POET: Parameterized Optimizations for Empirical Tuning

Qing Yi[*] Keith Seymour[†] Haihang You[†] Richard Vuduc[‡] Dan Quinlan[‡]
[*] University of Texas at San Antonio, TX, 78249, USA
[†] University of Tennessee at Knoxville, TN, 37996, USA
[‡] Lawrence Livermore National Laboratory, CA, 94551, USA

Abstract

The excessive complexity of both machine architectures and applications have made it difficult for compilers to statically model and predict application behavior. This observation motivates the recent interest in performance tuning using empirical techniques. We present a new embedded scripting language, POET (Parameterized Optimization for Empirical Tuning), for parameterizing complex code transformations so that they can be empirically tuned. The POET language aims to significantly improve the generality, flexibility, and efficiency of existing empirical tuning systems. We have used the language to parameterize and to empirically tune three loop optimizations—interchange, blocking, and unrolling—for two linear algebra kernels. We show experimentally that the time required to tune these optimizations using POET, which does not require any program analysis, is significantly shorter than that when using a full compiler-based source-code optimizer which performs sophisticated program analysis and optimizations.

I. Introduction

Over the past 30 years, both modern computer architectures and software applications have become extremely complex. A modern computer typically includes one or more core microprocessors, multiple levels of cache, a virtual memory system, and an interconnection network; while a non-trivial application often includes millions of lines of code distributed in thousands of different files. Such complexity exceeds the capability of typical compilers to statically model and predict application behavior. As a result, an application's performance has frequently suffered relative to its potential.

Recent research has demonstrated that empirical tuning of application performance can significantly improve the effectiveness of compiler optimizations [23], [16], [12], [13], [18], [9], [27]. In these approaches, sensitive restructuring transformations are parameterized and dynamically re-configured according to performance feedback of the optimized code. Previous research has used both library- and compiler-based approaches, where a library generator or an optimizing compiler is dynamically reconfigured to make different optimization decisions. In the library-based approach, the library code generator is manually written by program developers based on their domain-specific knowledge about both the applications and the machines to which their libraries might be ported. A number of such libraries have been shown to be highly successful in extracting portable high performance

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

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

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

<code Loop pars=(i,start,stop,step) >
@init=(if start=="" then "" else (i "=" start ));
 test=(if stop=="" then "" else (i "<=" stop ));
 incr=(if step=="" then "" else (i "+=" step ));
@for (@init@; @test@; @incr@)
</code>


<xform Unroll pars=(input,uloop) tune=(ur=16)>
  if (!(input : Nest#(loop,body))) then (
    if (input : Function#(head,decl,body)) then
       Function#(head,decl,Unroll#(body,uloop))
    else if (input : Sequence#(s1,s2)) then
       Sequence#(Unroll#(s1,uloop),Unroll#(s2,uloop))
    else input
  )
  else if (loop != uloop) then
    Nest#(loop, Unroll#(body,uloop))
  else if (!ur || ur == 1) then input
  else (
    loop : Loop#(ivar,start,stop,step);
    dup = (body DUPLICATE#(_, ur-1,(ivar "+=1;" ENDL body)));
    Sequence#(Nest#(Loop#(ivar,start,(stop "-" ur-1),step),dup),
              Nest#(Loop#(ivar,"",stop,step),body))
  )
</xform>
```

Fig. 1. POET definition of loop unrolling

1-4244-0910-1/07/$20.00 ©2007 IEEE.

```
include opt.poet

<define loopJ Loop#("j",0,"n",1)>
<define loopI Loop#("i",0,"m",1)>
<define loopK Loop#("k",0,"l",1)>
<define mmStmt "c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];">
<define nest1 Nest#(loopK,mmStmt)>
<define nest2 Nest#(loopI,nest1)>
<define nest3 Nest#(loopJ,nest2)>
<define mmHead "void dgemm(int m, int n, int l, double alpha,
                double *a, double *b, double *c)">

<define dgemm Function#(mmHead, "int i, j, k;", nest3)>
<define mm_unroll (Unroll#(dgemm,loopK)) >
<define mm_block  (Block#(dgemm,nest3,mmStmt))>
<define mm_permute (Permute#(dgemm,nest3,mmStmt))>
<define mm_block_unroll (Unroll#(mm_block,
                                 InnermostLoop#(mm_block)))>

<output mm.c dgemm>
<output mm_unroll.c (Unroll.ur=8; mm_unroll)>
<output mm_block.c (Block.bsize=(8 64 128); mm_block)>
<output mm_permute.c (Permute.order=(3 2 1); mm_permute)>
<output mm_block_unroll.c
        (Block.bsize=(16 16 8);Unroll.ur=8; mm_block_unroll)>
```

Fig. 2. POET definition of matrix multiplication

across different machines [22], [5], [8], [14], [19]. To extend the success of empirically tuned libraries, recently, many iterative optimizing compilers have been built to support empirical performance tuning of general applications.

Although both the library-based and compiler-based empirical tuning approaches are effective, there are some limits to their generality and portability. The library-based approach requires manual orchestration of sophisticated low-level optimizations and is therefore time consuming and error-prone. Additionally, the optimization techniques cannot easily transfer to other applications. The compiler-based approach applies to all applications that have access to the optimizing compiler. However, it restricts the applications to optimizations available only within the compiler. The empirical optimizing compiler cannot incorporate customized code transformations, and it typically does not provides much information to the outside world, e.g., why particular transformations were or were not applied.

This paper presents a language, POET (Parameterized Optimization for Empirical Tuning), to decouple the empirical tuning aspect of performance optimization from the specifics of any library or compiler. POET is an embedded scripting language which supports highly efficient and flexible parameterization of code optimizations for scientific computing. It can be embedded in code written in any other language, such as C, C++, or FORTRAN, by treating input code fragments as parameterized strings without attempting to interpret the underlying language. Figures I and 2 illustrate the POET language definitions for optimizing a matrix multiplication kernel. The output of the optimization is shown in Figure 5. The details of the language is explained in Section II.

The goal of the POET language is to compactly describe parameterized code optimizations and how these optimizations can be applied differently to improve the performance of input applications. A POET script can be created for each application, either by an optimizing compiler or by a professional library developer. The script can then be ported to different machine architectures and dynamically configured by an independent empirical search engine, which invokes a POET language interpreter to build different instances of optimized code. We have carefully designed the POET language to offer strong support for the following capabilities.

- Generic restructuring transformations can be easily defined and can be used to optimize arbitrary application codes. The definition of loop unrolling in Figure I illustrates an example of such generic code transformation. Library developers can easily use POET to define their customized code transformations without having to build any specialized compiler. We will define and provide as part of the language distribution a code transformation library which includes a large collection of predefined generic code transformations. Both library developers and optimizing compilers can use these predefined transformations to optimize their code. Figure 2 illustrates how to apply predefined code optimizations to a matrix-multiplication kernel.
- Important properties and special semantics of code fragments can be easily expressed in the description of input code. The information can then be utilized in the definition of generic code transformations. In Figure I, *Function*, *Nest*, *Sequence* and *Loop* are predefined code templates which convey special meaning to the loop unrolling transformation. By describing the input code in Figure 2 in terms of these code templates, we can then easily apply different loop optimizations to the matrix-multiplication kernel. Through the language support for specially tagged code templates, library developers can encode their domain-specific knowledge within the input code description, and optimizing compilers can easily make the results of their program analysis available to the external world.
- Each restructuring transformation allows a collection of integer tuning parameters (*e.g.*, the ur parameter for loop unrolling in Figure I) as the interface of reconfiguration. An optimization space is therefore explicitly available to all independent search engines in the empirical exploitation of best application performance. Generic search engines can consequently be developed without being tied to any specific compiler or library optimization. The design of a better interface to independent search engines is our ongoing work.

Note that the POET language parameterizes code transformations, which are typically the final stage of any program optimization, instead of parameterizing the configuration of any optimizing compiler. In fact, POET is designed to be the output language of optimizing compilers. An optimizing compiler will first perform program analysis to discover all optimizations that might potentially improve application performance. It then decomposes the possible result of optimization into a collection of POET code transformation routines that can be applied to improve the input code. The

POET output will then serve as the distribution form of the application and can be empirically tuned whenever the application needs to be ported to a different machine.

POET offers more flexible empirical tuning of application performance because it serves as a modular communication interface among independent optimizing compilers, application developers, and empirical search engines. It offers a generic tool to library developers in building their customized collection of code optimizations and in allowing such optimizations to be generalized for other applications. It offers a portable output language for optimizing compilers to generate parameterized code transformations and to explicitly formulate program analysis results to the outside world. The optimizing compiler no longer needs to reside on the target machine for the application to be empirically tuned. Moreover, programmers can modify and extend the output of optimizing compilers to additionally incorporate their domain-specific knowledge.

Besides the convenient flexibility provided by the POET language, using POET can greatly improve the efficiency of tuning since the compiler or professional library developer needs to perform the analysis only once when creating the scripts. The analysis result is then tuned as many times as necessary without reapplying the analysis. To verify this belief, we have implemented the POET interpreter and used POET to parameterize three preliminary optimizations—loop interchange, blocking and unrolling—for two linear algebra kernels. We measured the time to empirically tune these kernels both through POET parameterization and through a full-blown source-to-source optimizer, and show that tuning using POET takes significantly less time.

II. The POET Language

The concrete grammar of the POET language is shown in Figure 3 and illustrated in Figure I and 2. In summary, a POET script contains a collection of disjoint sections, each uniquely identified by a global name and implemented by mapping the unique name (the handle of the section) to a POET expression.

A. Defining POET Sections

POET supports four different kinds of sections: $code$, $xform$, $define$ and $output$. The $code$ and $xform$ sections are used to define generic code transformations which can be applied to arbitrary code; the $define$ and $output$ sections are used to describe transformations applicable to any particular input code. Both $code$ and $xform$ sections use the $pars$ attribute to define a sequence of input parameters. The $xform$ section additionally uses the $tune$ attribute to define a sequence of integer parameters which can be used to reconfigure the transformation. Each tuning parameter must have a default value which defines the default behavior of the transformation.

```
(1) poet : {section}
(2) section :
       "<" "code" ID {codeAttr} ">" Exp "<" "/code" ">"
(3)   | "<" "xform" ID {xformAttr} ">" Exp "<" "/xform" ">"
(4)   | "<" "define" ID Exp ">"
(5)   | "<" "output" FNAME Exp ">"
(6) codeAttr : "pars" "=" "(" ID {"," ID} ")"
(7) xformAttr : "pars" "=" "(" ID {"," ID} ")"
(8)   | "tune" "=" "("ID"="INT{","INT} {";"ID"="INT{","INT}}")"
(9) Exp : Op
      | Op ";" Exp | Op "," Exp | Op Exp
(10)Op : "if" Op "then" Op "else" Op
(11) | ID {"." ID} "=" Op
(12) | ID ":" Op
(13) | ID {"." ID} "#" Unit  | ["car","cdr",INT] "#" Op
(14) | REPLACE "#" "(" Op "," Op "," Op ")"
(15) | PERMUTE "#" "(" Op "," Op ")"
(16) | DUPLICATE "#" "(" Op "," Op "," Op ")"
(17) | Op ["+","-","*"] Op  | "-" Op
(18) | Op ["<","<=",">",">=","==","!="] Op
(19) | "!" Op | Op "&&" Op  | Op "||" Op
(20) | Unit
(21)Unit : ID {"." ID} | INT |  SOURCE | "(" Exp ")"
```

ID	identifiers, e.g. Loop, Nest;
FNAME	file names, e.g., mm.c, ../dgemm.c;
INT	integer literals, e.g., 1, 0, 16, 64;
SOURCE	strings of code fragments, e.g. “c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];”
$\{s_1 s_2 ... s_n\}$	Zero or more occurrences $s_1 s_2 ... s_n$ e.g., $\{“,”ID\}$ (zero or more of “, ID”);
$[t_1, t_2, .., t_m]$	one token out of $t_1, t_2, .., t_m$ e.g., $[“+”, “-”, “*”]$ (“+”, “-”, or “*”).

Fig. 3. Formal grammar of the POET language

(1) e := i	$e.val = i.val$
(2) $\mid$ s	$e.val = s.val$
(3) $\mid e_1\ e_2$	$e.val = list(eval(e_1), eval(e_2))$
(4) $\mid e_1,e_2,...,e_m$	$e.val = tuple(eval(e_1), eval(e_2), ..., eval(e_m))$
(5) $\mid$ car # e_1	$t = eval(e_1); e.val = is_list(t)?first(t) : t$
(6) $\mid$ cdr # e_1	$t = eval(e_1); e.val = is_list(t)?rest(t)$: “”
(7) $\mid$ i # e_1	$t = eval(e_1); e.val = tuple_elem(t, i.val)$
(8) $\mid$ cv # e_1	$if(!replaceCode)\ e.val = cv\#eval(e_1)$ else $\{set_pars(cv, eval(e_1)); e.val = eval(find_code(cv))\}$
(9) $\mid e_1 : e_2$	$e.val = match_AST(e_1, e_2)$
(10) $\mid$ dv	$e.val = find_code(dv)$
(11) $\mid$ lv	$e.val = find_value(lv)$
(12) $\mid lv = e_1$	$set_value(lv, eval(e_1)); e.val =$ “”
(13) $\mid$ xv # e_1	$set_pars(xv, eval(e_1)); e.val = eval(find_code(xv))$
(14) $\mid$ repl(e_1,e_2,e_3)	$e.val = Replace(eval(e_1), eval(e_2), eval(e_3))$
(15) $\mid$ perm(e_1, e_2)	$e.val = Permute(eval(e_1), eval(e_2))$
(16) $\mid$ dup(lv, t, e_1)	$for(i = 0; i < t; ++i)$ $\{set_value(lv, i); e.val = list(eval(e_1), e.val)\}$
(17) $\mid e_1;e_2$	$eval(e_1)$: $e.val = eval(e_2)$
(18) $\mid$ if(e_1,e_2,e_3)	$e.val = bool(eval(e_1))?\ eval(e_2) : eval(e_3)$
(19) $\mid e_1\ op_i\ e_2$	$e.val = eval(e_1) op_i e_2.val$
(20) $\mid$! e_1	e.$val =\ !\ bool(e1.val)$
(21) $\mid e_1\ op_b\ e_2$	$e.val = bool(e1.val)\ op_b\ bool(e2.val)$

Notations: i: integer constants; s: string constants; lv: local variables of the current scope; cv: global handles of $code$ sections; dv: global handles of $define$ sections; xv: global handles of $xform$ sections; op_i: integer binary operations $(+,-,*,<,>,==,<=,>=,!=)$; op_b: boolean binary operations (&&, $||$).

Fig. 4. Evaluation rules for POET expression

a) **The** $code$ **Section:** POET uses $code$ sections to define parameterized code templates that convey special semantics. When an input program is defined in terms of these code templates, a generic code transformation (defined by $xform$ sections) can recognize the structure of the input program and apply optimizations accordingly. For example, Figure I includes four code sections: $Loop$, $Nest$, $Sequence$ and $Function$. These templates have predefined meanings which are recognized by the loop unrolling transformation

defined in the $Unroll$ section. When the matrix multiplication kernel is defined using these templates in Figure 2, it can then benefit from the predefined generic optimizations in $opt.poet$. Note that a reserved token, '@', appears in the code templates in Figure I but is skipped in the grammar specification. The '@' symbol is implicitly handled by the POET lexical analyzer and is used solely for context switching between POET definitions and source strings of the underlying language.

b) **The** $xform$ **Section:** Each $xform$ section defines a generic code transformation that can be applied to optimize arbitrary code. As illustrated by the $Unroll$ section in Figure I, each $xform$ section operates on an input code, reorganizes various components within the input, and returns the restructured code. The transformation relies on a collection of predefined code templates to recognize the structure of input code. All the inputs to a $xform$ section are defined using the $pars$ attribute. In Figure I, the $Unroll$ transformation has two parameters, the input code to transform ($input$) and the loop within $input$ to unroll ($uloop$).

Each $xform$ section can additionally have a sequence of tuning parameters which controls the re-configuration of code transformation. In Figure I, the transformation $Unroll$ has a single tuning parameter, ur, which controls the loop unrolling factor. Transformation tuning parameters can be modified anywhere within a POET script, as illustrated by the output sections of Figure 2.

c) **The** $define$ **and** $output$ **Sections:** In POET, the $define$ and $output$ sections are used to decompose input programs into pre-defined code templates and to dynamically apply different transformations to optimize the input code. Each $define$ section is a macro definition which associates a global name with a particular POET expression. Each $output$ section defines a file name and the POET expression that should be output to the file as result of code optimization. In the expected common case where all the necessary optimizations have been predefined, users of POET only need to write a collection of $define$ and $output$ sections to optimize their code, as illustrated by Figure 2.

B. POET expressions

The non-terminal Exp in Figure 3 defines the concrete syntax of a POET expression. Each POET expression is internally translated into an abstract syntax tree (AST) representation, which is dynamically evaluated when the result of expression is needed. The abstract syntax of POET expressions, as well as rules to evaluate each operation, are defined in Figure 4. The following summarizes the composition of POET expressions in more detail.

d) **Atomic Values:** POET supports two kinds of atomic values, integers and strings, denoted by symbols i and s, respectively, in Figure 4. A string value is defined either by enclosing the content within a pair of double quotes

```
<xform BlockHelp pars=(nest, inner, blocks)>
  if (nest == inner) then ("",nest,"")
  else (
    nest : Nest#(Loop#(i,start,stop,step),body);
    ii = (i i);
    if (blocks : ((bsize rsize))) then ""
    else (bsize=blocks; rsize=blocks);
    rest = BlockHelp#(body, inner, rsize);
    ivars = 3#rest;
    ( (Loop#(ii,start,stop,bsize) 1#rest),
       Nest#(Loop#(i,ii,("min("ii"+"bsize","stop")"),step),
             2#rest),
       (if (ivars=="") then ii else (ii "," ivars)))
  )
</xform>

<xform Block pars=(input, nest1, inner) tune=(bsize=16)>
  if (! (input : Nest#(loop,body)))  then (
    if (input : Function#(head,decl,body)) then (
        "#define min(a,b) ((a < b)? a : b)" ENDL
        Function#(head,decl,Block#(body,nest1,inner))
    )
    else if (input : Sequence#(s1,s2)) then
       Sequence#(Block#(s1,nest1,inner),
                 Block#(s2,nest1,inner))
    else input
  )
  else if (input != nest1) then
      Nest#(loop, Block#(body,nest1,inner))
  else (
      rest = BlockHelp# (input, inner, bsize);
      Sequence#( ("int " 3#rest ";" ENDL),
                 BuildNest#(1#rest, 2#rest))
  )
</xform>

<xform Permute pars=(input, nest1, inner) tune=(order=0)>
  if (order == 0) then input
  else if (! (input : Nest#(loop,body)))  then (
    if (input : Function#(head,decl,body)) then
        Function#(head,decl,Permute#(body,nest1,inner))
    else if (input : Sequence#(s1,s2)) then
       Sequence#(Permute#(s1,nest1,inner),
                 Permute#(s2,nest1,inner))
    else input
  )
  else if (input != nest1) then
      Nest#(loop, Permute#(body,nest1,inner))
  else (
      loops = FindLoopInNest#(nest1, inner);
      nloops = PERMUTE#(order, loops);
      BuildNest#(nloops, inner)
  )
</xform>
```

Fig. 6. POET definition of loop blocking and permutation

(*e.g.*, "i < b", "3.215"), or by simply placing the content inside template definitions of $code$ sections (*e.g.*, the content of $Function$ in Figure I). POET provides a special string, ENDL, to denote line-breaks in the underlying language.

e) **Compound Data Structures:** POET supports two built-in compound data structures: list and tuple. Operations supporting these data structures are defined in lines (3–7) of Figure 4.

Lists are composed by simply concatenating elements together. For example, a "$<=$" b produces a list ℓ with at least three elements, a, "<=", and elements from b (if b is a list, ℓ contains all elements in b; otherwise, ℓ contains b). Because the type of b is unknown,lists are dynamic data structures that may contain arbitrary numbers of elements. In contrast, tuples are composed by connecting a predetermined number of elements with commas. For example, "i'', 0, "m'', 1 produces a tuple t with four elements,

```
void dgemm(int m, int n, int l, double alpha,
           double *a, double *b, double *c)
{
  int i, j, k;
  int kk;
  for (i=0; i < m; i += 1) {
     for (j=0; j < n; j += 1) {
        for (k=0; k < l-7; k += 1) {
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];  k+=1;
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];  k+=1;
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];  k+=1;
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];  k+=1;
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];  k+=1;
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];  k+=1;
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];  k+=1;
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];
        }for ( ; k < l; k+=1) {
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];
        }
     }
  }
}
```

(a) Output in mm_unroll.c

```
#define min(a,b) ((a<b)? a : b)
void dgemm(int m, int n, int l, double alpha,
           double *a, double *b, double *c)
{
  int i, j, k;
  int ii, jj, kk;
  for (ii = 0; ii < m; ii+=8)  {
     for (jj = 0; jj < n; jj+=64)  {
        for (kk = 0; kk < l; kk+=128)  {
           for (i = ii; i < min(ii+8,m); i+=1)  {
              for (j = jj; j < min(jj+64,n); j+=1)  {
                 for (k = kk; k < min(kk+128,l); k+=1)  {
                    c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];
                 }
              }
           }
        }
     }
  }
}
```

(b) Output in mm_block.c

```
void dgemm(int m, int n, int l, double alpha,
           double *a, double *b, double *c)
{
  int i, j, k;
  for (k=0; k < l; k += 1) {
     for (j=0; j < n; j += 1) {
        for (i=0; i < m; i += 1) {
           c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];
        }
     }
  }
}
```

(c) Output in mm_permute.c

Fig. 5. Optimized matrix multiplication output from interpreting Figure 2

"i'',0,"m'',and 1. In practice, lists are used in almost all cases due to their flexibility, and tuples are used to define values with a known structure.

Elements in a list ℓ are accessed through two operations: $car\#\ell$, which returns the first element of ℓ (if ℓ is not a list, it simply returns ℓ); and $cdr\#\ell$, which returns the rest of the list (if ℓ is not a list, it returns empty string). Similarly, each element in a tuple t is accessed by invoking $i\#t$, where i is the index of the element being accessed. For example, if ℓ = (a "<=d" 3), then $car\#\ell$ returns a, $car\#cdr\#\ell$ returns "<=", and $car\#cdr\#cdr\#\ell$ returns 3; if $t = (i, 0, \text{"}m\text{"}, 1)$, then $1\#t$ returns "i", $2\#t$ returns 0, $3\#t$ returns "m", and $4\#t$ returns 1.

f) **Code Templates:** Each $code$ section in POET defines a parameterized code template that conveys some special semantics. The code template is treated as a unique user-defined compound data structure, where the template parameters are treated as data fields, until the final optimized code needs to be output to an external file. As shown at line (8) of Figure 4, unless the $replaceCode$ is set to $true$, which indicates the final output is being constructed, the invocation of code template cv is simply treated as building a compound data structure using the given template argument(e_1).

Treating code templates as user-defined compound data structures not only allows compact representation of the input program but also allows generic code transformations being easily built and conveniently applied successfully, as illustrated by the mm_block_unroll optimization in Figure 2, where loop blocking and unrolling are applied as separate passes over the input code.

g) **Determining Types Of Expressions:** POET is a dynamically typed language; that is, the types of all values are dynamically checked immediately before they are used. POET provides a pattern matching operation, syntactically defined at line (12) of Figure 3, to examine types of expressions. The evaluation of pattern matching is defined at line (9) of Figure 4, where the structures of e_1 and e_2 are recursively examined to determine whether their types match.

Note that when uninitialized local variables appear in the pattern matching operation, these variables are treated as place holders which can be matched to arbitrary expressions. If the type matching is successful, all the uninitialized variables would have been assigned a valid value. Therefore the pattern matching operation can be used both for dynamic type checking and for initializing local variables. Both purposes are illustrated extensively in the $Unroll$ section of Figure I.

h) **Global and Local Variables:** POET expressions may contain both global and local variables. Global variables are introduced by $define$ sections and are denoted as dv in Figure 4. Local variables are denoted as lv in Figure 4 and can be introduced through the $pars$ and $tune$ attributes or simply through the use of new names in $code$ or $xform$ sections. Examples of local variables in Figure I include $init$, $test$, $incr$ in section $Loop$, and $loop$,$body$,$head$,$decl$, dup in section $Unroll$. Two POET operations can be used to initialize local variables, direct assignment (line (11) of Figure 3 and pattern matching (line (12) of Figure 3. Both operations are used extensively in Figure I.

i) **Applying Code Transformation:** Invocation of code transformations (*e.g.*, invocation of the $Unroll$ section in Figure I) is syntactically defined at line (13) of Figure 3. As shown by line (13) of Figure 4, the evaluation first sets the transformation parameters and then simply evaluates the POET expression associated with the transformation handle xv. Each invocation of xv is associated with a dynamic activation record implemented within entries of the local symbol table, so that values of local variables are not disrupted by recursive invocation of the transformation routines. Before invoking each *xform* handle xv, the tuning parameters of xv can be modified to allow dynamic reconfiguration of transformation.

Besides code transformations defined in *xform* sections, POET supports three built-in code transformations: $replace$, which replaces all appearances of a data item with a different value in a target expression; $duplicate$, which replicates the target code by a pre-configured number of times; and $permute$, which rearranges the order of elements in an predefined list. All transformations return a new expression as result instead of modifying any of the input parameters. The syntax of these buildin transformations are defined at

lines (14–16) of Figure 3, and their evaluations are defined at lines (14–16) of Figure 4.

j) **Control Flow and Integer/Boolean Operations:** POET supports three control-flow structures: sequence, conditional and function call (*e.g.*, invocations to code transformations). Loops are not directly supported, and users are expected to use recursive function calls instead.

Line (15–18) of Figure 4 defines the collection of integer and boolean operations supported by POET. These operations include integer arithmetics (+,-,*), integer comparisons (<,<=,>,>=,==,!=), and boolean arithmetics (!, && and ||). The semantic definition of these operations is straightforward and follows the C language. When evaluating boolean operations, all input values are converted to integers (1 and 0) by invoking the *bool* function in Figure 4, which converts empty strings to 0 and all other non-integer values to 1.

III. Using POET

The POET language is designed to support easy definition of generic code transformations and efficient configuration of parameterized transformations for empirical tuning. This design objective is supported by treating user-defined code templates as extensible compound data structures, by dynamic pattern matching in code transformations, and by dynamically re-configurable tuning parameters. The support of generic code templates will particularly help both optimizing compilers and professional library developers to communicate their special knowledge about the input application. The POET language can express all code transformations which can be encoded using a recursive algorithm.

We have used POET to support the tuning of three code optimizations: loop interchange, blocking, and unrolling. Figure 2 shows POET definitions for applying these transformations to optimize a matrix multiplication kernel. The definition of loop unrolling is shown in Figure I. The definition of loop blocking and permutation is shown in Figure 6.

The output code from applying unrolling, blocking and permutation transformations are shown in Figure 5. We purposefully defined all the transformations to produce almost identical output as produced by our source-to-source loop optimizer [24]. The transformation can certainly be adapted so that the output code is more efficient. How to best apply important transformations in POET is an important research topic that we will address in future work.

IV. Experimental Results

To show that using POET can significantly reduce the code generation time of empirical tuning, we experimentally compared the tuning time using POET to the time using a full-blown source-to-source optimizer, for two dense linear algebra kernels.

A. Experimental Setup

We empirically searched the optimization space of two kernels, dense double-precision matrix-matrix multiply (`DGEMM`) and matrix-vector multiply (`DGEMV`), on a 3.4GHz Pentium 4, using gcc 3.4.4 as the back-end compiler. We used baseline C implementations of these kernels that are simplified from the complete BLAS implementations. We exploited the search space of applying loop interchange, blocking and unrolling optimizations to both kernels.

We used the source-to-source loop optimizer developed by Yi, *et al.* [24]. The optimizer is called LoopProcessor and is implemented within the ROSE compiler infrastructure [17]. We used command-line options to configure what transformations LoopProcessor should apply to our kernel implementations.

For empirical tuning, we used a search engine developed by You, *et al.* [26]. The search engine is based on the Nelder-Mead simplex search, which is a non-derivative based direct search method. For comparison, we have implemented a second search technique which is completely random and stops after a predetermined number of evaluations (trials).

B. Results and Discussion

Table I shows the overall search time and the best performance of optimized kernels using the POET approach and the LoopProcessor respectively. Each table entry is the averaged result of ten runs of applying the simplex search for the best blocking and unrolling factors. Since the simplex search starts from random points and continues until a desired performance level is observed, the number of trial evaluations typically varies from run to run. On average using POET and LoopProcessor perform a similar number of evaluations. The search time in Table I includes the time to generate, compile (using gcc), and execute instances of the optimized kernels.

¿From Table I, the overall search time using POET is more than two times faster than using LoopProcessor, whereas the optimized kernels produced by POET also perform better compared to kernels produced by the LoopProcessor. The lagging performance of optimizations by LoopProcessor is due to several calls to a function min which is not inlined by the back-end gcc compiler, while the POET script defines min as a macro instead of a function call. The LoopProcessor lacks the ability to generate macros due to implementation issues.

To ensure a fair comparison between the empirical tuning time spent in the POET interpreter and in the LoopProcessor, Figures 7 and 8 separate out the time spent in generating all the blocked code (with varying block sizes but no inner-loop unrolling) and all the unrolled code (with varying unrolling factors but no blocking) respectively. ¿From these two figures, using POET is nearly an order of magnitude faster, largely because it avoids the expensive analysis required by LoopProcessor. When using large unrolling factors with

Routine	Code Generator	Number of Evaluations	Search Time (sec)	Search time(sec) per evaluation	Generated Code Run Time (sec)
DGEMM N=1000	POET	193.0	356.43	1.8468	7.95
	LoopProcessor	164.6	869.77	5.2841	9.69
DGEMV N=2000	POET	151.2	287.57	1.9019	0.02
	LoopProcessor	146.0	734.72	5.0323	0.04

TABLE I. Comparison of search times

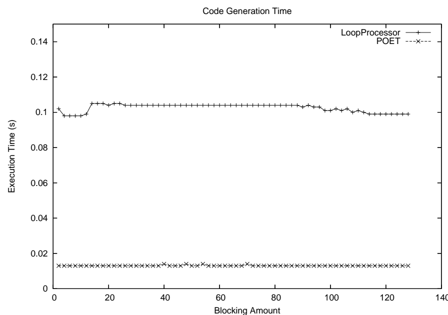

Fig. 7. Code generation time for different block sizes

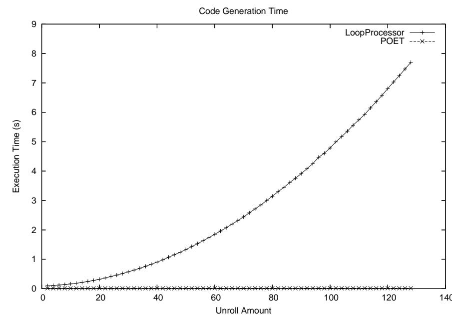

Fig. 8. Code generation time for different unroll factors

LoopProcessor, the code generation consumes most of the overall evaluation time. The POET interpreter does not suffer from this performance degradation, which results in much shorter code generation time.

To further inspect the quality of our optimized kernels, Figures 9 and 10 present the performance results of comparing our best optimized kernels with ATLAS, a highly tuned linear algebra library [22]. Here we increased the search space to four dimensions for `DGEMV` (loop interchange, unrolling, and two levels of blocking) and to five dimensions for `DGEMM` (loop interchange, unrolling, and three levels of blocking). Simplex search performs consistently better than random search for `DGEMM`, while random search performs better consistently for `DGEMV`.

Figures 9 and 10 also show that the best POET-optimized versions perform performs slightly worse than ATLAS for `DGEMV`, but more than four times slower for `DGEMM`. The performance gap is due to the currently non-existing statement- and instruction-level optimizations in the POET-produced versions. Our future work will extend the POET scripts for these kernels to include more low-level optimizations.

V. Related Work

A key purpose served by the POET language is *parameterization* of generic code transformations for empirical tuning, where the application of transformations to any given input code is dynamically experimented and modified until a desired optimization is found. This distinguishes POET from the large, existing body of work on powerful languages and tools for expressing static code transformations [20], [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 [2], [7], [10], [11].

Our work is influenced by research on domain-specific automatic tuning systems, such as those for dense and sparse linear algebra [5], and signal processing [8], [14], among others [19], [21]. These systems feature special parameterized code generators which take as input a desired kernel and specific parameter values, and output a kernel implementation, typically in FORTRAN or C. The POET approach targets general applications, but could aid in the development and maintenance of such domain-specific generators.

POET supports existing iterative compilation frameworks [12], [13], [16], [18], [9]. 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 [21], [15], [25], [3].

Similar to POET, the X language [6] 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 the final code transformation result. While the original program source is more readily available using X annotations, the only mechanism for defining new transformations, besides those pre-defined, is based on 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 would benefit greatly from compiler technologies that effectively parameterize code optimizations. For example, Cohen, *et al.* [4] 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.

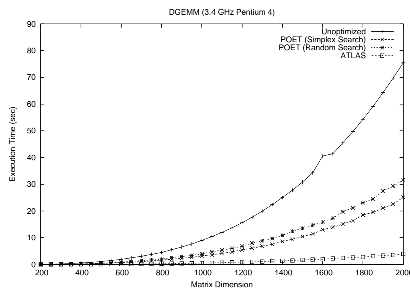

Fig. 9. DGEMM performance

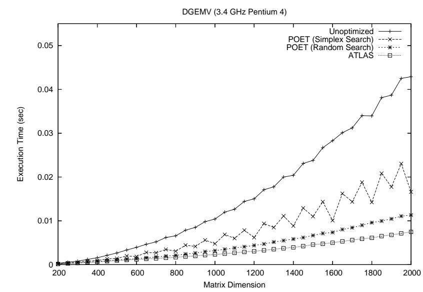

Fig. 10. DGEMV performance

VI. Conclusion

This paper presents an embedded scripting language, POET, for both flexible and efficient empirical tuning of application performance. The POET language can be embedded within arbitrary programming languages such as C, C++, or FORTRAN, and support efficient parameterization of general code transformations produced either by compilers or by professional programmers. POET is an essential part of the automated tuning process, serving to simplify the generation of complex code transformations. Our experimental results have verified that using POET parameterization can significantly reduce the empirical tuning time of otherwise using a sophisticated source-code optimizer.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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.

[17] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proc. Joint Modular Languages Conference*, 2003.

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

[19] 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.

[20] 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.

[21] 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.

[22] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[23] R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *ICPP*, pages 89–98, Oslo, Norway, 2005.

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

[25] 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.

[26] H. You, K. Seymour, and J. Dongarra. An effective empirical search method for automatic software tuning. Technical Report ICL-UT-05-02, Dept. of Computer Science, University of Tennessee, May 2005.

[27] 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.