Automated Transformation for Performance-Critical Kernels

Qing Yi
University of Texas at San Antonio
qingyi@cs.utsa.edu

R. Clint Whaley
University of Texas at San Antonio
whaley@cs.utsa.edu

ABSTRACT

The performance of many scientific applications depends on a small number of key computational kernels which require a level of efficiency rarely satisfied by existing native compilers. We present a new approach to high performance kernel optimization, where a general-purpose transformation engine automates the production of highly efficient library routines. The library routines are then empirically tested until an implementation with a satisfactory performance level is found. Our framework requires an annotated kernel specification and can automatically produce optimized implementations based on tuning parameters controlled by a search driver. The transformation engine includes an extensive suite of optimizations which can be easily expanded using a custom transformation language. We have applied our framework to generate code for key linear algebra kernels and have achieved similar performance as that achieved by ATLAS's highly tuned kernels. In several cases, our kernels were faster than ATLAS's native kernels; we have made these kernels available to ATLAS, which results in speedups for the ATLAS library, as we show.

1. INTRODUCTION

There are more than a few application areas where performance needs are not fully addressed by current compilation techniques, either because the compiler lacks domain-specific knowledge about the application, or because the compiler cannot fully address the extreme complexity of modern computer architectures. To overcome this problem, many applications rely on performance-critical libraries which have been hand-tuned (often directly in assembly) for each architecture of interest. For a few computational libraries, there exist empirical tuning frameworks that can automate this tuning process, as in ATLAS [27, 25] and FFTW [10, 20], among others. The demand for such well-tuned library routines has led to several application-specific empirical tuning frameworks where both domain-specific knowledge and direct timings are used to guide the optimization of important kernel implementations [7, 11, 17, 22, 24, 2, 4].

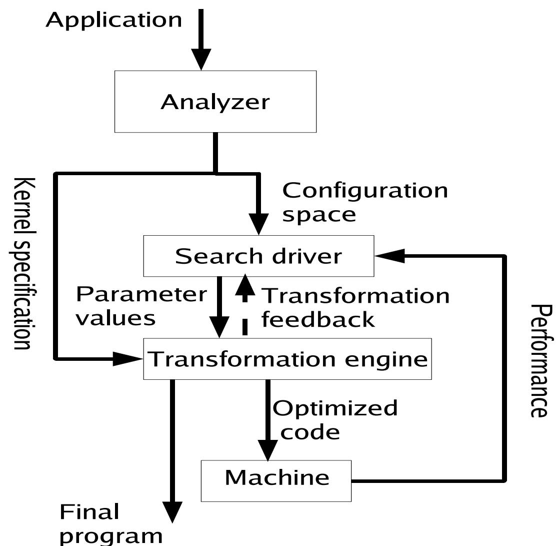


Figure 1: Our empirical tuning approach

Despite the success of many domain-specific empirical tuning systems, there are limits to the generality and portability of this approach. Since these frameworks require significant investment to create, and are typically not as effective when the problem at hand deviates from their main domain, many computational kernels beyond their domain are not well supported and thus do not achieve adequate performance.

This paper presents a new approach, where a general-purpose framework is proposed to automate the production of highly-optimized library kernels. As shown in Figure 1, our framework includes three components: an analyzer, a transformation engine, and an empirical search driver. The analyzer role could be performed manually by a programmer or automatically with a source-to-source compiler. Since the analyzer understands the computational kernel and knows what transformations should be investigated to improve performance, such information is expressed in a *kernel specification* file. On each platform that the library routine needs to be tuned for, the kernel specification is used as input to

a transformation engine, which works together with a customized search driver to automatically search the transformation space in finding a highly optimized kernel implementation. This paper focuses on presenting our POET transformation engine which we have used for optimizing floating point kernels.

A key feature of our approach is the portability and extensibility of our transformation engine (TE). Additionally, the TE is language neutral and can be easily specialized to produce optimized kernels in an arbitrary source language, including C, C++, FORTRAN, or assembly. Compared to the conventional domain-specific empirical tuning frameworks, our approach has the following advantages:

- First, our approach is targeted at producing high performance implementations for general-purpose library routines. The transformation engine includes an extensive library of code optimizations that can significantly improve application performance. The analyzer needs to only specify where to apply these transformations in order to extract high performance for an arbitrary computational routine. The kernel specification can then be empirically tuned on different machines without re-applying any of the required program analysis.
- Second, our transformation engine supports the parameterization and re-configuration of all the relevant code optimizations, so that a single version of kernel specification can be used to generate different implementations and empirically tuned to find the best implementation. In contrast, although most domain-specific empirical tuning frameworks parameterize their kernel implementations to ensure portability, their parameterization is often limited to variations specific to a single or small set of kernels, and thus are not as comprehensive or as easy to extend.
- Finally, our transformation engine is based on an embedded scripting language, POET, which can be used by library developers to easily define their own customized algorithm-specific optimizations. A kernel specification file can then be similarly written to invoke these optimizations. Therefore, using our approach, both the transformation engine and the kernel specification file are readily extendible to incorporate new architecture features and domain-specific knowledge. As a common case, a source-to-source compiler can be first employed to automatically produce a kernel specification file based on results of program analysis, the generated POET scripts can then be manually modified or extended by library developers to incorporate domain-specific optimizations.

Our work is related to the research of empirical tuning compilers, which iteratively re-configure architecture-sensitive optimizations according to the performance feedback of their optimized code. This body of research has demonstrated that empirical tuning of application performance can significantly improve the effectiveness of compiler optimizations [28, 19, 15, 16, 21, 12, 32, 5]. Our approach can significantly improve the flexibility, efficiency, and extendibility of existing iterative compilers. Specifically, by providing POET kernel specifications as the output of optimizing compilers, applications need to be analyzed only once for optimization, and the resulting code transformations can be empirically tuned many times on arbitrary computing platforms. Optimizing compilers do not need to reside on the machines where the applications will be ported, and both library developers and independent search drivers can work directly on the POET output from compilers to modify the optimization configurations as well as to add their own customized transformations.

To demonstrate that our transformation engine can produce kernel implementations as efficient as those produced by domain-specific libraries, we have applied our framework to generate highly optimized code for several of the most important and representative linear algebra kernels used in the ATLAS library [27, 25]. In this work, we have manually written the kernel specification (the library routine to optimize plus markup indicating what transformations to investigate, see Figure 5 for an example). Our future work will extend an optimizing compiler [29] to automate this process. Section 4 presents a comparison of our POET kernels with those produced by the ATLAS framework. Our POET-tuned kernels achieve similar levels of performance as those tuned by ATLAS, winning in some cases and losing in others, but competitive across the spectrum. We also present data showing that even with such a highly tuned library, POET can provide further speedup. Therefore these results provide strong evidence that that our POET code transformation engine can achieve portable high performance for general-purpose kernels, while requiring significant less time and effort than hand-tuning the routines.

2. THE TRANSFORMATION ENGINE

Our transformation engine (TE) is based on a small special-purpose language named POET (Parameterized Optimizations for Empirical Tuning) [30]. The POET language is designed to specifically support parameterized code generation for empirical tuning and includes sophisticated features to support easy definition of arbitrary customizable code transformations. Our TE has used POET to support an extensive code transformation library, an annotation interface for parsing and representing arbitrary computational routines, and a programming interface for applying different code optimizations to the kernel computation. POET can also be used to implement customized search drivers for the empirical tuning of arbitrary kernel implementations. This paper focuses on how to use the POET TE to automatically produce high-performance kernel implementations.

As shown in Figure 2, our transformation engine includes three components: a POET language interpreter, a transformation library, and a collection of front-end definitions which specialize the transformation library for different programming languages such as C, C++, FORTRAN, or Assembly. In the center of the TE is the POET language interpreter, which takes as input a kernel specification from an analyzer and a collection of parameter values from a separate search driver, invokes a specialized language frontend

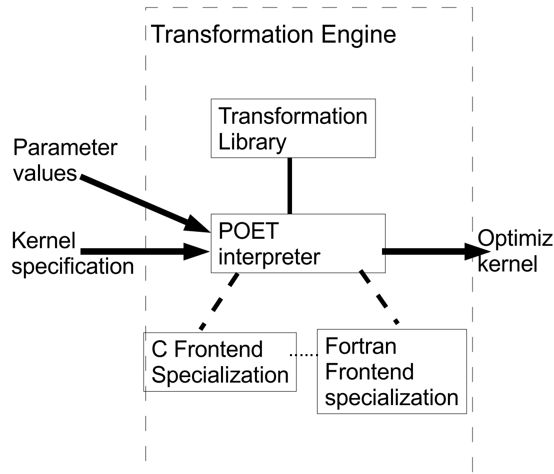


Figure 2: POET transformation engine

to help parse the input computation, and then invokes the transformation library to optimize the kernel implementation. An optimized kernel implementation is output as the result, which is then empirically tested and measured by the search driver until a satisfactory implementation is found.

In order to build an optimized kernel implementation, the analyzer needs to only provide a kernel specification, which invokes a specialized language frontend to parse the input computation and then invokes transformation routines from the TE library to optimize the computation. Both the TE library and the language frontend can be used without detailed knowledge about their implementation. Additionally, developers can easily expand the TE library with domain-specific customized transformations. In the following, Sections 2.1 and 2.2 first briefly introduce our existing frontend and transformation library. Section 2.3 then focuses on how to use the POET TE to build optimized kernels for empirical tuning.

2.1 Frontend Specialization

Our transformation engine is language neutral in that both POET and the TE library are independent of what language that the input computation is coded in. POET is a scripting language which can be embedded in an arbitrary source language and treats code fragments in the source language as strings wrapped inside a collection of customized abstract syntax tree (AST) definitions called "code templates". Figure 3 shows some examples of code templates defined for optimizing kernels written in C. Each POET code template conveys a special meaning and serves to present an abstract view of the input computation to the TE library, which applies transformations to the code templates without knowing how the code templates are defined.

POET code templates are compound data structures which are used by the transformation engine to provide an internal representation of the input computation. As shown in

```
<code Exp pars=(str)>
@str@
</code>

<code Stmt pars=(str) >
@str@;
</code>

<code ArrayRef pars=(arr,sub) >
@arr@[@sub@]
</code>

<code PtrRef pars=(ptr)>
*(@ptr@)
</code>

<code Assign pars=(lhs, rhs)>
@lhs@ = @rhs@
</code>

<code Function pars=(head,body)>
@head@
{
  @body@
}
</code>

<code Loop pars=(i,start,stop,step)  attr=(maxiternum)>
@for (@i@=@start@; @i@<@stop@; @i@+=@step@)
</code>

<code Nest pars=(loop, body)>
@loop@ {
   @body@
}
</code>

<code Sequence pars=(s1,s2) >
@s1@
@s2@
</code>
```

Figure 3: C frontend specialization

Figure 3, each code template can have two attributes, *pars* and *attr*, which define the parameters and additional properties of the source code. The concrete syntax of each code template is then defined in a programming language such as C and is parameterized by variables declared in *pars* (in Figure 3, the reserved token, '@', is used for context switching between POET parameters and source strings of the underlying language). As an example, Figure 3 includes several code templates defined for the C language. These code templates are used in the definition of the loop blocking transformation in Figure 4 and are used to parse the matrix multiplication kernel given in Figure 5.

Code template specializations like those shown in Figure 3 are used both for parsing the input source and for emitting the transformed output. The POET TE library uses these templates as internal representations of the input computation without knowing how these representations are defined in the input code. When an input program is defined in terms of code templates, generic routines predefined in our TE library can recognize the structure of the input program and apply optimizations accordingly. The definition of code

```
<xform Stripmine
       pars=(inner,bsize,outer)
       tune=(unroll=0,split=0)
       output=(_nvars, _bloop, _tloop,_cloop,_body)>
  switch outer {
     case inner : ("","","","",inner)
     case Loop#(i,start,stop,step): ......
     default:  ......
  }
</xform>

<xform BlockHelp
       pars=(bloop,tloop,rloop,bbody,cbody,cloop)>
  if (bloop=="") ... <*base case*>...
  else { ...<*recursively call BlockHelp*>... }
</xform>

<xform BlockLoops
       pars=(inner,outer,decl,input)
       tune=(bsize=16,split=0,unroll=0)>
  ...=Stripmine[unroll=::unroll,split=::split]
                (inner,bsize,outer);
  ... call BlockHelp ...
  ... modify input ...
</xform>
```

Figure 4: Skeleton of Loop blocking as defined in the transformation library

templates therefore serves to specialize the TE library to a specific programming language. To process kernels implemented in a language other than C, the programmer only needs to switch to another predefined code template header file. The POET transformation engine can therefore be used to optimize kernels in different languages without significant adaptation.

2.2 The Transformation Library

Our TE library includes an extensive collection of code optimizations that have proven to be able to significantly improve application performance, including loop transformations such as loop blocking, interchange, fission, fusion, unroll-and-jam, unrolling, splitting; memory optimizations such as array copying and scalar replacement; as well as low-level optimizations such as strength reduction and SSE vectorization. All transformations are implemented using POET, a high-level scripting languages with an xml-like syntax. Figure 4 shows a few skeletons of POET routines relevant to applying a loop blocking transformation. Table 1 lists a subset of the POET operations used in the examples.

As shown in Figure 4, POET uses the keyword *xform* to define routines that can be invoked to transform input code fragments. Each *xform* routine uses the *pars* attribute to define a sequence of function parameters, uses the *tune* attribute to define tuning parameters which can be used to reconfigure the transformation (each tuning parameter has a default value which defines the default configuration), and uses the *output* attribute to define return values of the $xform$ routine. The body of each *xform* routine examines the input parameters and returns a new code fragment which replaces the original one. Additional information may be returned when the *output* attribute is defined.

The entire TE library comprises *xform* routines as shown in Figure 4. These routines can apply pattern matching to the input code fragment (e.g., the *switch* statement in the *Stripmine* routine tries to match the *outer* parameter against different code templates), use complex control-flow support such as conditionals and loops, and make recursive function calls. Each routine can build new fragments using compound data structures such as lists, tuples, hash tables, and predefined code templates. The full programming support for defining customized transformations distinguishes POET from most other existing special-purpose transformation languages, which rely on template- or pattern-based rewrite rules to support definition of new transformations.

POET *xform* routines can be separated into two categories: internal routines such as *Stripmine* and *BlockHelp*, which are helper routines used by other facilities within the library; and interface routines such as *BlockLoops*, which can be invoked directly from a kernel specification file. Programmers need only be aware of the syntax and semantics of interface routines when defining the kernel specification for an input application.

We choose to use POET to implement our TE library because using a scripting language is orders of magnitude easier than using general-purpose languages such as C/C++ in writing dynamic code transformation routines. In addition to supporting common language features such as loops and recursive functions, POET has a special focus on program transformation by supporting easy construction and manipulation of code fragments in a customized internal representation. The extensive support for building code transformations in POET allows programmers to easily extend the TE library with their own *xform* routines.

Most of the code transformations in our TE library are also typically included in optimizing compilers, where the routines would be part of the compiler implementation and written in C/C++ (or whatever language the compiler is implemented in). In essence, we have implemented many of the conventional compiler transformations using the POET language and have provided these transformations as a library as a part of a comprehensive programming support for building extremely optimized kernel implementations. A source-to-source optimizing compiler can then be employed to automatically produce invocations to the library routines to optimize an input code. Through the POET language, our transformation engine provides a flexible interface for programmers both to extend the TE library with additional customized optimizations and to extend the kernel specification with additional optimizations based on domain-specific knowledge. The additional empirical tuning support by the transformation engine also makes POET-optimized kernels much more portable than hand-written assembly.

2.3 Kernel Specifications

The main input of POET transformation engine is a kernel specification file which includes two components: an input specification, which defines the input computation to be tuned as a kernel; and a transformation specification, which defines where and how to apply various parameterized trans-

```
<input gemm>
//@; BEGIN(gemm)
void ATL_USERMM(const int M, const int N, const int K,
          const double alpha, const double *A, const int lda,
          const double *B, const int ldb, const double beta,
          double *C, const int ldc)                          //@=>_:Exp
{                                                            //@; BEGIN(_)
   int i, j, l;                                              //@=>gemmDecl:Stmt; BEGIN(gemmBody)
   for (j = 0; j < N; j += 1)                                //@ =>loopJ:Loop BEGIN(nest3)
   {                                                         //@; BEGIN(body3)
      for (i = 0; i < M; i += 1)                             //@=>loopI:Loop BEGIN(nest2)
      {                                                      //@;BEGIN(body2) BEGIN(parse)
         C[j*ldc+i] = beta * C[j*ldc+i];                     //@END(parse) =>_:Stmt
         for (l = 0; l < K; l +=1)                           //@=>loopL:Loop BEGIN(nest1)
         {                                                   //@;BEGIN(parse)
            C[j*ldc+i] += alpha * A[i*lda+l] * B[j*ldb+l];   //@END(parse) =>stmt1:Stmt
         }                                                   //@END(nest1:Nest) END(body2:Sequence)
      }                                                      //@END(nest2:Nest) END(body3:Nest)
   }                                                         //@END(nest3:Nest) END(gemmBody:Nest) END(_:Sequence)
}                                                            //@END(gemm:Function)
</input>
```

Figure 5: Input specification for kernel *dgemm*

formations to the input code. For example, Figure 5 shows the POET input specification for *dgemm*, the matrix multiplication kernel from the ATLAS library [27, 25], and Figure 6 shows the transformation specification for the kernel.

Input Specification. In order to optimize a computational kernel, the POET transformation engine needs to parse the input code and translate it into an internal code template representation which can be understood by the TE library. Figure 5 illustrates an input specification for the ATLAS *dgemm* routine, where fragments of the the input code are annotated with information to help parse the matrix computation into a code template representation. Each POET annotation either starts with "//@" and lasts until the line break, or starts with "/*@" and ends with "@*/". Programmers can embed these annotations as comments in their C/C++ code, where the source code of the computational routine is readily accessible for both readability and easy maintenance of the kernel implementation.

POET supports both single and nested template annotations. A single template annotation starts from the end of the last annotation and ends with an annotation in the format "=> x : T", where x is the name of a global variable that will be used to store the result of parsing the code fragment, and T is the code template that should be used to parse the annotated code. For example, in Figure 5, the annotation "void ATL_USERMM(...const int ldc) //@=>_:Exp" indicates that the entire source string "void ATL_USERMM(...const int ldc)" should be treated as the content of a single expression as defined by the *Exp* code template, and the variable name "_" indicates that the code fragment does not need to be stored in any global variable. Similarly, the annotation " int i, j, l; //@=>gemmDecl:Stmt" indicates that "int i, j, l;" is a statement that should be parsed using the *Stmt* template, and the result should be stored in the global variable *gemmDecl*. The definitions for both *Exp* and *Stmt* can be found in Figure 3.

In contrast to single template annotations, nested annotations in POET are used to help parse compound language constructs such as functions and loop nests, which include other code fragments as components. Each nested POET annotation starts with "BEGIN(x)", where x is the variable that should be used to store the compound code template, and ends with "END(x:T)", where T is the name of the code template that should be used to parse the annotated code. In Figure 5, the annotation "for (l = 0; l < K; l += 1) //@ =>loopL:Loop BEGIN(nest1) ... END(nest1)" is a nested annotation which starts with the *for* loop (a singly annotated fragment stored in *loopL*) and ends after parsing the loop body *stmt*1. Other nested annotations in Figure 5 include code fragments stored in *gemmBody*,*nest*3, *nest*2, *body*2,etc. The special nested annotation "BEGIN(PARSE) ... END(PARSE)" indicates that the built-in POET expression parser should be used to parse the enclosed code fragment, where appropriate code templates for parsing have been pre-defined in the frontend specialization of the POET TE.

The input specification as illustrated in Figure 5 is necessary so that the POET language interpreter can parse the input computation correctly without being source language specific (note that much of this could be handled automatically by a source-to-source analyzing compiler). Because each code template used in parsing the input code can alternatively be defined using a different programming language, the POET TE can be easily specialized to optimize code written in different source languages such as C or FORTRAN without requiring a parser for each language. We have designed the annotation syntax to minimize intrusion to the source code, so that if written in C, POET annotations can be treated merely as comments, and the source code can be compiled with a regular C compiler without requiring any additional bookkeeping.

```
<parameter SSELEN=16, SSENO=16 />
<parameter mu=6, nu=1, ku=36, NB=36, MB=36, KB = 36, PF=1 />
<trace nest3,loopJ,body3,nest2,loopI,body2,
       nest1,loopL,stmt1,gemm,gemmDecl,gemmBody/>
<define Specialize DELAY { if (SP) {
    REPLACE("N",NB,loopJ); REPLACE("M",MB,loopI); REPLACE("K",KB,loopL);
    REPLACE("lda",MB, gemmBody);  REPLACE("ldb",NB, gemmBody);
    if (alpha == 0) {  REBUILD(REPLACE("alpha",1, gemmBody)) }
} } />
<define nest3_UnrollJam  DELAY { if (mu > 1 || nu > 1) {
    UnrollJam[factor=(nu mu)](nest1,nest3,gemmBody);
} } />
<define nest1_Unroll  DELAY {  if (ku > 1) {
     UnrollLoops[factor=ku](stmt1,nest1,body2);
} }/>
......
```

(a) transformation definitions

```
<output dgemm_kernel.c (
   TRACE gemm;
   APPLY Specialize;
   APPLY A_ScalarRepl;
   APPLY nest3_UnrollJam;
   APPLY B_ScalarRepl;
   APPLY C_ScalarRepl;
   APPLY array_ToPtrRef;
   APPLY Abuf_SplitStmt;
   APPLY body2_Vectorize;
   APPLY array_FiniteDiff;
   APPLY body2_Prefetch;
   APPLY nest1_Unroll;
   gemm
) />
```

(b) output definition

Figure 6: Defining transformations for kernel *dgemm*

f [tuneArgs](inputArgs)	invoke the *xform* routine *f* with arguments for both the input and tuning paramters
DELAY exp	delay the evaluation of a POET block *exp* until later
APPLY exp	force the evaluation of a delayed POET block *exp*
REPLACE(c1,c2,input)	replace the code fragment *c*1 in *input* with *c*2
REBUILD exp	rebuild a code fragment *exp* after modifications to it
TRACE exp	start tracing all the pre-declared *trace* variables in the code fragment *exp*

Table 1: A subset of built-in operations in POET

Transformation specifications. After the input specification is processed by a POET language interpreter, an internal representation of the given kernel computation is constructed and stored in a collection of global variables. The programmer can then invoke the POET TE library to optimize the input code. Figure 6 illustrates some of the transformation specifications for optimizing the *dgemm* kernel in Figure 5. These transformation specifications include four different kinds of POET declarations: *parameter*, *trace*, *define*, and *output*, for defining and manipulating the global variables used to store the input computation. The set of the operations used in Figure 6 is defined in Table 1.

In POET, each keyword *parameter* declares a number of global variables that can be used to re-configure transformations applied to the input code. The values of these parameters can be set from command line by an independent search driver when the transformation engine is invoked, which allows the search driver to generate different kernel implementations for empirical tuning. The *parameter* declarations therefore serve as the communication interface between the transformation engine and the search driver.

Similar to the *parameter* declaration, each keyword *trace* serves to declare global variables which can be embedded inside the input computation to keep track of selected code fragments as they go through a sequence of transformations. In Figure 6(b), the TRACE operation inserts several *trace* variables, *gemmDecl*,*gemmBody*,*nest*3, *nest*2,and *nest*1, into *gemm*, the global variable which stores the internal representation of the input code. As various transformations are applied to optimize the input code, the values of these trace variables are replaced with equivalent code fragments which may display better performance. In Figure 6, the input code is optimized by applying 11 different transformations, each transformation can operate on the *trace* variables without worrying about what transformations have already been applied. The tracing capability therefore makes the ordering of different code transformations extremely flexible, and the programmer can easily adjust transformation orders and even determine the best ordering through empirical tuning if desired.

Each keyword *define* in POET serves to assign new values for global variables. At each assignment, the target code fragment is first evaluated and the result is then assigned as the new value of the variable. If the value of a global variable is a code transformation, the evaluation of the transformation can be delayed using the DELAY operation, which packages the code fragment until an APPLY command is invoked, which forces the evaluation of delayed transformations. Figure 6 illustrates the definition of three code transformations, *Specialize*, which specializes the input code by substituting constant values as bounds for loops; *nest3_UnrollJam*, applies unroll-and-jam transformation to *nest*3; and *nest1_Unroll* applies loop unrolling to *nest*1. Predefined transformation routines are invoked within these definitions, where REPLACE and REBUILD are built-in functions within the POET language, and *UnrollAndJam* and *UnrollLoops* are routines defined in the TE library. The invocations to both *UnrollAndJam* and *UnrollLoops* have redefined their tuning parameters based on the reconfigurable parameters of the kernel specification.

Finally, the *output* declaration in POET defines what code should be output to external files. The *output* declaration in Figure 6 first applies a sequence of transformations to the input code and then outputs the optimized code. A trans-

formation specification can define multiple code fragments to output to different files so that multiple implementations can be simultaneously produced by the transformation engine.

2.4 Optimizing Kernel Implementations

The goal of our transformation engine is to support compact description of both parameterized code optimizations and how these optimizations can be applied differently to improve the performance of input applications. We have carefully designed our framework to offer strong support for the following capabilities:

- Generic transformations can be easily defined and applied to optimize arbitrary application codes. In addition to an extensive library of predefined code optimizations commonly adopted by compilers, library developers can use POET to readily define their own customized code transformations.
- Important properties and special semantics of code fragments can be conveniently expressed in the description of input code. This information can then be utilized in the definition and application of generic code transformations. POET provides language support for specially tagged code templates, through which library developers can encode their domain-specific knowledge and can make the results of their program analysis available both to the transformation engine and to the external world for better readability and maintenance.
- Each transformation specification allows a collection of tuning parameters as the interface of re-configuration. An optimization space is therefore explicitly available to external search drivers in the empirical exploitation of best application performance. Generic search drivers can consequently be developed without being tied to any specific compiler or library optimization.

As shown in Figure 1, our transformation engine can collaborate with a source-to-source compiler (the analyzer) and a search driver to empirically find efficient implementations for a library routine. This paper focuses on using our transformation engine as a generic tool box for library developers who would like to manually build highly optimized kernel implementations. Our future work includes developing a source-to-source compiler which can perform program analysis, identify profitable transformations, and then produce a POET kernel specification file as result of parameterization for subsequent empirical tuning. Either manually produced by library developers or automatically by an optimizing compiler, the POET kernel specification can serve as the distribution form of a kernel implementation which can then be empirically tuned whenever the application needs to be ported to a different machine.

3. THE OVERALL APPROACH

The key technical contribution of this paper is the empirical tuning approach shown in Figure 1 for automatic generation of efficient implementations for performance-critical library routines. The core of this approach is the POET language and transformation engine, which offer the flexible empirical tuning of application performance by providing a modular communication interface among independent optimizing compilers, application developers, and empirical search drivers. The transformation engine offers a generic tool box to library developers for building customized code optimizations and allows such optimizations to be generalized for other applications. Additionally, it offers a portable output language for source-to-source compilers to generate parameterized code transformations and to explicitly formulate program analysis results to the external world. Programmers can modify and extend the output of optimizing compilers to additionally incorporate their domain-specific knowledge. Using our POET TE can greatly improve the efficiency of tuning since the compiler or library developer needs to perform the analysis only once when creating the kernel specification. This original analysis result may then be used without change for an arbitrary number of tuning sessions across the architectures of interest.

Correctness of transformations. The POET language provides a means for library developers and optimizing compilers to express parameterized optimizations to an arbitrary input computation. The correctness of the optimized code depends on two factors: whether the transformation routines are defined correctly in the POET TE library, and whether the transformation routines are invoked correctly in the kernel specification script. If either the TE library or the kernel specification is incorrect, the optimized code may have different semantics than those of the original input computation, and flawed algorithm implementations may be produced.

Since both the TE library and kernel specifications may be modified by library developers for the purpose of applying domain-specific optimizations, the POET transformation engine provides debugging support for tracing the results of different code transformations to an input code. Our existing work assumes that each input library kernel has an external mechanism for testing correctness of its implementation (we have used the ATLAS testing and timing mechanism in our experimental evaluation in Section 4). Each algorithm implementation generated by our transformation engine is first tested for correctness before the performance of the implementation is measured and used in the empirical tuning of code optimizations.

Searching for best performance. As shown in Figure 1, the POET transformation engine relies on a separate empirical search driver to determine the configuration of transformation parameters that lead to the best performance. The transformation engine can collaborate with arbitrary search drivers so that different search strategies can be used to find a satisfiable algorithm implementation. For the purpose of our experimental evaluation, which does not yet focus on search, we have produced straightforward shell scripts to provide the search driver for each library routine. Our search drivers iteratively experiment with different configurations

for the transformation parameters based on domain-specific information about each library routine being tuned.

Language Independence. POET aims to serve as the basis for building general transformation engines independent of what programming language that the input computation is coded in. As illustrated in Figure 3, POET uses a collection of code templates to specialize the transformation engine to an arbitrary programming language. The code templates serve as templates not only for code generation but also for parsing the input computation into an internal structured representation (a specialized abstract syntax tree). Specifically, the collection of code templates serves as the grammar description of a programming language, and the POET parser will interpret the grammar description when attempting to decompose the input computation into a structured representation.

The POET parser first reads the input computation as a list of strings. It then decompose the string list into structured code templates like a top-down recursive descent parser. The process is different from a conventional recursive descent parser in that it allows the grammar description to be provided dynamically as input data together with the input computation. As a result it can be used to parse computations coded in an arbitrary programming language as long as a valid grammar description for the language is also provided as input. Since all recursive descent parsers require the grammar description to satisfy certain constraints, POET require the code template definitions to satisfy similar properties. When such properties are violated, input annotations such as those in Figure 5 are required to ensure the success of evaluation. The input annotations also serve to improve both the efficiency and flexibility of parsing. Specifically, the input annotations allow the POET parser to quickly discover the computation structure without exploring multiple parsing alternatives. Additionally, code fragments that are not targets of optimization can be saved as a list of strings without being parsed into structured code templates.

Expressing Transformations. POET provides full programming support, including compound data structures (lists, tuples and code templates), arbitrary control flow (conditionals, loops and recursive function calls), pattern matching, as well as built-in transformation operations (e.g. replace, permute, duplication code fragments) for conveniently building arbitrary customized optimizations. Consequently, any transformation that can be expressed in a general-purpose programming language such as C/C++ can also be easily expressed in POET. Additionally, POET provide specialized support, such as automatic tracing of code fragments going through transformations, to ease the programming of parameterized transformations for empirical tuning.

Programming in POET. POET is designed to be a scripting language that can be easily understood and used by developers to build their own customized optimizations. Although we expect that kernel specifications for library routines will be eventually automatically produced by an optimizing compiler, library developers are expected to inspect the transformation script and modify the kernel specifications when necessary. As mentioned earlier, in this present work we have manually written the kernel specifications for the ATLAS routines surveyed in Section 4.

Since POET is a high-level scripting language which includes support for both general-purpose control flow and specialized operations for program transformation, writing transformations in POET is expected to be much easier than writing specialized code generators in C/C++ or any other general purpose programming language. The challenge of POET programming lies in that instead of writing a concrete algorithm implementation for a library routine, developers are required to formulate parameterized transformations to an input code. Since the majority of code transformations can be automated by optimizing compilers and the POET TE library, the difficulty level of extending a POET kernel specification is expected to be significantly less than hand-tuning variations of algorithm implementations in low-level C or assembly.

4. RESULTS

We have used our POET transformation engine to tune several linear algebra kernels from the popular ATLAS library [27]. By comparing our performance results with the best kernel performance of both ATLAS and the native compilers, we have verified that (1) Even highly aggressive compilers used in isolation rarely achieve the level of performance required by HPC applications; (2) this level of efficiency can be satisfied by our POET approach, which we show produces kernels with performance comparable to ATLAS's highly tuned implementations; (3) the POET TE can achieve better performance than hand-tuned kernels when those kernels are not updated frequently enough in the face of ongoing architectural evolution; (4) by integrating our POET TE with empirically tuned libraries such as ATLAS, we can improve the performance of existing HPC libraries by providing a complementary kernel optimization approach which is highly portable across different computer architectures.

Our work in tuning ATLAS kernels includes implementing relevant optimizations in the POET TE library and then hand-writing a kernel specification file for each ATLAS kernel being tuned. Because none of the transformation routines in the POET TE library utilizes any domain-specific information about the ATLAS kernels, we expect that the POET transformation engine can similarly achieve high-performance levels for other computational kernels and performance-critical library routines in general.

4.1 Methodology

ATLAS first tunes some simplified performance kernels, and then uses these kernels to implement fast BLAS and LAPACK routines [26]. To evaluate the overall performance impact when using POET TE to generate important ATLAS library kernels, we performed two sets of experiments. First, we used ATLAS's timing routines to measure the performance of PTE produced kernels and compared them

directly against the best ATLAS implementations. Sections 4.2 and 4.3 present relevant results for Level 3 and 2 BLAS kernels respectively. Second, we integrated the POET-produced kernels within ATLAS as user-contributed routines, and evaluated the overall performance impact when this extended ATLAS is used to implement higher-level LAPACK routines such as the QR-solve (Section 4.4).

Platform	**Cmp**	**Flags**
2.66Ghz C2D	icc 9.1	-xP -msse3 -O3 -mp1 -fomit-frame-pointer
(Core2Duo)	gcc 4.0.1	-mfpmath=sse -msse3 -O2 -m64 -fomit-frame-pointer
2.2Ghz ATH (Athlon 64 X2)	gcc 4.2.0	-mfpmath=387 -falign-loops=4 -fomit-frame-pointer -O2

(a) Compiler and flag information by platform

	Core2Duo		**Athlon-64 X2**	
Prec	**scal**	**vec**	**scal**	**vec**
single	5,320	21,280	4,400	8,800
double	5,320	10,640	4,400	4,400

(b) Theoretical peak by platform (MFLOPS) (Prec: precision of floating-point operations; scal: using scalar op; vec: using vectorized op.)

Table 2: Platform Summary

We concentrate on the ubiquitous x86 platform, and report performance for the newest machines from both AMD (2.2 Ghz Athlon-64 X2) and Intel (2.66 Ghz Core2Duo) that we have access to (abbreviated as ATH and C2D, respectively). The ATH runs Linux, and the C2D OS X. The theoretical peak of the platforms are summarized in Table 2(b). These architectures have different peak performance depending on the precision of the floating-point operations used, and whether vectorized vs. scalar operations are used.

All timings were done with ATLAS version 3.7.30, using the best available compiler version and flags, as shown in Table 2(a). We used the Intel compiler *icc* solely on the C2D platform, as *icc* was not specialized for the AMD architecture. Since profiling runs of *icc* using the actual data never produced speedup on our machine, and occasionally caused slowdown, we report the best performance achieved by *icc* without profiling. All timers used ATLAS's cycle-accurate walltimer, and since walltime is prone to outside interference, we repeated each timing six times (on an unloaded machine) and took the minimum time. All results were obtained using the ATLAS timers, which flush the cache (this means that our numbers will be lower, but more accurate for usage, than those often reported elsewhere). We report performance in MFLOPS, rounded to the nearest whole number.

4.2 Level 3 BLAS Kernels

ATLAS uses a simplified GEMM kernel to support the entire Level 3 BLAS [26] (we will refer to this simplified kernel as `gemmK` to distinguish it from the full BLAS routine GEMM). The POET input specification for this kernel is shown in Figure 5. This kernel is specialized into three cases in order to handle varying β in Figure 5; in this section we report on the performance for ATLAS's most commonly-used β variant, $\beta = 1$; typically the $\beta = 0$ case is slightly faster, and the $\beta = X$ case is slightly slower.

Since the cost of Level 3 BLAS kernels tends to dominate in the majority of algorithms, ATLAS tunes Level 3 BLAS much more aggressively than the Level 1 or 2. In particular, `gemmK`, like all of ATLAS's kernels, is tuned by the *multiple implementation* [26, 27] method, where a series of hand-tuned and generic implementations are searched, and the best performing is selected. ATLAS additionally tunes `gemmK` by a second and orthogonal tuning strategy, where a completely automated ANSI C source generator is used to find the best implementation for a given architecture and C compiler combination. Since the source generator search is ATLAS's most general strategy, we track the performance it achieves separately as **ATLAS-gen**; the full search, which includes both multiple implementation and source generator search, is labeled **ATLAS-full**.

Table 4.1 shows the performance of `gemmK` for each architecture and precision (kernel names are prefixed by '`s`' for single precision, and '`d`' for double precision). The performance results of three different methodologies are presented: The performance of using *gcc* (**gcc+ref**) and *icc* (**icc+ref**) to compile a reference implementation of *gemm* similar to the code shown in Figure 5; the performance of ATLAS kernels achieved using code generator search only (**ATLAS-gen**) and achieved using both the code generator search and multiple implementation search (**ATLAS-full**); and the performance results achieved by our POET kernel specification when empirically tuned using our transformation engine (**PTE+spec**).

The first thing to notice is that our PTE-tuned implementations handily outperforms `ATLAS-gen` for all problems except double precision on the Athlon-64. This is primarily because SIMD vectorization is required to get good performance for all other surveyed precision/architectures, but `ATLAS-gen` uses the scalar FPUs only (as shown in Table 2(b), ATH has the same scalar and vector peak for double precision, thus the code generator is competitive for this case). This is because ATLAS uses *gcc* as its default compiler, and *gcc* cannot yet successfully autovectorize these kernels. Additionally, we see that the `PTE` numbers are substantially better in all cases when compared against reference compilation using *gcc/icc* (in our worst case, we are still more than twice as fast as the fastest compiler). Therefore, we succeed in our first goal of outperforming or matching the most general part of ATLAS.

When we compare `PTE` and `ATLAS-full`, we get mixed results. For three of the four cases we see that our numbers are competitive with those of ATLAS's best hand-tuned codes, but that we lose by a modest amount. For `sgemmK/C2D`, however, we win by a reasonable margin. The reason for this is clear: for the three cases where we lose, ATLAS has kernels which have been hand-optimized by the ATLAS developers for both the architecture and kernel in question. However, ATLAS's multiple implementation shows the Achilles' heal of hand-tuning: the last case has not yet been hand-tuned specifically for the `C2D`, and thus our automated process

Kernel name	2.66Ghz Core2Duo gcc +ref	icc +ref	ATLAS gen	ATLAS full	PTE+ spec	2.2Ghz Athlon-64 X2 gcc +ref	ATLAS gen	ATLAS full	PTE+ spec
sgemmK	571	6226	4730	13972	15048	1009	4093	7651	6918
dgemmK	649	3808	4418	8216	7758	939	3737	4009	3754

Table 3: Performance in MFLOPS of Various gemmK Implementations (gcc+ref/icc+ref: reference implementation compiled with gcc/icc; ATLAS gen/full: ATLAS implementation using source-generator/full search; PTE+spec: implementation produced by POET transformation engine.)

Kernel name	2.66Ghz Core2Duo gcc +ref	icc +ref	ATLAS full	TE+ spec	2.2Ghz Athlon-64 X2 gcc +ref	ATLAS full	TE+ spec
sgerK	1230	2927	3751	3400	639	1005	962
dgerK	439	438	462	519	411	518	500
dgemvNK	382	574	939	1069	408	799	902
dgemvTK	556	574	835	1079	579	739	1049
sgemvNK	438	859	1838	2097	528	1185	1986
sgemvTK	556	1826	1752	2171	835	1389	2056

Table 4: Performance in MFLOPS of various Level 2 BLAS Kernels (gcc+ref/icc+ref: reference implementation compiled with gcc/icc; ATLAS full: ATLAS implementation using full search; PTE+spec: implementation produced by POET transformation engine.)

is able to outperform the best available hand-tuned kernel (which in this case is a kernel originally tuned for the Pentium 4). We have not yet implemented all the relevant optimization techniques in our `TE`, so we expect to further narrow the performance gap with the hand-tuned codes as the work progresses. However, these numbers are already impressive enough to convincingly demonstrate the promise of this more automatic (and thus more persistent in the face of architecture change) tuning process.

4.3 Level 2 BLAS Kernels

ATLAS uses three simplified kernels to optimize the entire Level 2 BLAS, and we will call these kernels `gemvNK`, `gemvTK`, and `gerK`. Because they are less critical for application performance and require more kernels to cover the required functionality, ATLAS tunes the Level 2 BLAS only through multiple implementation (i.e. there are no Level 2 code generators). Therefore, the category of **atlas-gen** is meaningless here, and not tracked. In looking at the Level 2 kernel performance (summarized in Table 4.1), we see that this reliance on only an empirical search of hand-tuned kernels, coupled with their relative neglect by the developers when compared to the Level 3, results in less well-optimized implementations of these kernels. Therefore, our PTE-optimized kernels exceed the performance obtained by ATLAS in the majority of the Level 2 BLAS kernels. As before, both `ATLAS` and `PTE` substantially exceed the performance obtained by simple compilation. We have several optimizations known to be beneficial for these types of kernels still to be added to our `PTE`, and so we expect our performance advantage in these kernels to widen yet further.

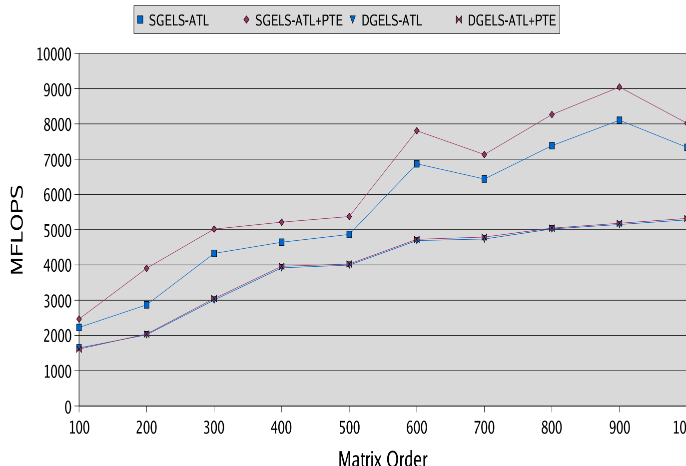
4.4 Improvements for LAPACK

So far, we have reported speedups in ATLAS's kernel routines, which are used to optimize the entire Level 2 and 3 BLAS, which are in turn the performance engine of a host of Linear Algebra applications. A question arise as to whether speeding up such kernels indeed speeds up the higher-level codes as expected. A survey of Linear Algebra applications is beyond the scope of this paper, but to give some indication, Figure 7 shows the performance of LAPACK's widely used least squared solve (`[D,S]GELS`) driver routine (solved using one right hand side), which performs the solve using the QR factorization. Here we report the performance achieved by ATLAS alone (`xGELS-ATL`) versus that achieved when we allow ATLAS's multiple implementation search to use our `PTE`-tuned kernels (`xGELS-ATL+PTE`). For the Athlon-64 (Figure 7(b)), we sped up the Level 2 BLAS, with much greater advantage achieved in single precision. Thus we see that `ATL+PTE` is noticeably faster for single precision results than pure ATLAS. `ATL+PTE` is slightly faster for double precision, but only barely.

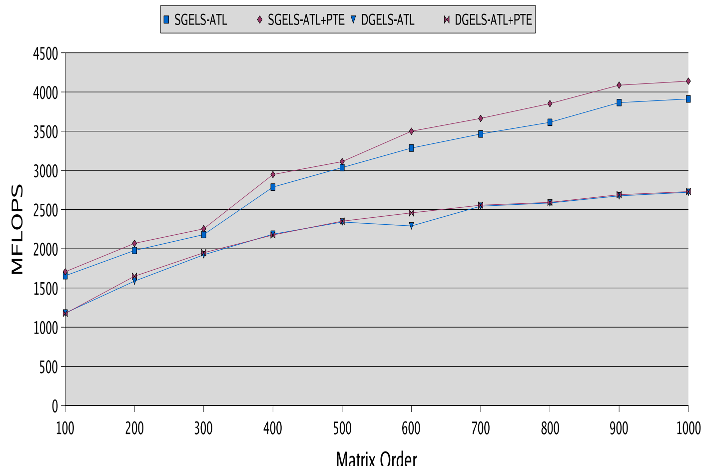
The results are largely the same on the Core2Duo (Figure 7(a)), but since we sped up both the Level 2 and 3 BLAS for single precision on this platform, the results are even more impressive. Therefore, these tunings are indeed more widely useful, and we can additionally observe a key feature of our approach: we can use it to improve existing tuning frameworks. In the short term, we plan to submit our `PTE`-tuned kernels to the ATLAS group. Longer term, it should be possible for packages such as ATLAS to directly leverage our `PTE` just as they presently do the native compilers.

5. RELATED WORK

There are more than a few highly successful empirical tuning frameworks which provide efficient kernel implementations for important scientific domains, such as those for dense and sparse linear algebra [7], signal processing [11, 17], among



(a) On OS X/2.66Ghz Core2Duo



(b) On Linux/2.2Ghz Athlon-64 X2

Figure 7: Performance vs. Problem size of LAPACK QR Factor and Solve

others [22, 24]. Additionally, some systems permit users to specify the desired kernel operation in a high-level mathematical notation [17, 2, 4]. Our approach can be applied to general purpose applications beyond those targeted by domain-specific research. Further, it complements existing domain-specific research by providing an efficient transformation engine to help existing libraries more readily port to different computer architectures.

Recent research has produced some general-purpose empirical tuning frameworks where compilers are employed to support performance tuning of arbitrary applications. These empirical tuning compilers iteratively re-configure well-known optimizations according to performance feedback of the optimized code and have demonstrated that empirical tuning of application performance can significantly improve the effectiveness of compiler optimizations [28, 19, 15, 16, 21, 12, 32, 5]. These compiler-based frameworks apply to all applications that have access to the optimizing compiler. However, they restrict applications to optimizations available only within the compiler, which typically does not provides much information to the outside world, e.g., why particular transformations were or were not applied. Additionally, each empirical compiler is by itself a significant infrastructure which typically includes a large and growing collection of routines for program analysis, code optimization, and language processing capabilities. Our own infrastructure is significantly lighter weight, and therefore should be more suitable for inline use by applications or other tuning frameworks.

POET supports existing iterative compilation frameworks [15, 16, 19, 21, 12] by providing an output language for parameterizing code optimizations for empirical tuning. In particular, POET's explicit parameterization is designed to clearly separate analysis and code generation phases from the search phase. This permits the arbitrary use of search and modeling techniques [24, 18, 31, 5].

A key purpose served by the POET language is *parameterization* of generic code transformations for empirical tuning. This distinguishes POET from the large, existing body of work on powerful languages and tools for expressing static code transformations [23, 1]. We intend to use POET in the context of an empirical search process; we do not specifically address run-time code generation as performed by more general multistage languages and systems [3, 9, 13, 14].

Similar to POET, the X language [8] also aims at supporting compact representation of multiple program versions for empirical tuning. The X language is an annotation language which uses C/C++ pragma and macro substitution to guide the application of a pre-defined collection of loop- and statement-level optimizations by a compiler. The X language parameterizes the behavior of an optimizing compiler instead of explicitly expressing applicable code optimizations. As a result, it provides limited support for defining new transformations, besides using pattern-matching rewrite rules. In contrast, POET is a extensible language that allows programmers to build arbitrary customized optimizations and allows more flexible parameterization and control of both predefined and customized optimizations.

Both the POET and X languages reply on compiler technologies that effectively parameterize code optimizations. For example, Cohen, *et al.* [6] uses the polyhedral model to parameterize the composition of loop transformations applicable to a code fragment. Developing compiler techniques to effectively parameterize complex code optimizations is a focus of our future research.

5.1 Conclusion

We have presented a new cost-effective approach to high performance optimization, where we use a transformation engine to automatically generate efficient library kernel implementations for empirical tuning. The transformation engine offers both flexible and efficient empirical tuning of appli-

cation performance by providing a modular communication interface among independent optimizing compilers, application developers, and empirical search drivers. Additionally, it can be used as a generic tool box for library developers to build customized code optimizations and allows such optimizations to be generalized for other applications. Our approach produces kernels with essentially the same level of performance as those achieved by the popular ATLAS empirical tuning framework. We have shown that our POET TE can improve overall ATLAS performance, and thus forms a valuable addition to existing empirical tuning frameworks. Further, since the POET TE can process kernel specifications from arbitrary problem domains, it can be used in the empirical optimization of kernels beyond those supported by application-specific tuning frameworks such as ATLAS.

6. REFERENCES

[1] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[2] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of *ab initio* quantum chemistry models. *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[3] O. Beckmann, A. Houghton, M. Mellor, and P. H. J. Kelly. Run-time code generation in C++ as a foundation for domain-specific optimisation. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume LNCS 3016, pages 291–306, Internationnal Seminar, Dagstuhl Castle, Germany, March 23–28, 2003, Revised Papers, 2004.

[4] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. Quintana-Orti, and R. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[5] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO*, San Jose, CA, USA, March 2005.

[6] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS*, pages 151–160, Boston, MA, USA, June 2005.

[7] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[8] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *LCPC*, Hawthorne, NY, USA, October 2005.

[9] D. R. Engler, W. Hsieh, and M. Kaashoek. 'C: A language for high-level, efficient, and machine-independent code generation. In *POPL*, pages 131–144, 1996.

[10] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.

[11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[12] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A pratical method for quickly evaluating program optimizations. In *HiPEAC*, November 2005.

[13] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248:147–199, 2000.

[14] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

[15] T. Kisuki, P. M. Knijnenburg, and M. F. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT*, Philadelphia, PA, October 2000.

[16] G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *SC*, Baltimore, MD, USA, November 2002.

[17] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[18] A. Qasem and K. Kennedy. A cache-conscious profitability model for empirical tuning of loop fusion. In *LCPC*, October 2005.

[19] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. In *Proc. Los Alamos Computer Science Institute (LACSI) Symposium*, 2004.

[20] See page for details. FFTW homepage. `http://www.fftw.org/`.

[21] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, San Jose, CA, USA, March 2005.

[22] T. L. Veldhuizen and D. Gannon. Active Libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Philadelphia, PA, USA, 1998. SIAM.

[23] E. Visser. A survey of strategies in rule-based program transformation systems. *J. Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[24] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for automatic performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[25] R. C. Whaley and A. Petitet. Atlas homepage. `http://math-atlas.sourceforge.net/`.

[26] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[27] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[28] R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *The 2005*

International Conference on Parallel Processing, June 2005.

[29] Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *The Journal Of Supercomputing*, 27:219–264, 2004.

[30] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.

[31] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[32] Y. Zhao, Q. Yi, K. Kennedy, D. Quinlan, and R. Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical Report UCRL-TR-217808, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, December 2005.