component loop fusion. MEng Thesis, Department of Computing, Imperial College London, June 2002.
- [6] R. Glück and J. Jørgensen. Fast binding-time analysis for multi-level specialization. In *Perspectives of System Informatics*, number 1181 in LNCS, pages 261–272. Springer-Verlag, 1996.
- [7] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, Oct. 2000.
- [8] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *PLDI '99: Programming Language Design and Implementation*, pages 293–304, 1999.
- [9] A. Houghton. Run-time specialisation using a C++ meta-language. MEng Thesis, Department of Computing, Imperial College London, June 2000.
- [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] N. D. Jones. Mix Ten Years Later. In *PEPM '95: Partial Evaluation and Semantics-Based Program Manipulation*, 1995.
- [13] P. Liniker, O. Beckmann, and P. H. J. Kelly. Delayed evaluation self-optimising software components as a programming model. In B. Monien and R. Feldmann, editors, *Euro-Par 2002 Parallel Processing: Proceedings of the 8th International Euro-Par Conference*, number 2400 in LNCS, pages 666–673, Paderborn, Germany, Aug. 2002.
- [14] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. *ACM SIGPLAN Notices*, 34(5):281–292, May 1999. Proceedings of PLDI'99.
- [15] J. 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, Mar. 2002. Available via www.openmp.org.
- [17] S. Sharma, A. Acharya, and J. Saltz. Deferred Data-Flow Analysis. Technical Report TRCS98-38, University of California, Santa Barbara, Dec. 30, 1998.
- [18] M. Subramanian. A C++ library to manipulate parallel computation plans. Msc thesis, Department of Computing, Imperial College London, U.K., Sept. 2001.
- [19] K. C. Yeung and P. H. J. Kelly. Automated optimisation of distributed java programs across network boundaries. In *Compilers for Parallel Computers*, Amsterdam, The Netherlands, 2003. To appear.