OPTIMISING COMPONENT COMPOSITION USING INDEXED DEPENDENCE METADATA

Lee W. Howes, Anton Lokhmotov, Paul H. J. Kelly, A. J. Field

Department of Computing, Imperial College London
email: {lwh01, anton, phjk, ajf}@doc.ic.ac.uk

ABSTRACT

This paper explores the use of *dependence metadata* for optimising composition in component-based parallel programs. The idea is for each component to carry additional information about how points in its iteration space map to memory locations associated with its input and output data structures. When two components are composed this information can be used to implement optimisations that would otherwise require expensive analysis of the components' code at the time of composition. This dependence metadata facilitates a number of cross-component optimisations – in this paper we focus on loop fusion and array contraction. We describe a prototype framework, based on the CLooG loop generator tool, that embodies these ideas and report experimental performance results for three non-trivial parallel benchmarks. Our results show execution time reductions of up to 50% using the proposed framework on an 8 core xeon.

1. INTRODUCTION

Component based programming consists of writing software entities to fulfill specified interfaces. Component models allow multiple component implementations to satisfy the same interface, offering flexibility on the choice of implementation for a particular problem or computing platform. However, treating components as black boxes described by their interfaces can limit the scope for optimisation. In particular, whilst individual components can be statically optimised when the component is defined, component compositions can only be optimised at the point of use. This requires an element of dynamic optimisation that exploits context information.

Powerful but expensive inter-procedural compiler optimisations such as enabled by the polyhedral framework [1] could be used once the composite component structure is known. However, the cost of the analysis would have to be paid each time the same components were composed in the same way.

Adaptive components are explicitly programmed to make use of context information, e.g. knowledge of the components with which they are composed, in order to produce optimised execution schedules. In this paper we propose to implement a form of adaptive behaviour through the use of supplied component metadata and to use that metadata to identify dynamic optimisation opportunities at the time of composition. The fact that the metadata is supplied rather than extracted at composition time, obviates the need to analyse a component's code each time it is used, in order to identify whether cross-component optimisation opportunities exist.

The metadata we explore in this paper, which we refer to as *indexed dependence metadata*, defines the set of memory locations that a component may access at a particular point in its iteration space. The relationship between these mappings in different components serves to define implicitly the communication requirements of their compositions.

By examining the memory dependence metadata of the components in a composition, we seek to expose opportunities for cross-component optimisation that are not possible by optimising the individual components in isolation.

Specifically, in this paper we use the dependence metadata to determine whether two loops occurring separately in the components of a composition can be aligned whilst respecting dependences, in which case the loops can be fused. Fusion in turn may facilitate array contraction, reducing the space requirements of the composition, and inter-processor communication in the case where the components themselves comprise parallel loops. We use CLooG [2, 3] to generate the code for a fused loop using a scheduling matrix generated from an analysis of the components' metadata and a matrix representation of the iteration space generated from the components' source code.

The contributions of the paper are as follows:

- We introduce the idea of indexed dependence metadata, which defines the set of memory locations that may be read from and written to by a component at each point in its iteration space (Section 3).

- We show how the dependence metadata can be used in conjunction with a representation of the components' iteration spaces to implement loop fusion and array contraction across the component boundaries in a composition (Section 5). In particular, we extend this to parallel components, where the contraction reduces inter-processor communication.

- We describe a prototype software component framework incorporating the above ideas, which has potential applications in multi-core software development (Sections 2 and 4).
- We illustrate the power of the approach by showing substantial performance improvements through fusion of parallel components in linear algebra and image processing benchmarks and a 3D multigrid solver (Section 6). On an eight-core Intel Xeon system, maximum performance improvements on these examples range from 12% to 50%.

```
<interface id="iContourfilter">
    <input type="float" name="image_in"
        format="array(in_x,in_y)" />
    <output type="float" name="image_out"
        format="array(out_x,out_y)" />
</interface>
<interface id="iConvolution">
    <input type="float" name="image_in"
        format="array(in_x,in_y)" />
    <input type="float" name="filter_in"
        format="array(filter_x,filter_y)" />
    <output type="float" name="image_out"
        format="array(out_x,out_y)" />
</interface>
```

Listing 1. Interface specifications for the contour filter and convolution.

```
<component id="cf" >
  <implements id="iContourfilter" />
  <uses name="conv">
    iConvolution(
      image_in(in_x, in_y), filter_in(3, 3),
      image_out(out_x, out_y) flow to F1)
  </uses>
  <constraint type="equality">
    conv.in_x=in_x
  </constraint>
  ...
</component>
```

Listing 2. Part of the contourfilter component specification.

2. ARCHITECTURE OVERVIEW

Our component programming system is designed to select and generate code from a library of components. Components carry metadata describing functional interfaces and data dependence relationships. We identify three elements: *Component*, *Interface* and *Manager*.

The application and individual *components* depend on one or more *interfaces*. Components also implement interfaces, satisfying the contract defined by the interface. The *manager* maintains the component dependence graph and allocates component implementations to the interfaces as necessary. If a component $C1$ depends on an interface that

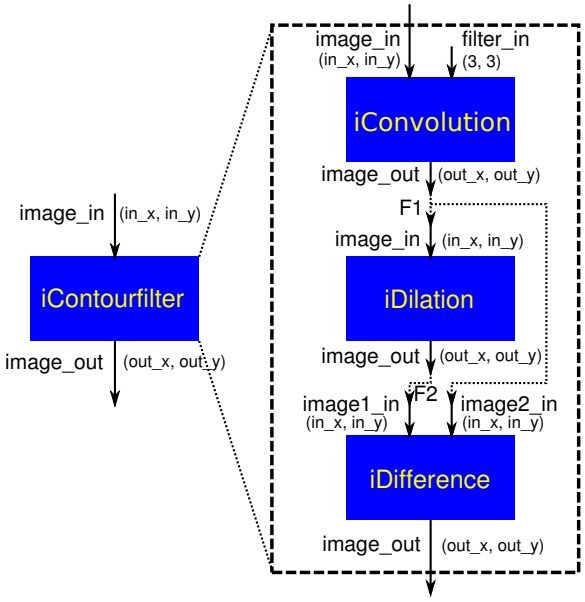

Fig. 1. A contour filter example showing dependencies, data flows and size descriptions of inputs and outputs.

is implemented by a component $C2$, we say that $C2$ is a subcomponent of $C1$. We generate the dependency graph of an application by recursively expanding the dependencies in the component graph. The assignment of components to interfaces is performed during a later graph pass.

Figure 1 shows the dependency relationships for an image filtering example. We see a *iContourfilter* interface with one input and one output, implemented by a component that depends on *iConvolution*, *iDilation* and *iDifference* interfaces to perform its computation. Flow annotations *F1* and *F2* define data flow dependencies at the composition level.

Listing 1 shows the specification for two of the interfaces in Figure 1: *iContourfilter* and *iConvolution*. Listing 2 shows part of the component specification for the contour filter (*cf*), including its dependence on its convolution subcomponent.[1] The *cf* component, which implements the *iContourfilter* interface, depends on the *iConvolution* interface. We name the dimensions of the input and output parameters, and specify a constant 3×3 size for the filter parameter. The *flow to* keyword names a data flow as in Figure 1.

The implementation language for a given component is flexible. We currently support C/C++, a high level polyhedral representation of C, or pre-compiled binaries. In principle the system can integrate components in any language, given support in the component manager.

[1]Note that our implementation currently uses XML to define interfaces, component specifications and dependence metadata, although we envisage the use of automated or GUI based tools in the future.

3. COMPONENT METADATA

In general, the input and output variables of components need to interact with those in their subcomponents. For example, variables in subcomponents can be configured to share values of variables in the parent component, and hence values can propagate through the component graph. Additional metadata can be attached to a component specification in order to express these properties. For example, Listing 2 shows an equality constraint specifying that the value *in_x* in the interface matches the *in_x* in the subcomponent named *conv*.[2] Additionally, data can flow from one subcomponent to another, and hence through various levels of the component graph when combined with parent/child relationships. In the example, the *image_out* value of *iConvolution* is connected (*flows to*) the flow *F1*, which will be connected again to an input variable in another dependency of the component. Component graph data flows are defined in the metadata, to avoid composition-time component analysis.

It should be emphasised that the aim is to provide dependence relationships on the component inputs and outputs *at composition time*, without analysis of the component code; indeed, this code might be in binary form, which could preclude such analysis.

3.1. Indexed Dependence Metadata

Indexed dependence metadata defines a set of memory addresses that a component may access at a point in its iteration space. By interpreting the metadata, the component manager can map a given set of iterations onto a set of memory locations and, assuming predictable and reasonably simple patterns, can infer dependencies across sets of iterations.

In Figure 2 we see the region constraints of our convolution filter from the running example, assuming a 3×3 filter. Listing 3 shows the generic component specification for the convolution filter assuming an arbitrary-sized filter. The specification includes various pieces of metadata that the component manager can use to optimise the composition to its context. Note that omitting some or all of the metadata will not break the code; it will simply limit the scope for optimisation.

The iteration space of the component corresponds to the indices into the input image (*image_in*), as shown. For each point in the iteration space a 3×3 rectangular region of *image_in*, relative to the point, will be read. This corresponds to a *radius* of size 1 in each dimension around the point. Additionally, the whole of *filter_in* will be read and the corresponding point in *image_out* (i.e. a radius of size 0 in each dimension) will be written. The filter input variables are defined in the interface and their values propagated through the component graph.

[2]To generalise this, we can specify *inequalities* rather than *equalities* to constraints, and hence define the possible ranges for subcomponent parameters. Relaxing the requirements of a subcomponent can allow more specific and efficient subcomponents to be selected.

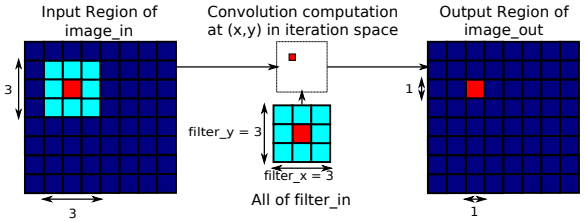

Fig. 2. Region dependencies at a point in the iteration space.

```
<component id="convolution">
  <iteration_space
    dimensions="(image_in.width,image_in.height)"
  />
  <constraint type="dependentregion"
    shape="rectangle">
    <constraintinput name="image_in"
      placement="relative"
      radius="((filter_in.w-1)/2,(filter_in.h-1)/2)"
    />
    <constraintinput name="filter_in"
      placement="absolute"
      range="(0->filter_in.w-1,0->filter_in.h-1)"
    />
    <constraintoutput name="image_out"
      radius="(0,0)" />
  </constraint>
</component>
```

Listing 3. Constraints in the specification of a component.

3.2. Component relationships through metadata

Metadata directly affects the relationships between components. If two components communicate either through a functional dependence, or through a data flow, the metadata will need to be propagated.

A component's metadata must be combined with the metadata of other components to give a full specification of a relationship. For example, in Listing 2 the contour filter requires a 3×3 convolution operation, which defines an access region on its input. The size of this access region depends on the size of the filter. Therefore, to specify fully the convolution's metadata we need to propagate the filter size specified by the contour filter through the graph. This propagation can be achieved by passing metadata bindings through parent/child and data-flow relationships.

When the application requests an interface, values are bound to the interface's parameters. These values are combined with constraints and dependence metadata throughout the component graph to bind values to variables and define component relationships as accurately as possible. Component selection or composition uses the propagated information to limit the binding of components to interfaces or to

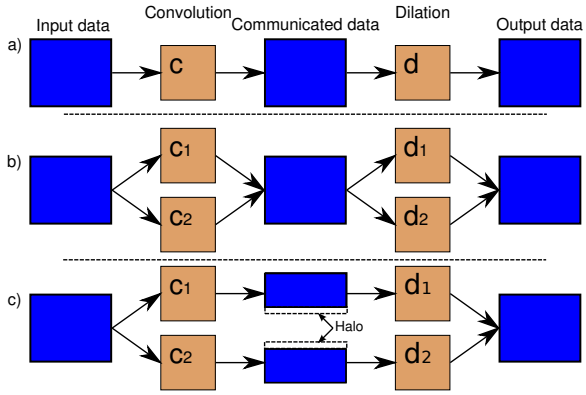

Fig. 3. The addition of region descriptors enables more efficient parallelism.

define possible composition optimisation opportunities.

Figure 3 shows how the information provided by combining region definitions with the size of the dataset can reduce the size of the required communication between two components, in this case the convolution and dilation components from Figure 1. Figure 3(a) is an example of a simple component composition communicating via an intermediate data set. If we parallelise the components with no knowledge of the components' internals, we do not know how much of the data each thread will need and must communicate it all. In this case the individual components would be parallel but not their composition, as illustrated in Figure 3(b).

With full region information we can minimise the communication between parallel components. For example, if the dilation component depends also on a 3×3 filter then parallelisation of the component as shown in Figure 3(c) requires only half the data set, plus an additional halo strip, to be sent from each convolution thread to its corresponding dilation thread. As a consequence, data can be kept in more localised, faster, memory for longer and communication is more predictable. If c_i and d_i both execute in the same memory region, only the halo strips would need to pass through higher levels in the memory hierarchy.

3.3. Scalability

The component metadata in the examples are currently written by hand. We envisage that in practice the information will be, at least partially, obtained by component analysis at construction time. Clearly, complicated components limit the feasibility of analysis. By limiting the dependence information to the input and output data structures of the component, and assuming the contents are correct, we simplify the run time workload, and improve scalability in that manner, ensuring that the complexity of individual components does not affect composition time scalability. Generation time analysis may not be possible for all components. However, the discussed system localises analysis at construction time and, as a result, increases the possibility of correct dependence construction over fully general system-wide analysis of all possible interactions.

4. CODE GENERATION

Our system supports components in various forms. In the simplest case we use a pre-compiled binary, which is linked at run time. Alternatively, we can compile and link a component code at run time. Delaying compilation to run-time offers scope for performance improvements as the compiler may have more information about the code, or the system.

A further possibility is to generate code at run time, before compilation and linking. Earlier work such as Taskgraph [4] shows that run time code generation and compilation can be effective. In this system we view both run time code generation and compilation as a lowering from one implementation level to another. For example, we can lower from a high level source representation, to C++; then through compilation of C++ to a binary. Each stage takes a component as input, and generates a replacement component as output, with correct lowered annotations. This approach is flexible and conveniently supports component caching.

We use the CLooG [2, 3] code generator to construct the code for compilation. CLooG-based components are high-level representations of iteration spaces, and are converted to C++ components in the first stage of the lowering process.

```
COMPONENT_TARGET(difference)
{
  POLYHEDRAL_LOOP(i) [ i < image1_in.height();
      i >= 0; ] {
    POLYHEDRAL_LOOP(j) [ j < image1_in.width();
        j >= 0; ] {
      image_out(x,y) = image1_in(x,y)-image2_in(x,y)
    }
  }
}
```

Listing 4. A simple polyhedral representation of the iteration space of an image difference operation.

CLooG is based on the polyhedral model [1] which represents execution schedules as polyhedra in multi-dimensional iteration spaces. CLooG's input defines a polyhedral iteration space using a set of affine half-spaces as individual inequalities in the rows of a matrix. An example of the input matrix can be seen in Figure 5(b). CLooG outputs the code necessary for each statement to visit each integer point within the polyhedron. CLooG does not perform dependence analysis and so for ill-considered input will generate incorrect output. As a result, our input to the code generator must satisfy data dependencies.

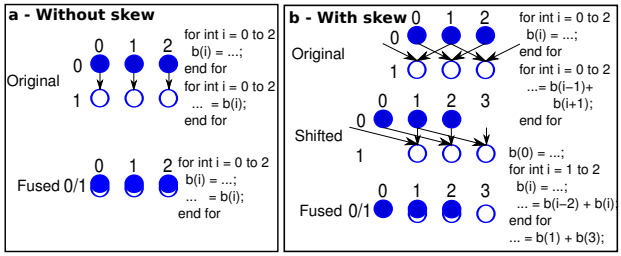

Fig. 4. A simplified one-dimensional loop fusion example.

We generate input to CLooG from a component implementation as in Listing 4. Full analysis of C code or a binary representation of analysed dependencies as polyhedra would work equally well but this syntax offers us a simple basis to work with for experimentation. We specify the execution polyhedron of the kernel using nesting to define dimensions and lists of inequalities to define ranges for the variables. This inequality syntax is converted into CLooG's input matrices during the process of lowering from CLooG input to C++. CLooG is capable of generating hundreds or thousands of lines of code to cover complicated iteration spaces which would be extremely difficult to write by hand.

5. USING METADATA FOR OPTIMISATIONS

The presence of dependence metadata on components allows the *manager* to perform component mapping decisions and, in addition, cross-component optimisations. In this work we illustrate the potential by applying loop fusion (and the enabled array contraction) to a connected subgraph of components.

5.1. Increasing temporal locality with loop fusion

Loop fusion [5] takes two or more consecutive loops and merges the bodies together as illustrated in Figure 4(a). Fusion reduces the number of control instructions, improves the temporal locality of data and, when fusing parallel loops, avoids unnecessary synchronisation (albeit with the risk of harming cache performance or instruction scheduling).

Loop dependencies can complicate fusion. In Figure 4(b) for example, statement 1 has a forward data dependence on the output of statement 0. These two statements from the same iteration number of the original loops cannot execute in the same iteration of the fused loop. The dependence can be resolved by shifting the iteration space of the second loop. The shift allows each loop to perform its given set of iterations with all dependencies satisfied before the data is required. The result of this fusion and shift (sometimes called "shift and peel" [6]) is a guarded or partially unrolled loop nest as in Figure 4(b), with a necessary loss of parallelism at the edges.

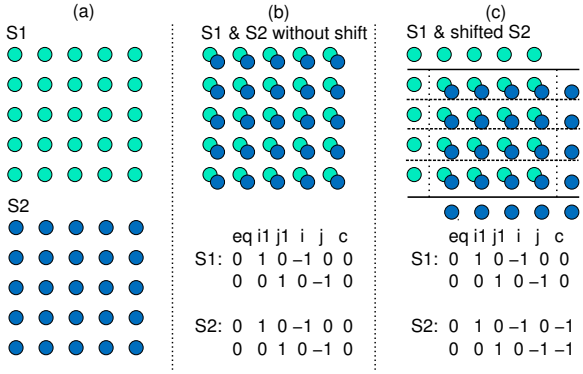

Fig. 5. The scatter matrix can be used to schedule the loop by changing the logical execution time of a given iteration.

Input and output regions defined in the metadata make the data dependencies explicit. We know which data values may be read or written at a given point in a component's iteration space, and hence can compute the shift necessary to resolve data dependencies.

We use CLooG to generate code representing the fused set of components. We supply the individual input matrices that define the iteration space. We also provide a mapping of points in the iteration space to a logical execution time, known as the *scatter* matrix. As demonstrated in Figure 5, we can specify that a point (i, j) in the iteration space (a) of a component can be mapped to (t_i, t_j) in time, where either $t_i = i$ and $t_j = j$ (b), or $t_i = i + 1$ and $t_j = j + 1$ (c), shifting the schedule.

The amount of shift required depends on the dependence relationship between two components. These relationships are computed from the access region metadata. For example, a 3×3 region as input to the second component requires a shift of 1 in the iteration space of the second component so that the output of the first is ready when it is needed. In the general case, we need to compute the last iteration in the source component that may generate data needed by the matching iteration in the target component. If the dependence distance is constant, we can compute a static schedule correction. We parameterise the scatter matrix by a set of shift values computed from the dependence relationships to shift the logical time of the component and therefore of its statements. With a correct scheduling defined in the scatter matrix, CLooG will generate a series of loops that respects the inter-component data dependencies.

Component selection for fusion depends on the flow of data between components. Unrelated components are easy to fuse, but unlikely to benefit from fusion. Components that share inputs, or communicate using a intermediate data structure, are more likely to benefit. Having analysed the data flow in the parent component at construction time, we can fuse the children at composition time. Calls to the sub-

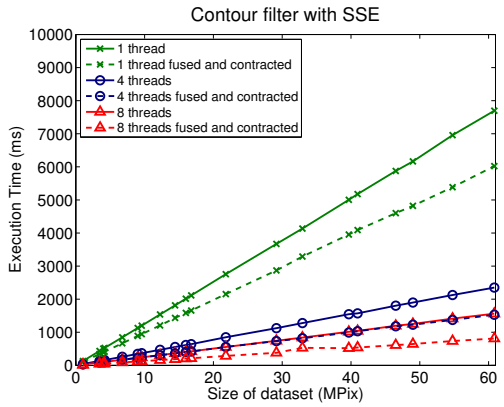

Fig. 6. Execution time of the contour filter example.

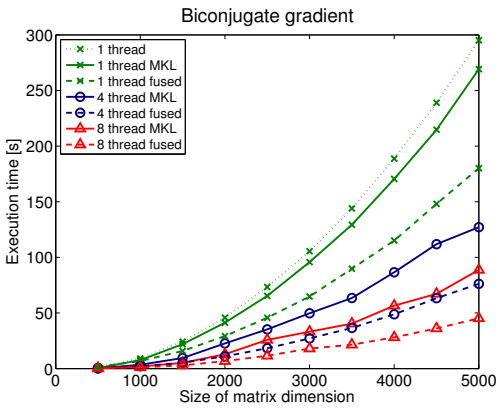

Fig. 7. Comparing MKL, with custom version of biconjugate gradient for 1, 4 and 8 threads (custom without fusion shown only for 1 thread).

components can be replaced with calls to stub functions that merely prepare data structures. The execution of the fused component can be delayed until the last subcomponent call. As a result, the parent component itself need not change.

5.2. Reducing storage through contraction

Loop fusion reduces the period between generation and use of intermediate data values, often leading to more efficient use of the cache and improved performance. Array contraction offers further scope for improvements and can be a key enabler of high performance in large parallel fused loops [7]. Rather than storing entire intermediate arrays, we reduce the intermediate storage to the minimum required to satisfy data flow requirements, reducing the use of memory bandwidth due to cache displacement.

6. EXPERIMENTAL RESULTS

We implement three examples using our component framework to demonstrate its capabilities and how we can improve the performance of an application. These examples possess different data flow situations and hence show varied performance.

To enable fusion, all subcomponents are implemented in a high-level polyhedral representation, as in Listing 4, and have appropriate dependent region and data flow metadata attached to describe the relationships between component inputs and outputs.

We compile using Intel C/C++ 10.1 or GCC 4.2 (whichever performs better) on an eight core, dual-socket Intel Xeon based machine running a 64-bit Linux 2.6 kernel and parallelise using OpenMP. The single threaded code is the unparallelised, sequential version.

6.1. Image processing

The contour filter (Figure 1) operates on four-component (RGBA) data and is vectorised using SSE instructions.

The dilation subcomponent selects a maximum value in a region of the output of the convolution subcomponent. To allow for this dependence, the fused execution space must shift. The execution of the elements of both the dilation and difference are delayed by the radius of the region.

Figure 6 shows performance results for the contour filter with SSE. There is a substantial reduction in execution time for fusion combined with contraction. Execution time is reduced by 21% for a single thread, 35% for four threads and 48% for eight threads. While not plotted on the graphs, fusion alone offers 4%, 11% and 20% respectively. The improvement from fusion alone is slightly erratic, but tends to decrease with data set size as the larger range of visited addresses increases the chance of an individual element being removed from the cache. A similar effect is not seen with the contracted data sets where the accessed address range is reduced to a circular buffer of a few image rows in size.

6.2. Linear Algebra

Our linear algebra example is a biconjugate gradient solver from the Iterative Template Library [8], with components defining various aspects of the computation flow. We allow fusion to occur between a standard matrix/vector multiplication, and a transposed matrix/vector multiplication. Note that in this benchmark we share input matrices between components, rather than having data flow from one component to another. A result of this lack of data flow is that there is no communicated array to contract and hence this example supports fusion only.

In this example we use 1×1 access regions because the execution maps a single iteration space point to a single data

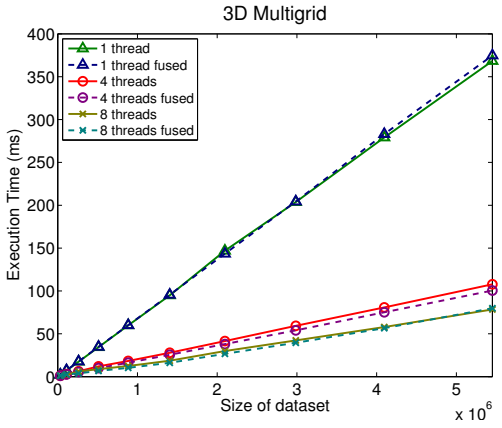

Fig. 8. Execution time for single and four and eight threaded 3D multigrid solver kernel.

element from each input. As the input and output vectors are present in a single dimension only, the mapping is a projection onto that dimension.

Figure 7 graphs performance results for the fused versions of the biconjugate gradient solver as well as results for Intel's Math Kernel Library (MKL) [9] as a baseline with a comparison with the original version on a single thread. We can see that while there is an improvement in performance over MKL for all numbers of threads, this improvement is more pronounced for 4 and 8 threads where memory contention between cores is reduced by fusion.

6.3. 3D Multigrid

Multigrid solves differential equations using a hierarchy of discretisation levels. We adapted this example from the NAS Parallel benchmarks suite [10] using fixed boundary conditions.[3] We created a sequence of dependent components based on the core functions that iterate on the data: *Data initialisation*, *Interpolation from a lower resolution computation stage*, *Computation of residuals* and *Application of a smoother to the data*.

The four components are related by region and data flow dependencies describing how a value in the iteration space of one component relates to a value in the iteration space of the next in sequence. We require $3 \times 3 \times 3$ regions around the input to the interpolation, residual and smoother applications. We make the kernel more efficient by absorbing the inner dimension of the loop nest, allowing hand tuning of the inner loop. Given such a kernel, our access region specifies an entire row of the data set in one dimension and a 3×3 region in the other two. Note that the component manager need not know that our tuned kernel has a carefully written inner loop, only that it needs to access an entire row of the data set to perform its work.

Figure 8 offers performance results for 1, 4 and 8 threads. The improvement from fusion peaks at 4 threads where we see a mean reduction in execution time of 12% over the range shown. For larger data sets the performance of fusion falls off as the amount of data maintained by the 3D computation skew creates stress on the cache and other shared data structures of the CPU. The peak at 4 threads is similarly explained because the L2 cache is shared between pairs of cores, reducing the effective cache size per core when 8 threads are used.

[3] In the original code the computation is complicated by a cyclic dependency due to a wrap-around boundary condition. While fusion is still possible with the cyclic dependency, performance benefits are lost due to the increased loop shift necessary to support wrapping on all three dimensions.

7. RELATED WORK

Adaptive component models have been widely studied, for example in embedded systems (e.g. see [11]), as well as more generally in distributed systems (e.g. [12]). Dowling and Cahill [13] offer a useful framework, emphasising the importance of separating adaptational from computational code. Recent work on the Common Component Architecture (CCA) looks at composing, substituting and reconfiguring components during application execution [14].

Our component composition builds on work on Architecture Description Languages (ADLs) such as Darwin [15] and xADL [16], and is similar to Think [17]. Our work differs from other ADLs in its support for indexed dependence metadata, that denotes dependence relationships for individual iteration space points.

CLooG arises from Bastoul's work [3] and builds on earlier work on code generation in the polyhedral model by Griebl and Wetzel [1]. Griebl [18] applies the polyhedral model to parallelisation of loop nests while recent work by Pop et al. [19] looks at integrating polyhedron based analysis into GCC.

This paper is an attempt to realise the THEMIS [20] proposal and is part of a larger body of work including the Task-graph [4] library, related work from Cornwall [7] and active libraries in linear algebra from Russell [21].

ZPL [22] (a precursor of Chapel [23]) and KeLP [24] (which led to Chombo [25]) had explicit regions - in fact a "region calculus". However their regions represent partitions of iteration and data spaces - whereas in this work we represent the mapping between points in the iteration space and memory locations.

Languages like StreamIt [26] use the concept of sequences of data items, called *streams*, which are operated on by pure functions, called *filters*. Clear (and often static) data-flow relationships between filters enable cross-component optimisations. In contrast, our framework enables cross-component optimisations for general programs operating on arbitrary data sets.

8. CONCLUSIONS AND FUTURE WORK

We have shown how interfaces with indexed dependence metadata can be used to improve the performance of component compositions. Our experimental results show that metadata can be used to perform aggressive component fusion, generating hundreds of lines of code (200-300 in the contour filter and over 1500 for the multigrid example) that would be challenging to implement by hand. We have also confirmed that loop fusion can substantially reduce execution time through improvements in temporal locality of data.

The THEMIS proposal discusses more possibilities for metadata than we have been able to implement to date. In the future we hope to proceed further with this investigation, particularly in the area of applying cross-component optimisation techniques to data layout by adding metadata annotations describing the access patterns for data. More varied access descriptors and tighter integration into the programming language using C++ pragmas or compiler support for iterator classes are other targets.

The multigrid example shows that in some cases fusion gives only a small benefit. In these cases we plan to use adaptive component mapping to use the original components rather than fused sets when a fusion attempt reduces performance. Optimal combinations may include calls to vendor libraries wrapped in components, as used in the MKL comparison for the linear algebra example.

Novel architectures such as heterogeneous multicore platforms require novel optimisation strategies. Hand coding is often impractical. We envisage that adaptive, metadata-driven optimisation techniques will be of increasing relevance as technology develops.

9. REFERENCES

[1] M. Griebl, C. Lengauer, and S. Wetzel, *Code generation in the polytope model*, Proc. PACT, IEEE Comp. Soc., 1998.

[2] *CLooG*, http://www.cloog.org/.

[3] C. Bastoul, *Code generation in the polyhedral model is easier than you think*, Proc. PACT, IEEE Comp. Soc., 2004.

[4] O. Beckmann, A. Houghton, P. H. J. Kelly, and M. Mellor, *Run-time code generation in C++ as a foundation for domain-specific optimisation*, Proc. Domain-Specific Program Generation International Seminar, 2003.

[5] K. Kennedy and K. S. McKinley, *Maximizing loop parallelism and improving data locality via loop fusion and distribution*, Proc. LCPC, Springer, 1994.

[6] N. Manjikian and T. S. Abdelrahman, *Fusion of loops for parallelism and locality*, IEEE Trans. Parallel Distrib. Sys.

[7] J. L. T. Cornwall, P. H. J. Kelly, P. Parsonage, and B. Nicoletti, *Explicit dependence metadata in an active visual effects library*, Proc. LCPC, Springer, 2007.

[8] A. Lumsdaine, L.-Q. Lee, and J. Siek. *Iterative template library*, http://www.osl.iu.edu/research/itl/, 2001.

[9] Intel. *Math Kernel Library*, 2008

[10] B. L. Chamberlain, S. J. Deitz, and L. Snyder, *A comparative study of the NAS MG benchmark across parallel languages and architectures*, Proc. SC, IEEE Comp. Soc., 2000.

[11] H. Ma, I.-L. Yen, F. Bastani, and K. Cooper, *Composition analysis of QoS properties for adaptive integration of embedded software components*, Proc. ISSRE, 2003.

[12] L. Baresi, S. Guinea, and G. Tamburrelli, *Towards decentralized self-adaptive component-based systems*, Proc. SEAMS, ACM, 2008.

[13] J. Dowling and V. Cahill, *The k-component architecture meta-model for self-adaptive software*, Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Springer, 2001.

[14] L. C. McInnes et al, *Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration*, Argonne Nat. Lab., Tech. Rep. 2006

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying distributed software architectures*, Proc. European Software Engineering Conference, Springer, 1995.

[16] E. Dashofy, A. van der Hoek, and R. Taylor, *A highly-extensible, XML-based architecture description language*, Software Architecture, 2001.

[17] A. E. Özcan, O. Layaida, and J.-B. Stefani, *A component-based approach for MPSoC SW design: Experience with OS customization for H.264 decoding*, ESTImedia, IEEE Comp. Soc., 2005.

[18] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*, Habilitation Thesis, University of Passau, 2004

[19] S. Pop, G.-A. Silber, A. Cohen, C. Bastoul, S. Girbal, and N. Vasilache, *GRAPHITE: Polyhedral analyses and optimizations for GCC*, Proc. GCC Summit, 2006.

[20] P. Kelly, O. Beckmann, A. J. Field, and S. Baden, *THEMIS: Component dependence metadata in adaptive parallel computations*, Parallel Processing Letters, 2001

[21] F. P. Russell, M. R. Mellor, P. H. J. Kelly, and O. Beckmann, *An active linear algebra library using delayed evaluation and runtime code generation*, Proc. LCSD, 2006.

[22] B. L. Chamberlain, E. C. Lewis, C. Lin, and L. Snyder, *Regions: an abstraction for expressing array computation*, SIGAPL APL Quote Quad, 1998.

[23] B. Chamberlain, D. Callahan, and H. Zima, *Parallel programmability and the chapel language*, Int. J. High Perf. Comp. Appl., 1997.

[24] S. J. Fink, S. B. Baden, and S. R. Kohn, *Efficient run-time support for irregular block-structured applications*, J. Parallel Distrib. Comp., 1998

[25] P. Colella et al. *Performance and scaling of locally-structured grid methods for partial differential equations*, SciDAC 2007 Annual Meeting.

[26] W. Thies, M. Karczmarek, and S. Amarasinghe, *StreamIt: A Language for Streaming Applications*, Proc. Compiler Construction, Springer, 2002.