Automated Transformation for Performance-Critical Kernels

Qing Yi R. Clint Whaley
Dept. of Computer Science, University of Texas at San Antonio

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. Our framework requires only 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 description language. We have applied our transformation engine to generate highly tuned code for key linear algebra kernels used in the ATLAS tuning framework. The time required to produce specifications for these kernels is orders of magnitude less than that required to hand-craft kernel implementations, and yet our framework has achieved similar performance to ATLAS's highly tuned kernels.

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 [17, 15] and FFTW [4, 11], 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 [3, 5, 9, 13, 14, 1, 2].

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 are not well supported and thus do not achieve adequate performance. These systems are therefore not of great assistance in optimizing applications beyond their domain that nonetheless require a high level of 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. Currently, the analyzer role must be performed manually by a programmer. Our research plan calls for replacing this manual analysis step with a source-to-source compiler, which can automatically discover potential optimizations through compiler analysis (possibly with the help of programmer markup). 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 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. Our transformation engine (TE) is portable, easy to extend, and simple to use. Additionally, the TE is language neutral and can be easily specialized to produce optimized

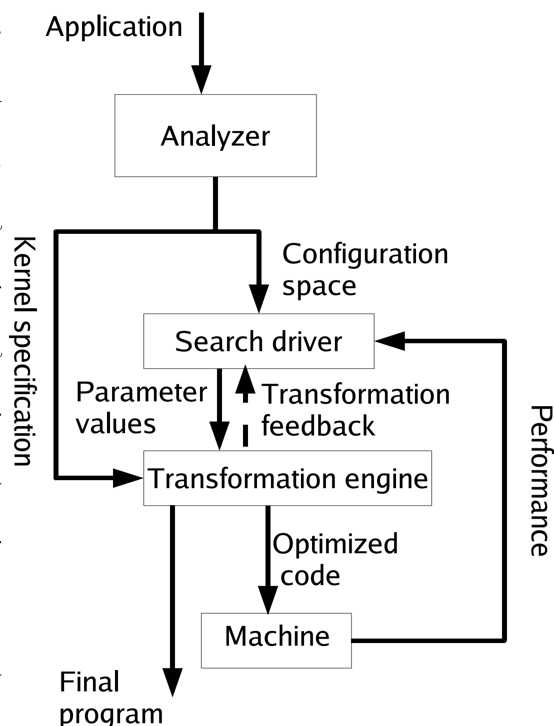


Figure 1: Our empirical tuning approach

kernels in an arbitrary source language, including C, FORTRAN, or assembly.

A key feature of our approach is that it formulates a kernel implementation in terms of the sequence of parameterized transformations which may be applied by the transformation engine to optimize performance. Compared to the conventional domain-specific empirical tuning frameworks, our approach has the following advantages:

- First, our approach is targeted at producing general-purpose kernel implementations. The transformation engine includes an extensive library of code transformations that have proven to be able to significantly improve application performance. The programmer need only specify where to apply these transformations in order to extract high performance for an arbitrary computational routine. Although domain-specific knowledge is required to correctly apply the transformations, the effort required to generate a specification describing the transformations to tune is orders of magnitude less than that required to write and hand-optimize a performance-critical kernel.

- Second, our transformation engine supports natural parameterization and re-configuration of all the relevant code optimizations, so that a single version of kernel specification can be copied to to different architectures and empirically tuned to find the best implementation. In contrast, although most empirical tuning frameworks parameterize their kernel implementations to ensure portability, the parameterization is often not as comprehensive, and adapting to different architectures and/or extending to other kernels takes significant more development and maintenance.

- Finally, using our transformation engine, the kernel specification file only needs to provide a straightforward implementation using the most simple and intuitive algorithm. The code is therefore easier to understand and maintain. As new architectures are brought forward, adapting the kernel implementation at worst involves adding a few new optimizations, while the rest of the kernel specification is left unchanged. This is significantly more efficient than having to rewrite the kernel implementations to accommodate new architectural features.

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 linear algebra kernels in the ATLAS library [17, 15]. The kernel implementations produced by our framework achieved similar performance to that seen in ATLAS's highly tuned kernels. Our results indicate that using the 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 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 [3], signal processing [5, 9], among others [13, 14]. Additionally, some systems permit users to specify the desired kernel operation in a high-level mathematical notation [9, 1, 2]. 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 [18, 10, 7, 8, 12, 6, 20]. 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.

In contrast to a full blown iterative compilation framework, our transformation engine is simply comprised of an interpreter for a small embedded language (POET), and a library of compact code transformation routines written in this language. The transformation engine if very light-weight (currently including about

3000 lines of C++ code implementing the POET interpreter and about 600 lines of POET code implementing various code transformations). Our transformation engine therefore can be easily included as part of the library or application distribution and serve as the automated code generator for the empirical tuning of kernel implementations.

3 The POET Transformation Engine

Our transformation engine (TE) is based on a small special-purpose language named POET (Parameterized Optimizations for Empirical Tuning) [19]. 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 interpreter, a transformation library, and a collection of front-end definitions which specialize the transformation library for different programming languages such as C, FORTRAN, or Assembly. In the center of the TE is the POET language interpreter, which takes as input a kernel specification from the programmer and a collection of parameter values from a separate search driver, invokes a specialized language frontend 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 a separate search

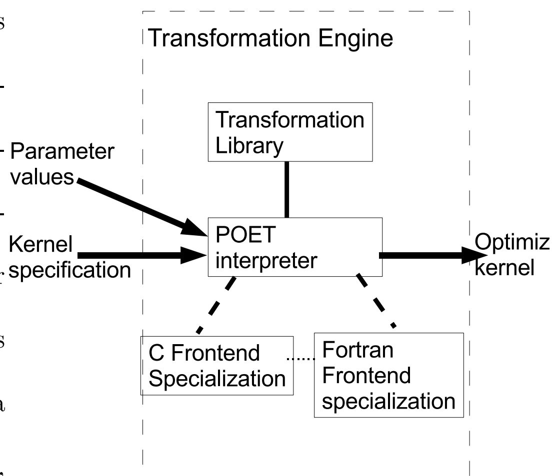


Figure 2: POET transformation engine

```
<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 3: Skeleton of Loop blocking as defined in the transformation library

driver until a satisfactory implementation is found.

In order to build an optimized kernel implementation, the programmer 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 library and the language specialization can be used without detailed knowledge about their implementation. Additionally, programmers can easily expand the transformation library and define their own customized transformations. In the following, Sections 3.1 and 3.2 first briefly introduce our existing transformation library. Section 3.3 then focuses on how to use the POET TE to build optimized kernels for empirical tuning.

3.1 The Transformation Library

Our transformation 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 3 shows a few skeletons of POET routines relevant to applying a loop blocking transformation.

As shown in Figure 3, POET uses keyword $xform$ to define routines that can be invoked to transform input code fragments. Each $xform$ routine uses the *pars* attribute to define the 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 as replacement of the original one. Additional information may be returned when the *output* attribute is defined.

The entire transformation library comprises *xform* routines as shown in Figure 3. These 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 transformation 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 AST (abstract syntax tree) representation. The extensive support for building customized transformations in POET allows programmers to easily extend the transformation library with their own $xform$ routines.

Most of the code transformations in our *xform* 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 for programmers to build extremely optimized kernel implementations. We argue that it is much easier and more cost-efficient for programmers to invoke the appropriate code transformations than to hand-tune an optimized assembly implementation. Through the POET language, our transformation library provides a flexible interface both for programmers to extend the library with additional customized optimizations and for users to invoke the predefined library routines with minimal compiler background. The built-in parameterization support by the transformation engine also allows natural empirical tuning of the optimized kernels which would be much more portable than hand-written assembly.

```
<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 4: C frontend specialization

3.2 Frontend Specialization

Our transformation engine is language neutral in that both POET and the transformation library are independent of what language that the input kernel 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 4 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 transformation 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 both by the transformation library and by the kernel specification as an abstract representation of the input computation. As shown in Figure 4, each code template can have two attributes, *pars* and *attr*, which define the parameters and additional properties of the source code. The concrete source code of each code template is then defined in a general programming language such as C and is parameterized by variables declared in *pars* (in Figure 4, the reserved token, ‘@’, is used for context switching between POET parameters and source strings of the underlying language). As an example, Figure 4 includes several code templates for C source which are recognized by the loop blocking transformation shown in Figure 3. These templates are used to parse the matrix multiplication

```
<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*

kernel given in Figure 5

Code template specializations like those shown in Figure 4 are used only for parsing the input source and for emitting the transformed output. The POET transformation library uses these templates as abstract representations of the input code without knowing how these representations are implemented. When an input program is defined in terms of code templates, generic routines predefined in our transformation library can recognize the structure of the input program and apply optimizations accordingly. The definition of code templates therefore serves to specialize the transformation library to kernels in 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.

3.3 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 transformations to the input code. For example, Figure 5 shows the POET input specification for *dgemm*, the matrix multiplication kernel from the ATLAS library [17, 15], and Figure 6 shows the transformation specification for the kernel.

Input Specification. In order to optimize a computational kernel, the POET TE needs to parse the input code and translate it into an abstract code template representation which can be understood by the transformation 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 (see Section 3.2). Each POET annotation either starts with "//@" and lasts until the end of the current line, 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 4.

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

```
<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*

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 interpreter can parse the input computation correctly without being language specific (note that eventually 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.

Transformation specifications. After the input specification is processed by a POET 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 transformation 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.

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*,$nest3$, $nest2$,and $nest1$, into *gemm*, the global variable which stores the internal representation of input code. As various code 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 $nest3$; and *nest1_Unroll* applies loop unrolling to $nest1$. Pre-defined 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 transformation library. Both routines from the library have their tuning parameters reconfigured when the routine name is invoked.

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

3.4 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 reconfiguration. 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.

Instead of utilizing any existing optimizing compiler, 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.

The POET transformation engine offers more flexible empirical tuning of application performance because it provides a modular communication interface among independent optimizing compilers, application developers, and empirical search drivers. It offers a generic tool box to library developers for building a customized collection of code optimizations and allows such optimizations to be generalized for other applications. It offers a portable output language for analyzers and source-to-source compilers to generate parameterized code transformations and to explicitly formulate program analysis results to the external world. Moreover, 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 scripts. This original analysis result may then be used without change for an arbitrary number of tuning sessions across the architectures of interest.

4 Results

We have used our POET transformation engine to tune several linear algebra kernels from the popular ATLAS library [17]. 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.

4.1 Methodology, Architecture and Version Details

ATLAS first tunes some simplified performance kernels, and then uses these kernels to implement fast BLAS and LAPACK routines [16]. 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 multiple implementation 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 (Core2Duo)	icc 9.1	-xP -msse3 -O3 -mp1 -fomit-frame-pointer
	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 1: 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 1(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 1(a). We only used the Intel compiler *icc* on the C2D platform as *icc* was not specialized for the AMD architecture. We do not report numbers for *icc* when using profiling because our profiling runs of *icc* using the actual data never produced speedup, and occasionally caused slowdown. Since we were unable to determine if this was due to the fact that *icc* is not yet well-tuned for OS X/Core2Duo, or if we were simply unable to discover the proper flags for profiling to shine, we omit our disappointing profiling results. 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 [16] (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* [16, 17] 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 4.2 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**).

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

Table 2: 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.)

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 1(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 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 4.3), 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.

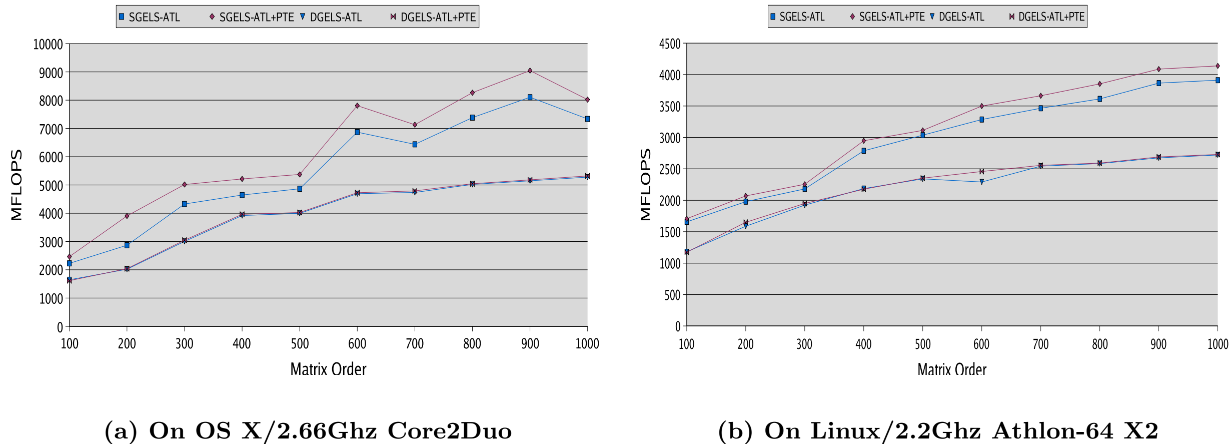
	2.66Ghz Core2Duo				**2.2Ghz Athlon-64 X2**		
Kernel name	**gcc +ref**	**icc +ref**	**ATLAS full**	**TE+ spec**	**gcc +ref**	**ATLAS full**	**TE+ spec**
sgerK	1230	2927	3751	3400	639	1005	962
dgerK	439	438	462	519	411	518	500
dgemvNK	382	574	939	1069	408	799	902
dgemvTK	556	574	835	1079	579	739	1049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

Table 3: 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.)

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



(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

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.

4.5 Conclusion

We have presented a new cost-effective approach to high performance optimization, where we use a transformation engine to automatically generate efficient kernel implementations for empirical tuning. While the time required to create a kernel specification is orders of magnitude less than that required by conventional hand-tuning, 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

[16] 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`.

[17] 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`).

[18] R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *The 2005 International Conference on Parallel Processing*, June 2005.

[19] 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.

[20] 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.