Parallel Processing Letters

GENERATIVE AND ADAPTIVE METHODS IN PERFORMANCE PROGRAMMING

PAUL H J KELLY and OLAV BECKMANN

Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2AZ, United Kingdom

ABSTRACT

Performance programming is characterized by the need to structure software components to exploit the context of use. Relevant context includes the target processor architecture, the available resources (number of processors, network capacity), prevailing resource contention, the values and shapes of input and intermediate data structures, the schedule and distribution of input data delivery, and the way the results are to be used. This paper concerns adapting to dynamic context: adaptive algorithms, malleable and migrating tasks, and application structures based on dynamic component composition. Adaptive computations use metadata associated with software components — performance models, dependence information, data size and shape. Computation itself is interwoven with planning and optimizing the computation process, using this metadata. This reflective nature motivates metaprogramming techniques. We present a research agenda aimed at developing a modelling framework which allows us to characterize both computation and dynamic adaptation in a way that allows systematic optimization.

Keywords: Software performance, components, parallel processing, compilers, metaprogramming

1 Introduction

This paper aims to establish principles for constructing software that make performance optimization tractable, and thereby to elaborate a manifesto for research towards achieving performance goals in software engineering.

1.1 Performance Programming

The title refers to "performance programming", rather than parallel programming, to emphasize the proper objective of parallelization — to achieve high performance — and this implies attention to all significant optimization opportunities. However, we must also consider the costs, both in time spent writing and optimizing a program, and in the long-term impact of optimizing transformations on the value of the software. Performance programming is the discipline of software engineering in its application to achieving performance goals.

1.2 Constructive methods

We focus here on performance programming "by construction", in the sense of "by design". How can we build performance into a software project? How can we build-in the means to detect and correct performance problems, as early as possible, with minimal disruption to the software's long-term value?

1.3 Adaptation to context

Most performance improvement opportunities come from adapting components to their context. This conflicts, necessarily, with the desire to re-use components in different contexts. Thus, most performance improvement measures break component abstraction boundaries.

So the art of performance programming is to figure out how to design and compose components so we can optimize effectively, while retaining abstraction and supporting re-use.

This paper is about two ideas which can help:

- Component metadata, characterizing data structures, components, their dependence relationships and their optimization opportunities (Sections 2 and 3), and
- Composition metaprogramming: optimisation by adaptation to context takes place when components are invoked by client code, and composed with other components. The optimizer uses the component metadata, together with the structure of the client code, to find an optimal execution plan based on the context in which each component is used (Sections 4 and 5).

We present specific examples to illustrate communication fusion, data alignment in sequences of parallel library calls, partial evaluation/specialization, adapting to the hardware platform/resources, and cross-component loop fusion.

1.4 Contributions of this paper

From these examples, we derive a unifying approach to the design of components which can be adapted to context — context here meaning not just the source code or method caller, but also dynamic information such as available resources, scheduling constraints, run-time data shapes, sizes and values.

To support this, components need to carry metadata. We explore what this metadata needs to do, and some of the challenges in using metadata characterizing dependence relationships and performance models to support composition-time adaptation.

2 Adaptation to context: communication fusion

Consider a parallel function using MPI to compute the variance of a distributed array `data`. We call a function `sum` to compute the sum, and another, `sumsq` to compute the sum of squares. Both `sum` and `sumsq` involve a global reduction

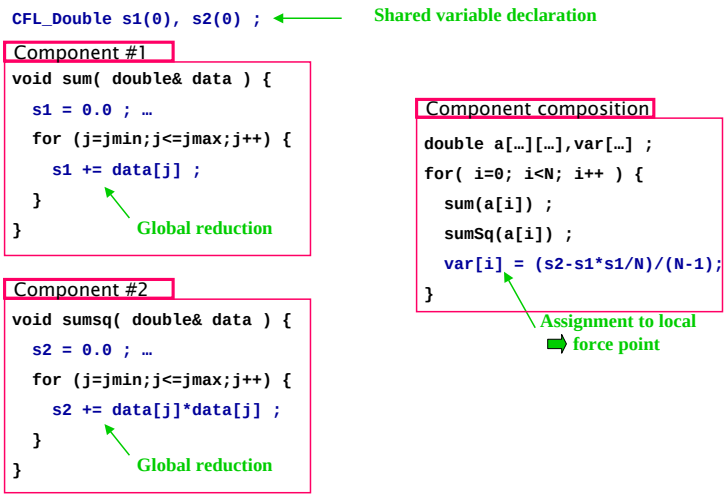

Fig. 1: Automatic fusion of MPI reduction operations. Both `sum` and `sumsq` involve an MPI_Allreduce, but we overload the C++ "+" operator to delay execution so that communication is automatically aggregated. This optimization yields a 44.5% speedup for N=3000 on a 4-processor Linux cluster.

operation, an `MPI_Allreduce()`. An efficient calculation of variance needs just one `MPI_Allreduce()`, that sums both the values and the squares in a single round of communication.

Figure 1 shows how the Communication Fusion Library [1] solves this problem. The `MPI_Allreduce()` call is hidden in an abstract data type for global, shared variables. Arithmetic on global variables is delayed until forced by an assignment of a global value to a local variable. At this point all delayed reductions can be executed in a single `MPI_Allreduce()` call.

This work was motivated by a parallel ocean plankton ecology modelling framework. Depending on the sophistication of the particular ecology being simulated, many different global variables need to be maintained. Variables, for example nutrient concentration, may be adjusted during a timestep as different plankton species are accounted for. Execution of delayed communication is forced when the model needs to use the current value of a global variable. In a typical model, 27 scalar reductions are required, but can be implemented in just 2 `MPI_Allreduce()` calls, leading to a 60% speedup on a 32-processor AP3000 [1].

2.1 The communication fusion library as a component model

This example shows a simple adaptation to context: a cross-component optimization, where the components are functions defining and using global shared variables. The library does this while preserving the source program's structure, and the reusability of the components.

The library works at run-time, though the idea could be used in a compile-time optimizer — the core idea is that each component carries a description (hidden in

the global shared variable abstract data type) of which global shared variables it defines and uses. When the components are composed, we can use this "component dependence metadata" to determine where the fused `MPI_Allreduce()` calls have to go.

We chose a run-time implementation because the run-time cost of optimizing relatively heavyweight communication operations usually outweighs the benefits — this turned out not to be so on some very fast networks [1], where the benefits are small. In general, architecture-specific performance models need to be used to identify the optimum implementation strategy [2].

```
template<class Matrix, class Vector, class Precond, class Real>
int CG( const Matrix &A, Vector &x,
        const Vector &b, const Precond &M,
        int &max_iter, Real &tol )
{
  // local vector and scalar declarations & initial convergence test omitted

  for( int i = 1; i <= max_iter; i++ ) {
    z = M.solve( r );
    rho(0) = dot(r, z);

    if (i == 1)
      p = z;
    else {
      beta(0) = rho(0) / rho_1(0);
      p = z + beta(0) * p;
    }
    q = A*p;
    alpha(0) = rho(0) / dot(p, q);

    x += alpha(0) * p;
    r -= alpha(0) * q;

    if( (resid = norm(r) / normb) <= tol ) {
      tol = resid;
      max_iter = i;
      return 0;
    }
    rho_1(0) = rho(0);
  }
  tol = resid;
  return 1;
}
```

Fig. 2: Conjugate-gradient algorithm.

3 Adaptation to context: data alignment

Figure 2 shows a generic conjugate-gradient solver algorithm, part of Dongarra et al's IML++ library [3]. It is parameterized by the Matrix and Vector types. The DESOBLAS library [4] implements this API for dense matrices, in parallel using MPI.

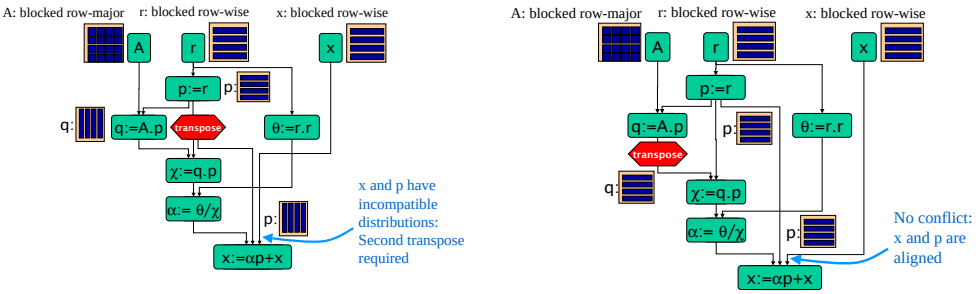

Fig. 3: Data flow for first iteration of the conjugate gradient solver shown in Figure 2. The vector-vector product $\chi = q.p$ needs vectors p and q to be aligned, but they are not, since the matrix-vector product $q = A.p$, produces a column-wise result if the operand p is row-wise. The first figure shows the effect of resolving this conflict by transposing p — this is short-sighted since p is later needed aligned with x. The second figure shows the benefit of foresight — if we transpose q instead, one transpose suffices.

Each vector-vector and matrix-vector operation (the overloaded "`+`" and "`*`" operators in the C++ code) operates on data distributed across a mesh of processors. The matrix `A` is distributed in a block-block fashion. The vectors `p`, `r` and `x` are replicated on each row of the processor array, and distributed block-wise within each row. For the matrix-vector multiply `q=A.p`, a copy of vector `p` is, thus, aligned with each row of `A`; we compute partial inner-products for each block, and sum them across the rows to produce a column-wise result `q` (see [5]).

As a consequence, `p` and `q` necessarily end up with conflicting distributions: one row-wise, one column-wise. The next step is to compute the inner product `q.p`, for which they have to be aligned. We need to transpose one of them. As illustrated in Figure 3, the choice of which one to transpose depends on how they will be used later in the computation.

Using foreknowledge of how distributed intermediate results will be used, we can avoid unnecessary redistributions. Results reported in [6] for the conjugate gradient algorithm on an ethernet-connected Linux cluster show a reduction in communication time of 15%–50%, although the impact on overall execution time is smaller.

3.1 Data alignment metadata

The optimization problem is to minimize time spent on redistributing data by aligning each result with the computation that uses it, if possible. Where an alignment conflict cannot be avoided, we need to resolve it using the lowest-cost redistribution.

The alignment of each vector or matrix x is characterized by an affine alignment function $align(x)$. Each library operator carries metadata, consisting of a set of affine functions relating operator's output data placement to the placement of each input. For example, for the matrix-vector multiply $q = A.p$, the alignment functions of its inputs and outputs are related:

$$align(p) \quad = \quad align(A)$$

$$align(q) \quad = \quad align(p)^T$$

The arcs of the data flow graph define a network of invertible linear equalities — in effect, a system of equations. To solve it, the optimizer can shift redistributions around the dataflow graph to minimize communication cost; a review of optimization algorithms and heuristics is presented in [7].

3.2 Component metadata in data alignment optimization

Again, each component carries metadata which is used to optimize it to its context of use. The optimizer that uses this metadata analyses the way the application program calls the component library, and encodes special knowledge of the optimization requirements and opportunities that arise when the components offered by the library are composed.

Again, our work in this area has adopted a run-time optimization strategy, and our results show the overheads of run-time optimization are small and can be reduced further by caching optimized execution plans for subsequent reuse. The principle is independent of whether optimization is at compile-time or run-time: metadata associated with functions/components is used by a domain-specific optimization pass which operates on the application code that calls the library.

4 Adaptation to context: specialization

The third form of optimization by adaptation to context we explore is specialisation, sometimes called partial evaluation. Much research [8] has been focussed on automatic partial evaluation, starting from a non-specialising code version, perhaps annotated to indicate run-time constant variables. A good example is DyC [9], a compiler for C that generates a run-time specialiser which is invoked when the program first uses an annotated run-time–constant value.

The more explicit alternative that we present here is to program the generation process explicitly. "Multi-stage" programming languages, such as MetaOCaml [10] support this as a first-class language feature. Our TaskGraph library for C++ exploits templates and overloading to achieve much of this without special compiler support.

The complete example program in Figure 4 dynamically creates a piece of code for the expression `x+y+1`. The resulting taskgraph represents a function with two integer arguments and an integer result. Finally the taskgraph is passed application program variables `a` and `b` as parameters and executes the code, printing `a+b+c = 6` as the result. The code for `x+y+c` is specialised for `c = 1`.

The type of an expression determines whether it forms part of the generated code, or instead is executed at code generation time. Expressions and statements which involve only primitive C/C++ variables (such as `c`) are executed normally. TaskGraph expressions are expressions that involve TaskGraph variables (parameters such as `x` and `y` above, and, in later examples, variables declared using `tVar`).

Arithmetic operators such as `+` are overloaded, so that if one of the operands is a TaskGraph expression or TaskGraph variable, instead of doing an addition, the

```
int main() {
  int c = 1;
  TaskGraph < Par < int, int >, Ret < int > > T;
  taskgraph( T, tuple2(x, y) ) {
    tReturn( x + y + c );
  }
  T.compile( tg::GCC, true );
  int a = 2;
  int b = 3;
  printf( "a+b+c = %d\n", T.execute( a, b ) );
}
```

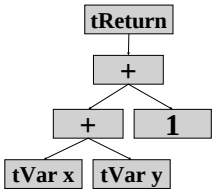

Fig. 4: *Left:* Simple example of using the TaskGraph library. The TaskGraph's argument is a 2-tuple containing the two integer formal parameters; it has an integer result type. *Right:* Abstract syntax tree (AST) for the simple TaskGraph constructed by the piece of code shown on the left. The variable `c`, which is static at TaskGraph construction time, does not appear in the tree – instead `c`'s value is used.

`+` operator creates a node for the taskgraph's abstract syntax tree. The `tReturn()` function creates a return statement node in the taskgraph. As we see in the next example, assignment is also overloaded to add an assignment node, and there are TaskGraph versions of the usual control structures — `tFor` for `for`, `tIf` for `if`, and so on.

4.1 Generalised image convolution

Figure 5 shows how specialisation can be used at run-time to achieve a substantial performance improvement. In this image convolution example, we sweep over an input image, computing at each pixel the sum of the `CSZ*CSZ` neighbouring pixels, weighted by a second image `mask`. The function builds a version of the loop nest, specialized to the particular `mask`. The loops over the mask array are executed as the taskgraph is generated, so the mask values appear as constants (possibly zero) in the generated code.

The effect is that the taskgraph contains control flow nodes for the outer `i` and `j` loops and a loop body consisting of `CSZ * CSZ` assignment statements.

Figure 6 illustrates the performance that can be achieved. The convolution mask 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 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 (-O3).
- The static C++ code, compiled with the Intel C++ compiler version 7.1 (-restrict -O3 -xc -xiMKW -tpp7 -fno-alias). The `icc` compiler reports that the innermost loop `for(cj..)`) has been vectorised[a]. Note, however, that this

[a]The SSE2 extensions implemented on Pentium 4 processors include 16-byte vector registers

```
void specialize_convolution(
  TaskGraph < Par <float[IMG_SIZE][IMG_SIZE], float[IMG_SIZE][IMG_SIZE]>,
              Ret < void > > &T,
  const int IMGSZ, const int CSZ, const float *mask ) {
  int ci, cj;
  assert( CSZ % 2 == 1 );
  const int c_half = ( CSZ / 2 );

  taskgraph( T, tuple2(tgimg, new_tgimg) ) {
    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][j] = 0.0;

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

Fig. 5: Generic image convolution: this function builds a taskgraph for a convolution operation specialised to a particular `mask`. The outer loops, `tFor(i..)` and `tFor(j..)`, produce loops in the generated code. The inner loops, `for(ci..)` and `for(cj..)`, are executed during taskgraph construction — producing multiple copies of the loop body. Each instance of the loop body is specialised for particular values of `ci` and `cj`. The expression `mask[c_half+ci][c_half+cj]` involves no taskgraph variables so its value is calculated at construction time.

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`.

The results show that the overhead of generating code and compiling it at run-time is around 100ms, so for a single convolution operation, speedup occurs only with images larger than 1024×1024. For large images the specialized code is more than three times faster.

and corresponding instructions which operate simultaneously on multiple operands packed into them [11].

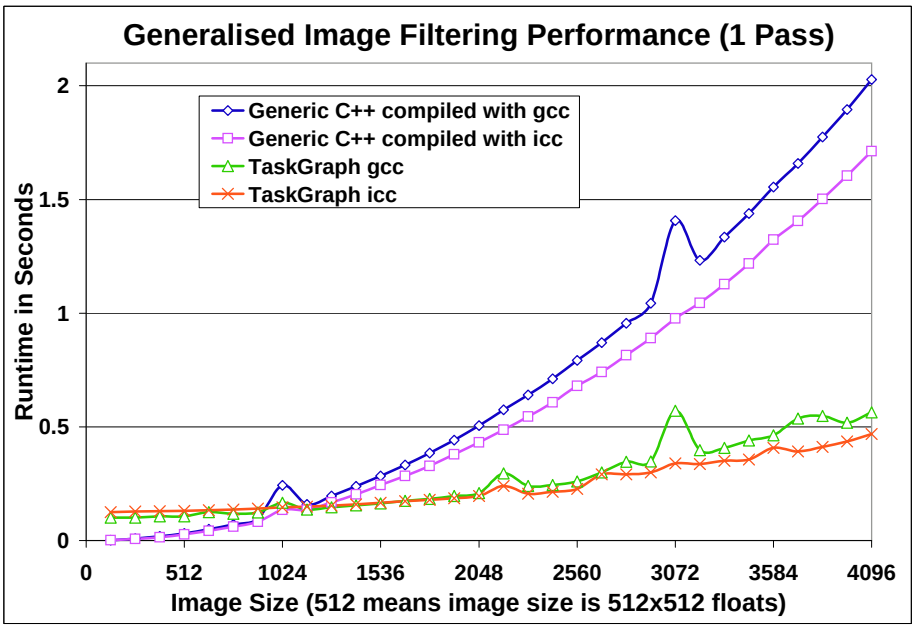

Fig. 6: Performance of image convolution example. The graphs show total execution time, including run-time compilation, for one pass over the image. The poor performance with 1024×1024 and 3072×3072 images is due to cache interference effects.

4.2 Specialisation and performance metadata

Components like image convolution can use specialization internally without exposing the fact to the caller. However, better performance can often be achieved if the caller is involved in deciding when, and whether, to invest in run-time code generation. To do this the component has to expose the information the caller needs to estimate performance, and to evaluate the likely costs and benefits of specialisation.

For example, in the case of image convolution, unspecialized execution time is characterized by an expression dominated by the product of the image size with the mask size. With specialization, execution time is a product of image size and the number of non-zero mask elements, but likely with lower constants due to vectorization and reduced loop overheads. However the specialization process itself takes time proportional to the mask size (although in our current implementation the 100ms overhead of invoking the compiler usually makes code size a minor factor).

5 Adaptation to platform

The fourth form of optimization by adaptation to context we explore is tuning the code for the particular hardware on which it will execute.

Many applications are distributed to customers with diverse hardware, or are executed on resources allocated on-demand. Re-optimization to exploit the available hardware may simply be a matter of recompilation. Libraries like Atlas [12]

```
typedef float MatrixType[MATRIXSIZE][MATRIXSIZE];
float MatrixType a, b, c;
typedef TaskGraph< Par<MatrixType, MatrixType, MatrixType>, Ret<void> > mm_TaskGraph;

void taskMatrixMult ( mm_TaskGraph &t, TaskLoopIdentifier *loop ) {
  taskgraph ( t, tuple3(a, b, c) ) {
    tVar ( int, x ); tVar ( int, y ); tVar ( int, z );

    tGetId ( loop[0] ); // label
    tFor ( x, 0, MATRIXSIZE - 1 ) {
      tGetId ( loop[1] ); // label
      tFor ( z, 0, MATRIXSIZE - 1 ) {
        tGetId ( loop[2] ); // label
        tFor ( y, 0, MATRIXSIZE - 1 ) {
          c[x][y] += a[x][z] * b[z][y];
}}}}}

main () {
  int bestTime; int bestSize = 0;
  for (int tsz = 4; tsz <= MATRIXSIZE; ++tsz) {
    int trip3 = { tsz, tsz, tsz };
    TaskLoopIdentifier loop[3];
    mm_TaskGraph MM;
    taskMatrixMult(loop, MM);
    interchangeLoops(loop[1], loop[2]);
    tileLoop(3, &loop[0], trip3);
    MM.compile(TaskGraph::ICC);
    tt3 = time_function();
    MM.execute(A, B, C);
    time = time_function()-tt3;
    if (time < bestTime || bestSize == 0) {
      bestTime = time; bestSize = tsz;
}}}
```

Fig. 7: This example illustrates the TaskGraph library's metaprogramming capabilities. Having built a simple loop nest to encode a matrix multiply, we call SUIF transformations to interchange and tile loops. We record the performance achieved with each tile size and record the optimum value for future runs. An example of the generated code is shown in Figure 8.

```
extern void taskGraph_1(void **params)
{
  float (*a)[512]; float (*b)[512]; float (*c)[512];
  int i; int j; int k; int j_tile; int k_tile;

  a = *params; b = params[1]; c = params[2];
  for (i = 0; i <= 511; i++)
    for (j_tile = 0; j_tile <= 511; j_tile += 360)
      for (k_tile = 0; k_tile <= 511; k_tile += 360)
        for (j = j_tile; j <= min(511, 359 + j_tile); j++)
          for (k = max(0, k_tile); k <= min(511, 359 + k_tile); k++)
            c[i][k] = c[i][k] + a[i][j] * b[j][k];
}
```

Fig. 8: Example of the code generated by Figure 7, for 360×360 tiles (slightly tidied). For large matrices on a 1.8GHz Pentium-4-M with 512KB level-2 cache, using the Intel C compiler version 7.1 this is the optimal tile size, and achieves ca.2 GFLOPs (single precision), more than five times the performance of the naive code. Further transformations are needed to maximise performance (the Atlas version can reach more than 4 GFLOPs), including index set splitting to remove the `min` and `max` operators, hierarchical tiling (for TLB, registers and multiple levels of cache) and copying of submatrices to reduce cache associativity conflicts.

and FFTW [13] run performance experiments at installation-time to select the optimal code variants to use for key library functions. "Iterative compilation" is the idea of extending this to application code by empirically searching the space of code transformations and synthesis alternatives [14]. Figure 7 shows a simple example of doing this using our TaskGraph library's metaprogramming tools, which provide access to analyses and restructuring transformations implemented using the SUIF [15] and ROSE [16] frameworks.

This simple example illustrates the potential for programmer control over application of sophisticated transformations. The performance benefits can be large — in this extreme example a factor of eight or more. Different target architectures and problem sizes need different combinations of optimisations. Matrix multiply is a simple example, but has a large space of available transformations — should the loops at each level be nested in "ijk" or "ikj" order? Should we tile hierarchically — for registers, TLB, and multiple levels of cache? Should we copy the reused submatrix into contiguous memory?

5.1 Abstraction in automatic adaptation to hardware architecture

The problem here is not finding valid optimizing transformations, but to constrain the empirical search for the optimum combination of transformations to apply. There are two parts:

- Leaf routines and fully-inlined code, and
- Code built by composing library functions

To extend the performance benefits of iterative compilation beyond libraries without exploring a enormous transformation space, we need to be able to characterise the ways in which a library function can adapt to its context of use.

For a very simple example, consider summing three images, represented as arrays of 32-bit `floats`. We can use the Intel Performance Primitives library's function to add two such images at a time:

```
ippiAdd_32f_C1R( image1, size , image2 , size, sum12, size , whole );
ippiAdd_32f_C1R( image3, size, sum12, size, sum123 , size , whole );
```

The first call adds `image1` and `image2`. The second adds this to `image3`. Inlining the naive code for image addition produces:

```
for ( i = 0; i <= size; i++)
  for ( j = 0; j <= size; j++)
    sum12[ i ][ j ] = image1[ i ][ j ] + image2[ i ][ j ];
for ( i = 0; i <= size; i++)
  for ( j = 0; j <= size; j++)
    sum123[i ][ j ] = image3[ i ][ j ] + sum12[ i ][ j ];
```

Loop fusion can be applied:

```
for ( i = 0; i <= size; i++)
  for ( j = 0; j <= size; j++)
    sum12[ i ][ j ] = image1[ i ][ j ] + image2[ i ][ j ];
    sum123[i ][ j ] = image3[ i ][ j ] + sum12[ i ][ j ];
```

However, Intel's library code for image addition is carefully implemented, and it is faster to forego loop fusion, and use the library code to sweep over the images twice — until the image size exceeds around 4000×4000 (on a 1.8GHz Pentium 4-M using the Intel C Compiler 8.0).

The role for component metadata here is to carry information like this — to select loop fusion only when the image size exceeds some architecture-specific threshold.

6 Discussion

We have presented four examples of optimization by adaptation to context:

- Communication fusion (Section 2)
- Data alignment (Section 3)
- Specialisation (Section 4)
- Adaptation to hardware platform (Section 5)

In each case we have observed how exploiting this optimization manually can lead to damage to the program's abstractions.

6.1 Component metadata

In each case, we have explored the proposition that using metadata associated with the program's components, this damage can be prevented:

- For communication fusion: metadata tracks the global shared variables defined and used.

  When components are composed, we can use this to determine when collective, fused, communications have to be executed.

- For data alignment optimisation: metadata characterises each component's data alignment constraints.

  When components are composed, we assemble these constraints and solve for the minimum-cost assignment of data alignments to intermediate operands.

- Specialisation: metadata characterizes the likely costs and benefits of specialization with respect to each parameter, possibly as a function of the parameter's value or size.

  This information is used to decide whether to create a version of the component specialised to a particular context.

- For adaptation to the hardware platform, metadata characterizes the available optimizing transformations, and provides a model for their effect on performance.

  This is used to guide the search for the optimum combination of optimizing transformations. When components are composed, it identifies fusion/blocking opportunities and characterizes their benefits and costs.

6.2 Composition metaprogramming

The code that uses component metadata operates at the point where components are assembled — in the calling, client code. This is a form of metaprogramming [17], since it plans program execution, possibly generating code.

For each of the adaptation optimizations above, there is a "composition metaprogram" that uses component metadata to find the optimum way to execute the components being assembled.

6.3 A generic framework

The form of the component metadata depends on the optimisation opportunities being considered, and this, in turn, depends on the application domain. The examples we have presented illustrate some of the potential variety, and demonstrate the need for an open, extensible framework. A starting point would be the metadata/annotation schemes found in Java [18] or attributes in .Net [19]. The research challenge is to devise a component metadata structure that supports a wide range of adaptations in a common framework, so that different composition metaprograms have access to all the adaptation opportunities the components offer.

7 Related work

Explicit composition metaprogramming — the idea of analysing component context to plan computation accordingly — has a considerable heritage. This is what

compilers do, of course, but more interesting examples operate at later stages. For example, KeLP [20], BSP [21] and CHAOS/PARTI [22] explicitly schedule message exchanges to implement the communication pattern required at run-time. Metadata in the form of performance models has been used at run-time to control granularity in task-stealing systems [23]. The macro data-flow system Mentat [24] supported dynamic partitioning of the data-flow graph. Veneer is a generic run-time optimization framework that uses dependence metadata about client code fragments to re-order and coalesce remote calls [25].

Procedure summary data is used in compilers to support scalable interprocedural analysis [26]. While this addresses the scalability of analysis rather than synthesis, "procedure cloning" attempts to gain the advantages of full inlining (and thus full adaptation to context) using a minimum number of implementation variants [27]. ICENI [28], a grid component framework, uses performance models to select implementation variants and to match them to available computing resources. In [29], cost models for computational components are combined at the composition level using cost models for higher-order skeleton operators.

"Active libraries" are libraries that take an active role in compilation of client code [30]. To build them, generic frameworks for building domain-specific optimizers are needed; we have been experimenting with ROSE [16]. Kennedy proposes a "telescoping" approach to optimization to context, where a large number of different component compositions are precompiled [31].

In [32] we proposed component dependence metadata, that characterizes the internal iteration space and dependence structure. For regular loop nests with well-behaved dependence structure, the Kelly-Pugh [33] or Polytope-based [34] compositional transformation frameworks can be adopted. The challenge is to extend such a model to irregular loops whose dependence structure depends on data values. Strout *et al* offer a step in this direction [35].

8 Conclusions

Optimizing software by adapting components to context manually reduces its long-term value because it breaks abstractions and blocks re-use. This can be avoided by automating the adaptation process. We have attempted here to map out an attack on this problem. A metaprogram — which may be a compiler plug-in, or a run-time mechanism — analyses how the components are composed, and uses metadata carried by the components to identify optimizing transformations and plan the execution.

Through a number of examples, we have explored specific instances of this general idea. Much research is needed to unify the different kinds of optimisation, the different kinds of context to which to adapt, and the different stages at which adaptation can take place.

Acknowledgements Particular thanks are due to the organisers of the CMPP 2004 workshop in Stirling who commissioned the invited talk on which this paper is based. Tony Field, Alastair Houghton, Michael Mellor, Peter Fordham, Peter

Liniker, Thomas Hansen, Kostas Spyropoulos and Peter Collingbourne also contributed to some of the work presented here. This work was funded by the EPSRC through a doctoral studentship and grants GR/R21486 (Oscar) and GR/R15566 (Desormi). Chris Lengauer and Scott Baden kindly offered advice on presentation.

References

1. A. J. Field, Paul H. J. Kelly, and Thomas L. Hansen. Optimising shared reduction variables in MPI programs. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002*. Springer Verlag LNCS 2400, 2002.
2. Sergei Gorlatch. Towards formally-based design of message passing programs. *IEEE Transactions on Software Engineering*, 26(3):276–288, March 2000.
3. J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Object Oriented Numerics Conference*, 1992.
4. Peter Liniker, Olav Beckmann, and Paul H. J. Kelly. Delayed evaluation, self-optimising software components as a programming model. In *Euro-Par 2002*. Springer Verlag LNCS 2400, 2002.
5. Michael J Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2003.
6. Olav Beckmann and Paul H. J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In *LCR98: Languages, Compilers and Run-time Systems for Scalable Computers*, pages 123–138. Springer-Verlag LNCS 1511, May 1998.
7. Olav Beckmann and Paul H J Kelly. A review of data placement optimisation for data parallel component composition. In Sergei Gorlatch and Christian Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice. Nova Science, 2000. (proceedings of CMPP2000).
8. John Hatcliff, Torben Mogensen, and Peter Thiemann, editors. *Partial Evaluation. Practice and Theory*. Springer Verlag LNCS 1706, 1999. DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998.
9. Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-directed run-time specialization in C. *SIGPLAN Not.*, 32(12):163–178, 1997.
10. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217. ACM Press, 1997.
11. Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 1999–2002. Available via `developer.intel.com`.
12. Jack J. Dongarra and Clint R. Whaley. Automatically tuned linear algebra software (ATLAS). In *Proceedings of SC'98 Conference*. IEEE, 1998.
13. Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.
14. G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computers (LCPC'02)*, pages 305–315, 2002.
15. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao,

E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.

16. Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference (JMLC'03)*. Springer Verlag LNCS 2789, 2003.
17. Tim Sheard. Accomplishments and research challenges in meta-programming. In *2nd Intl. Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44. Spring Verlag LNCS 2196, 2001.
18. Gilad Bracha et al. JSR 175: A metadata facility for the Java programming language, September 2004. Java Community Process (http://www.jcp.org).
19. Microsoft. .NET framework developer's guide: Extending metadata using attributes, 2004. (http://msdn.microsoft.com).
20. S. J. Fink, S. R. Kohn, and S. B. Baden. Efficient run-time support for irregular block-structured applications. *J. Parallel and Distributed Computing*, 50(1-2), April-May 1998.
21. Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, February 2004.
22. Ravi Ponnusamy, Joel H. Saltz, and Alok N. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing*, pages 361–370, 1993.
23. P. López-García, M. Hermenegildo, and S.K. Debray. A methodology for granularity based control of parallelism in logic programs. *Journal of Symbolic Computation*, 22, 1996.
24. Jon B. Weissman and Andrew S. Grimshaw. Network partitioning of data parallel computations. In *Third IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 1994.
25. Kwok Cheung Yeung and Paul H J Kelly. Optimizing Java RMI programs by communication restructuring. In D Schmidt and M Endler, editors, *Middleware 2003: ACM/IFIP/USENIX International Middleware Conference*. Springer Verlag LNCS 2672, 2003.
26. M. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D. *Journal of Parallel and Distributed Computing*, December 1996.
27. K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1999.
28. J. Hau, W. Lee, and Steven Newhouse. Autonomic service adaptation using ontological annotation. In *4th International Workshop on Grid Computing, Grid 2003*, November 2003.
29. Martin Alt, Holger Bischof, and Sergei Gorlatch. Program development for computational grids using skeletons and performance prediction. *Parallel Processing Letters*, 12(2):157–174, 2002.
30. Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.
31. Ken Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*. Springer Verlag, 2000.
32. Paul H J Kelly, Olav Beckmann, Tony Field, and Scott Baden. Themis: Component

dependence metadata in adaptive parallel applications. *Parallel Processing Letters*, 11(4), 2001.

33. Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, 1993.
34. C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*. Springer Verlag, 2003.
35. Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Programming Language Design and Implementation (PLDI)*. ACM, 2003.