Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2011) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Extensive Parameterization And Tuning of Architecture-Sensitive Optimizations [1]

Qing Yi[a], Jichi Guo[a]

[a] *Computer Science, University of Texas At San Antonio*

Abstract

The complexity of modern architectures require compilers to apply an increasingly large collection of architecture-sensitive optimizations, e.g., parallelization and cache optimizations, which interact with each other in unpredictable ways. We present a framework to support fine-grained parameterization of these optimizations and flexible tuning of their configuration space. Instead of directly generating optimized code, we extend an optimizing compiler to output its optimization decisions in POET, a scripting language designed for extensive parameterization of source-to-source program transformations. We then use a transformation-aware (TA) search algorithm to support flexible tuning of the parameterized transformation scripts to achieve portable high performance. We have used our framework to apply 6 highly interactive optimizations, parallelization via OpenMP, cache blocking, array copying, unroll-and-jam, scalar replacement, and loop unrolling, and present results of exploring their combined configuration space.

1. Introduction

Emerging multi-core architectures require a large collection of optimizations, including both thread-level parallelization and memory locality optimizations, for scientific applications to achieve high performance. Iterative compilation [1, 2, 3, 4] can use runtime feedbacks of evaluating differently optimized code to automatically select promising optimization configurations on modern architectures. However, as a single optimized code is generated as output, the optimizations cannot be later reconfigured for a different architecture, and developers have limited control over how a compiler may parameterize or tune the configurations of different optimizations.

We present a framework, shown in Figure 1, to support more extensive parameterization and tuning of architecture-sensitive optimizations. Instead of directly generating optimized code, we extend an optimizing compiler to output its optimization decisions into extensively parameterized program transformations in POET, a scripting language designed for applying source-to-source program transformations. The POET output can then be ported together with an

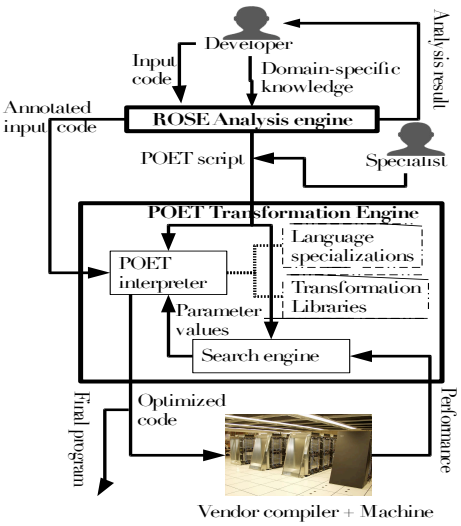


Figure 1: The optimization workflow

[1] This research is funded by NSF awards CCF0747357 and CCF-0833203, and DOE award DE-SC0001770.

```
 void dgemm_test(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)
{
  int i, j, l;
  for (j = 0; j <= -1 + N; j += 1)  /*@ BEGIN(nest1=Nest) @*/
     for (i = 0; i <= -1 + M; i += 1) /*@ BEGIN(nest3=Nest) @*/
     {
       C[j * ldc + i] = beta * C[j * ldc + i];
       for (l = 0; l <= -1 + K; l += 1) /*@ BEGIN(nest2=Nest) @*/
         C[j * ldc + i] = C[j * ldc + i] + alpha * A[l * lda) + i] * B[(j * ldb) + l];
     }
 }
```

Figure 2: Matrix Multiplication code with POET annotations

annotated input program to different machines, where an empirical transformation engine can dynamically interpret the POET scripts with different optimization configurations until satisfactory performance is achieved.

The merit of our overall approach lies in its unique integration of programmable control by developers, automated optimization by compilers, and empirical tuning of the optimization space by search engines. It permits different levels of automation and programmer intervention, from fully-automated tuning to semi-automated optimization to fully programmable control. The auto-generated POET transformations are extensively parameterized so that each optimization can be turned on or off independently for each relevant array or code region, and arbitrary integers can be given as the blocking or unrolling factor for each loop being transformed. The granularity of external control is far beyond those supported by existing iterative compilation frameworks [1, 2, 3, 4, 5]. Independent search engines can be substituted with ease, and developers can easily interfere by modifying the auto-generated POET scripts.

We have used our framework to apply 6 highly interactive optimizations, parallelization via OpenMP, cache blocking, array copying, unroll-and-jam, scalar replacement, and loop unrolling. The configuration parameters of these optimizations form a considerably large, complex, and randomly interacting multi-dimensional space. To study this optimization space, we have developed a transformation-aware (TA) search algorithm based on a number of commonly-adopted compiler heuristics. Our goal is to discover important patterns that compiler optimizations interact with each other and to gain insights in terms of how to significantly reduce empirical search time without sacrificing application performance. Our main contribution includes the following.

- We propose an empirical tuning framework to support extensive parameterization and flexible tuning of architecture-sensitive compiler optimizations.
- We study the configuration space of 6 highly interactive optimizations and present results of using commonly adopted compiler heuristics to explore this space. Our study provides answers to questions such as whether a large and complex configuration space of interacting optimizations can be effectively explored by tuning one optimization at a time, how should various optimizations be ordered within such a search strategy, what heuristics can be used to efficiently explore the configuration space of each optimization. and how sensitive are the various heuristics when facing unpredictable interactions between different optimizations.

The rest of this paper is organized as follows. Sections 2 and 3 present an overview of our tuning framework and our transformation-aware search algorithm. Section 4 presents experimental design and results of applying our framework to tune several linear algebra routines. Section 5 and 6 present related work and conclusions.

2. Tuning Infrastructure

Our tuning framework is shown in Figure 1 and includes two main components: the ROSE analysis engine, which we built by extending the ROSE loop optimizer [6] to automatically produce POET scripts as output without affecting how it works otherwise [7], and the POET transformation engine, which includes the POET language interpreter and a configuration search engine. The ROSE analysis engine analyzes the source code of an input program, discovers optimization opportunities, and then produces two optimization output: a slightly modified source code where POET annotations are inserted to tag the code regions to be transformed, and a parameterized POET transformation script which can be invoked to apply a long sequence of program transformations with different optimization configurations. Figure 3 shows the skeleton of a POET script auto-generated by our ROSE optimizer after analyzing the code in Figure 2, which includes POET annotations automatically inserted by the optimizer to tag various code regions.

```
1: include opt.pi
2: <trace target/>
3: <input to=target syntax="Cfront.code" from=("dgemm.C")/>
......

4:<parameter pthread_nest1 type=1.._ default=1
    message="number of threads to parallelize loop nest1"/>
5:<parameter psize_nest1 type=1.._ default=256
    message="block size to run by each thread for nest1"/>
6:<parameter bsize_nest1 type=(INT INT INT) default=(8 8 8)
    message="Blocking factor for loop nest nest1"/>
7:<parameter copy1_config_C type=0..2 default=1
    message="configuration for copy array C at loop nest1;
         0: no opt; 1: array copy; 2: strength reduction"/>
8:<parameter copy2_config_A type=0..2 default=1
   message="configuration for copy array A at loop nest1"/>
9:<parameter copy3_config_B type=0..2 default=1
   message="configuration for copy array B at loop nest1"/>
10:<parameter ujsize_nest1 type=(INT INT) default=(2 2)
      message="Unroll and Jam factor for loop nest nest1"/>
11:<parameter scalar1_config_C type=0..2 default=1
          message="configuration for scalarRepl array C;
       0: no opt; 1: scalar repl; 2: strength reduction"/>
12:<parameter scalar2_config_A type=0..2 default=1
          message="configuration for scalarRepl array A"/>
13:<parameter scalar3_config_B type=0..2 default=1
          message="configuration for scalarRepl array B"/>
14:<parameter usize_nest2 type=1.._ default=4
          message="Unroll factor for loop nest2"/>

15:<eval par_nest1 = DELAY{......}/> <*OMP parallelization*>
16:<eval block_nest1 = DELAY{......}/>     <*loop blocking*>
17:<eval copyC_nest1 = DELAY{......}/>      <*copy array C*>
18:<eval copyA_nest1 = DELAY{......}/>      <*copy array A*>
19:<eval copyB_nest1 = DELAY{......}/>      <*copy array B*>
20:<eval unrolljam_nest1 = DELAY{......}/>  <*unroll & jam*>
21:<eval scalarC_nest1 = DELAY{......}/>    <*scalarRepl C*>
22:<eval scalarA_nest1 = DELAY{......}/>    <*scalarRepl A*>
23:<eval scalarB_nest1 = DELAY{......}/>    <*scalarRepl B*>
24:<eval unroll_nest2=DELAY{UnrollLoops[factor=usize_nest2]
                                (nest2[Nest.body],nest2)}/>
25:<eval cleanup_nest1=DELAY{CleanupBlockedNests
                           [trace=top_nest1](clnup_nest1)}/>

26:<eval APPLY{par_nest1};
27:      APPLY{block_nest1};
28:      APPLY{copyC_nest1};
29:      APPLY{copyA_nest1};
30:      APPLY{copyB_nest1};
31:      APPLY{unrolljam_nest1};
32:      APPLY{scalarC_nest1};
33:      APPLY{scalarA_nest1};
34:      APPLY{scalarB_nest1};
35:      APPLY{unroll_nest2};
36:      APPLY{cleanup_nest1}/>
37:<output from=(target) syntax=("Cfront.code")/>
```

Figure 3: Auto-generated POET scripts for Figure 2

2.1. The Analysis Engine

Within our framework, the ROSE optimizing compiler has essentially delegated the actual program transformations to POET, and it only modifies the input code to tag code regions which may be transformed later. Details of how to adapt the ROSE optimizer to ensure both correctness and safety of the auto-generated POET scripts is presented in [7] and beyond the scope of this paper. The auto-generated POET script can be modified by a developer if necessary to change the ordering of transformations or to integrate additional domain-specific optimizations. The final script together with the *tagged* input source code can then be ported to a variety of different machines and empirically tuned.

2.2. The POET Language

POET is an interpreted program transformation language designed for parameterizing general-purpose compiler optimizations for auto-tuning [8]. To optimize an input program, a POET script needs to specify exactly which input files to parse using which language syntax descriptions, what transformations to apply to the input code after parsing, and how to invoke each transformation. Each POET script can be extensively parameterized, where values for the parameters can be flexibly reconfigured via command-line options when invoking the POET interpreter. For example, line 3 of Figure 3 parses the matrix multiplication code in Figure 2 using C syntax descriptions specified in file *Cfront.code* and then stores the resulting AST to a global variable named *target*. The *output* command at line 37 serves to unparse the optimized AST to standard output. The inclusion of file *opt.pi* at line 1 ensures that the POET *opt* library, which supports a large collection of compiler transformations, can be invoked by the given script.

POET provides strong programming support for flexibly combining a long sequence of heavily parameterized program transformations. In Figure 3, the 11 optimizations that will be later applied to transform the input code are defined at lines 25-35 using the *DELAY* operator. The configurations of all transformations are extensively controlled by the command-line parameters declared at lines 4-14. Lines 26-36 then apply the 11 pre-defined transformations one after another using the *APPLY* operator, providing developers a clear view of all the potential optimizations that the compiler has discovered. Developers can modify optimization decisions by the compiler if necessary, e.g., by adjusting the ordering of applying different transformations at lines 26-36 or by adding additional optimizations.

2.3. The Search Engine

Our search engine in Figure 1 works with the POET language interpreter to automatically explore the optimization configuration space defined in an auto-generated POET script. It orchestrates the whole tuning process by iteratively determining what parameter values to use to properly configure each optimization, invoking the POET interpreter

Input: *tuneParams*: tuning parameters declared in POET script;
Output: *config_res*: a set of configurations found for *tuneParams*;
Algorithm:
Step1: *cur_config* = new_configuration(*tuneParams*); /* initialization */
For (each parameter $p \in$ *tuneParams*):
set *cur_config*(p) = default_value(p);
Group *tuneParams* by the loop nests they optimize;
opts = {parallelization, blocking, inner_unroll, unroll&jam, array_copy, scalar_repl};
cur_opt = first_entry(opts); *config_res* = {*cur_config*};
Step2: Set *cur_tune* = ∅; /* Set up the current tuning space. */
For (each *config* ∈ *config_res* and each loop nest L being optimized by *config*):
cur_tune∪ ={gen_tune_space(tuneParams(L), cur_opt, config) };
cur_config = *gen_first_config*(*cur_tune*);
Step3: /*Apply and evaluate each optimization configuration*/
Invoke POET transformation engine with *cur_config*;
Verify the correctness of optimized code;
cur_score = Evaluate the optimized code on hardware machine;
Step4: /*Modify *config_res* if necessary */
If (*cur_score* is better or close to those in *config_res*):
config_res = *config_res* ∪ {(*cur_config*, *cur_score*)};
If (*cur_score* is better than those in *config_res*):
Eliminate weak configurations from *config_res*;
Step5: /* try the next configuration of cur_tune */
cur_config = gen_next_config(*cur_config*, *cur_score*, *cur_tune*);
If (*cur_config* ≠ *null*): go to Step3.
Step6: *cur_opt* = next_entry(*cur_opt*, *opts*); /* try to tune the next optimization*/
If (*cur_opt* ≠ *null*): go to Sep 2;
Step7: return *config_res*; /* return result */

Figure 4: The transformation-aware search algorithm

with the parameter values, compiling the optimized code using a vendor compiler (e.g., gcc), running the compiled code, and evaluating the empirical feedbacks to guide future search. Our search engine currently uses the search algorithm described in Section 3. However, since the optimization space is explicitly made available for external control, alternative search algorithms can be easily used to substitute.

2.4. The Overall Infrastructure

Our tuning infrastructure currently supports the following six optimizations.

- Loop parallelization via OpenMP, where blocks of iterations of an outermost loop are allocated to different threads to evaluate. The optimization is parameterized by the number of threads to run in parallel and the size of each iteration block to allocate to different threads.
- Loop blocking for cache locality, where iterations of a loop nest are partitioned into smaller blocks so that data accessed within each block can be reused in the cache. The optimization is parameterized by the blocking factor for each dimension of the loop nest.
- Array copying and strength reduction, where selected arrays accessed within a blocked loop nest are copied into a separate buffer to avoid cache conflict misses, and strength reduction is applied to reduce the cost of array address calculation. For each array, the optimization is parameterized with a three-way switch to turn on both array copying and strength reduction (switch=1), strength reduction only (switch=2), or neither (switch=0).
- Loop unroll-and-jam, where given a loop nest, selected outer loops are unrolled by a small number of iterations, and the unrolled iterations are jammed inside the innermost loop to promote register reuse. It is parameterized by the number of loop iterations unrolled (the unroll factor) for each outer loop.
- Scalar replacement combined with strength reduction, where array references are replaced with scalar variables when possible to promote register reuse. The configuration of scalar replacement is similar to array copying.
- Loop unrolling, where an innermost loop is unrolled by a number of iterations to create a larger loop body. The optimization is parameterized by the loop unrolling factor (i.e., the number of iterations unrolled).

Each POET script applies the above optimizations in the order that they are discussed above. The set of standardized parameter declarations, illustrated at lines 4-14 of Figure 3, are automatically extracted by the search engine to be used as input to the transformation-aware search algorithm to determine proper configurations for the parameters.

Compared to existing iterative compilation frameworks, our infrastructure offers better modularity, flexibility and portability, as compiler optimizations are completely opened up for programmable control by developers and tuning by independent search engines. The optimizing compiler does not need to reside on the same machine that the user application is optimized for, and an explicit parameter space can be tuned using arbitrary independent search engines.

3. The Transformation-Aware Search Algorithm

Figure 4 shows our transformation aware search algorithm (implemented using Perl), which takes as input a collection of POET tuning parameters and returns a set of desirable configurations for these parameters as the result of empirical tuning. In contrast to alternative generic search algorithms which look for the maxima/minima in a multi-dimensional generic space, our search algorithm is optimization-specific in that it has full knowledge of how each

POET parameter is used to control the optimizations and tunes their configurations in a deliberate fashion, through the following steps.

Step 1. The algorithm initializes each optimization parameter with a default value given by the POET script and groups all the parameters by the loop nests that they optimize. Each optimization is then tuned independently one after another in a predetermined order. Note that the order of tuning individual optimizations when exploring their configuration space is determined by the TA search algorithm and is different from the order of applying these optimizations in the POET scripts (discussed in Section 2.4). In Figure 4, the default tuning order is defined when initializing the variable $opts$ at step (1) and is based on the following strategies.

- Loop parallelization determines the overall data size operated by each thread and thus needs to be tuned before all the other sequential optimizations.
- Architecture-sensitive optimizations such as loop blocking, unrolling, and unroll&jam should be tuned before more predictable optimizations such as scalar replacement (almost always beneficial) and array copying (rarely beneficial due to its high overhead).
- Optimizations with more significant impact should be tuned early. For example, loop blocking is tuned immediately after tuning parallelization as it critically determines whether data can be reused in the cache and thus impacts how other sequential optimizations should be configured.

Section 4 evaluates the effectiveness of the above strategies together with other varying heuristics.

Steps 2-7. These steps repetitively tune each optimization following the predetermined tuning order, where variable *cur_opt* keeps track of the current optimization being tuned, and $config\_res$ keeps track of the group of best optimization configurations found so far. In particular, Step 2 generates a new tuning space (*cur_tune*) by expanding each item in $config\_res$ with a set of new configurations to tune for *cur_opt*. Step 3 invokes the POET transformation engine with each new configuration in *cur_tune* and then collects empirical feedbacks from running the optimized code. Step 4 modifies $config\_res$, the set of desirable optimization configurations, based on performance feedbacks of running each configuration in *cur_tune*. Step 5 ensures all necessary configurations in *cur_tune* are experimented. Step 6 ensures that all optimizations have been tuned. Finally, Step 7 returns $config\_res$ as the result. Note that both steps 5-6 can skip optimizations that are known to have a negative impact based on previous experiments.

Summary. Our TA search algorithm essentially tunes the configurations of a number of optimizations one after another, where optimizations that have bigger performance impact are evaluated first before trying the less significant ones. By tuning each optimization independently of others, our search algorithm allows us to easily experiment with different heuristics to tune each optimization. For example, to reduce tuning time, the default search algorithm uses the same blocking factor for all the dimensions of a loop nest when tuning cache blocking, and uses a user-specified increment (by default, the increment is 16) to select different blocking factors to try. Further, at step 4 of the algorithm, we limit the number of top configurations in $config\_res$ to be less than a user-specified small constant (by default, at most 10 configurations are kept in $config\_res$). Since only a small constant number of best configurations can be selected after tuning each optimization, the overall tuning time is proportional to the sum of tuning each optimization independently. The algorithm is therefore fairly efficient and requires only a small number of iterations to terminate. Section 4.2 studies the effectiveness and performance tradeoffs of these heuristics.

4. Experimental Results

Our empirical tuning framework has essentially exposed all the optimization decisions by a compiler for external control by having the compiler producing a collection of parameterized program transformations in the POET language. While arbitrary search engines can be used to explore the optimization configuration space, generic search algorithms such as simulated annealing [9] are likely to get stuck at local minima/maxima due to unpredictable interactions between the large number of different architecture-sensitve optimizations. In contrast, conventional compiler heuristics for selecting configurations of these optimizations have been fairly effective for a large number of applications and may thrive similarly when used in empirical tuning. This section evaluates the effectiveness and performance tradeoffs of various optimization-specific heuristics when used in our transformation-aware search algorithm, described in Section 3. In particular, we aim to provide insights to the following open questions.

Optimization parameters	Search space for each dimension gemm	gemv	ger	default value	Additional constraints if applied to the same loop
omp-thread	(1,16)	(1,16)	(1,16)	1	
omp-block	(16,256)	(16,256)	(16,256)	72	
cache-block	$(1, 128)^3$	$(1, 128)^2$	$(1, 128)^2$	72	omp-block % cache-block = 0
unroll-jam	$(1, 16)^2$	(1,16)	(1,16)	1	cache-block % unroll-jam = 0
loop-unrolling	(1,32)	(1,32)	(1,32)	1	cache-block % loop-unrolling = 0
array-copy	$\{0, 1, 2\}^3$	$\{0, 1, 2\}^2$	$\{0, 1, 2\}^2$	2	array-copy > 0 if blocking is applied
scalar-repl	$\{0, 1, 2\}^3$	$\{0, 1, 2\}^2$	$\{0, 1, 2\}^2$	1	

(l,u): every integer i s.t. $l \leq i \leq u$; **{0,1,2}**: the three integers: 0, 1, and 2.

Table 1: The optimization search space of *gemm*, *gemv* and *ger*

- How effective are the various strategies used in our search algorithm and what are their performance tradeoffs. Is tuning one optimization at a time effectively, and how should various optimizations be ordered.
- How sensitive are the various heuristics when facing complex interactions between optimizations. What are the common groups of interacting optimizations. What alternative heuristics can be used.

Note that while an exhaustive search of the entire configuration space could extract the best performance possible using our optimizations, it is out of the question due to the astronomic amount of time required. As an alternative, we significantly increase the number of configurations tried within the sub-dimensions of an optimization to evaluate the performance impact of heuristics specific to the optimization. To model interactions among multiple optimizations, we compare the default heuristics adopted by the TA search with alternative more expensive strategies. While a promising optimization configuration could still be pruned prematurely by the more expensive strategies, the comparison illustrates the cost-effectiveness of our heuristics in terms of achieving relatively high performance without requiring overly prolonged tuning time.

4.1. Experimental Design

We evaluate our framework using three matrix computation kernels: gemm (matrix-matrix multiply), gemv (matrix-vector multiply), and ger (vector-vector multiply). We have selected these benchmarks for two reasons.

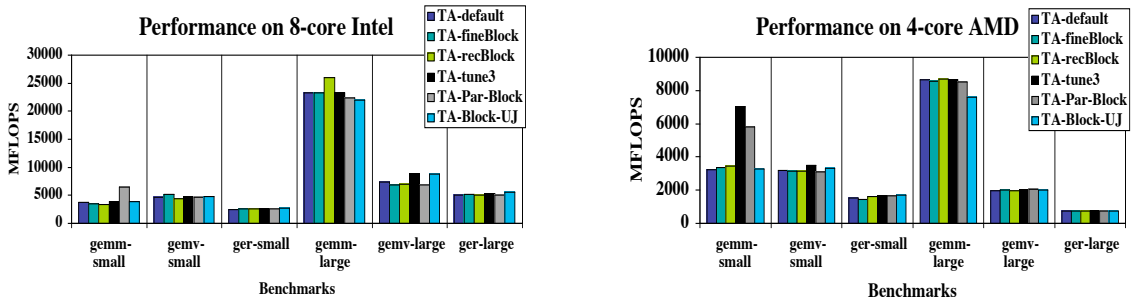
- All of them are computationally intensive, and their efficiency can be improved significantly via the collection of source-to-source optimizations (see Section 2.4) supported by our framework.
- They vary significantly in computation/data-access ratio. In particular, *gemm* is compute-bound as it reuses every data item a large number of times during evaluation; *gemv* is memory bound as only a small fraction of data are reused; *ger* is severely memory-bound as no data item is reused in the computation. Consequently, these benchmarks are expected to be representative of different behaviors demonstrated by scientific codes.

We automatically generated the POET script to optimize each code using our ROSE analysis engine [7]. Each benchmark is tuned using both small (100^2) and large (1000^2) matrices as input data. Table 1 shows the optimization search Space for each of the benchmark kernels. The search space and default values for the tuning parameters are determined by combining optimization-specific knowledge with architectural parameters (e.g., L1/L2/L3 cache capacity and the number of processing cores) of the underlying machine.

We tuned each benchmark on two multi-core machines: a quad-core machine running Linux with two dual-core 3 GHz AMD Opteron 2222 Processors (each with 128KB L1 and 1MB L2 cache per core), and an eight-core machine running MacOS with two quad-core 3 GHz Intel processors (each with 32KB L1 cache per core and a unified 4MB L2 cache). All benchmarks are compiled with -O2 option using gcc 4.2.4 on the AMD machine and gcc 4.0.1 on the Intel machine. We used the -O2 instead of -O3 option to prevent gcc from applying overly aggressive loop optimizations to our already heavily optimized code. Each optimized code is first tested for correctness and then linked with its timing driver, which sets up the execution environment, repetitively invokes the optimized routine a pre-configured number of times to ensure the evaluation time is always above clock resolution, and then reports the elapsed time and the MFLOPS achieved across multiple runs of invoking the targeting routine. For this paper, all the search heuristics use the reported MFLOPS as performance feedbacks from empirical evaluation.

4.2. Evaluating Optimization-specific Heuristics

Our TA search algorithm uses three heuristics to significantly prune the tuning space: tuning individual optimizations one after another in a predetermined order, significant pruning of the cache blocking optimization space, and



TA-default: use the default TA search heuristics;
TA-fineBlock: try consecutive blocking factors by increment of 2 instead of 16;
TA-recBlock: allow different dimensions of a loop nest to have distinct cache blocking factors;
TA-tune3: tune each optimization three times in a round-robin fashion;
TA-Par-Block: tune parallelization and cache blocking together as a group;
TA-Block-UJ: tune cache blocking and loop unroll-and-jam together as a group;

Figure 5: Best performance achieved by different TA search heuristics

# of evals	TA-default			TA-fineBlock			TA-recBlock		
small/large	gemm	gemv	ger	gemm	gemv	ger	gemm	gemv	ger
8-core Intel	338/392	171/258	165/260	359/811	235/644	237/681	1093/4151	248/656	247/659
4-core AMD	367/383	222/246	235/246	502/748	124/649	192/640	3260/4141	165/687	692/665
# of evals	TA-tune3			TA-Par-Block			TA-Block-UJ		
small/large	gemm	gemv	ger	gemm	gemv	ger	gemm	gemv	ger
8-core Intel	901/753	428/685	335/642	384/456	215/312	191/303	353/1025	184/415	184/429
4-core AMD	1035/735	508/639	630/642	409/407	200/303	158/289	521/1031	174/419	152/418

Table 2: Number of configurations tried by different search heuristics

maintaining a small constant number of top configurations after tuning each optimization. These heuristics are based on common practice (e.g., the search space for blocking is too large without significant pruning), domain-specific knowledge (e.g., parallelization should be tuned before sequential optimizations), and extensive experiments (e.g., different values are used to limit the number of top configurations before settling for the most cost-effective one).

To evaluate the effectiveness of these search heuristics, Figure 5 compares the best performance achieved via these heuristics with those achieved when using alternative more expensive search strategies. Table 2 shows the number of different optimization configurations tried when using different search strategies.

Tuning cache blocking. By default, when multiple loops are blocked for cache locality in a loop nest, our TA search algorithm assigns the same blocking factor to all loop dimensions. Further, it increases the current blocking factor by 16 each time to find the next value to try. In Figure 5, these heuristics are implemented by the *TA-default* search.

As shown in Figure 5, the performance loss by these heuristics is minimal in most cases when compared with alternatively using a finer-grained blocking factor increment (*TA-fineBlock*) or supporting a different blocking factor for each loop dimension (*TA-recBlock*). Using the same blocking factor for all dimensions of a loop nest is effective because most of the loops within a single nest are symmetric, e.g., they typically access different dimensions of an array but behave similarly otherwise. The three matrix kernels we pick certainly demonstrate this property. Although we may miss more desirable configurations by dramatically reducing the number of different blocking factors to try, the possibility is low as minor differences in the cache block size are usually insignificant.

Table 2 shows that the number of optimization configurations tried by the TA search increases significantly when using the alternative more expensive strategies (*TA-fineBlock* and *TA-recBlock*) except for *gemv* and *ger* using small matrices (100^2), where cache blocking has minimal performance impact due to the lack of data reuse. Note that the wildly differing number of configurations tried by the TA search is due to the dynamic pruning of top-configurations after tuning each optimization, discussed in Section 3.

In Figure 5, the more expensive strategies for tuning cache blocking have resulted in noticeable better performance in a few cases, e.g., using *TA-recBlock* for *gemm-large* and using *TA-fineBlock* for *gemv-small*. However, the performance difference is actually not a direct result of better tuning for cache blocking. From Figure 6, which shows the best performance achieved by the alternative strategies after tuning each optimization, the performance achieved immediately after tuning cache blocking is almost identical across different strategies, and the ultimate performance

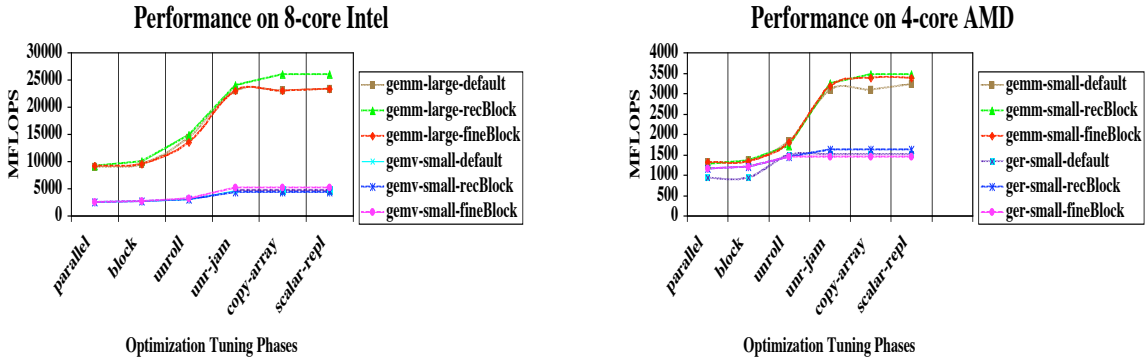


Figure 6: Best performance achieved at different optimization tuning phases

difference is the result of later interactions between cache blocking and other optimizations, specifically array copying (*gemm-large* and *gemm-small*) and unroll-and-jam (*ger-small* and *gemv-small*).

Ordering of tuning different optimizations. By default, our TA search algorithm tunes each optimization independently one after another, in the order enumerated by the horizontal axis of Figure 6. This default tuning order is determined after experimenting with various alternative orderings. Although no particular order is optimal, the one in Figure 6 has been highly effective when reasonable initial configurations are given for each optimization and when a sufficient number of top-configurations are maintained after tuning each optimization.

To evaluate the effectiveness of our optimization tuning order, Figure 5 compares the best performance it achieved with that achieved by alternatively tuning each optimization multiple (3) times in a round-robin fashion (*TA-tune3*). For all the benchmarks, the top configurations have stablized after tuning each optimization twice. From Figure 5, the performance improvement from the extra round of tuning is mostly minor except for *gemm-small* on the 4-core AMD and *gemv-large* on the 8-core Intel, where interactions between optimizations have resulted in dramatically different top-configurations being selected. Note that the tuning time (reflected by the number of trial evaluations) also increase significantly to find the better performance in these cases, shown in Table 2.

Interactions among optimizations. After tuning each optimization, our TA search selects at most 10 top configurations that are within 10% of each other in performance. While significantly reducing tuning time, this strategy sometimes fails to select a promising configuration which can result in dramatically better performance when a later optimization is tuned. These configurations can be recovered if the interacting optimizations are tuned together.

We have experimented grouping various optimizations to be tuned together and have identified three optimizations, OpenMP parallelization, cache blocking, and loop unroll-and-jam, as most likely to interact with each other. These optimizations target the multi-threading, memory, and register level performance respectively, and their configurations often need to be coordinated to be effective. Further, loop unroll-and-jam may interact with innermost loop unrolling, both of which impact register allocation. Cache blocking may occasionally interact with array copying, as shown in Figure 6 when different loop dimensions are given distinct blocking factors. Due to space constraints, Figure 5 shows only the interactions among parallelization, blocking, and unroll-and-jam.

In Figure 5, *TA-Par-Block* shows the result of tuning parallelization and blocking together, which dramatically enhanced the performance of *gemm* using small matrices. *TA-Block-UJ* shows the result of tuning blocking and unroll-and-jam together, which made noticeable performance improvement for *gemv-large* on the 8-core Intel machine. From Table 2, grouping these optimizations together does not significantly increase tuning time except for *gemm* using large matrices, so they are fairly cost-effective. However, grouping blocking and unroll-and-jam together can occasionally degrade performance due to interactions with loop unrolling. In particular, *TA-Block-UJ* has performed significantly worse than *TA-default* for *gemm-large* on the 4-core AMD because the best configurations from the combined tuning has resulted loop unrolling being turned off in the end. Note that loop unrolling was tuned in between blocking and unroll-and-jam in *TA-default* but moved to go after unroll-and-jam in *TA-Block-UJ*.

5. Related Work

The initial design of the POET language was published by Yi *et al.* [8]. Yi and Whaley demonstrated that by manually writing POET scripts to optimize several linear algebra kernels, they can achieve performance comparable to that achieved by manually written assembly in ATLAS [10]. Yi [7] extended a source-to-source optimizing compiler, the ROSE loop optimizer [6], to *automatically* produce parameterized POET scripts. Rahman, Guo, and Yi [11] used a similar auto-tuning infrastructure to tune the power consumption of applications. This paper extends the work by Yi [7] to present the empirical tuning infrastructure as a whole and to investigate the effectiveness of different search heuristics to explore the complex configuration space of the auto-generated POET scripts.

Empirical performance tuning has been used to build many successful scientific libraries, including ATLAS [12], PHiPAC [13], OSKI [14], FFTW [15], SPIRAL [16], among others, which use specialized kernel generators to parameterize and orchestrate differently optimized code. More recent research on *iterative compilation* has empirically modified the configurations of general-purpose compiler optimizations based on performance feedbacks [1, 2, 3, 4, 5]. While most of these compilers support parameterization of architecture-sensitive optimizations so that these optimizations can be reconfigured based on performance feedbacks, the parameterization is typically inside the compiler and cannot be easily controlled externally by developers or independent search engines. The work by Hall *et al.*[17] allows developers or search engines to provide a sequence of *loop transformation Recipes* to guide transformations performed by an optimizing compiler. The *X* language [18] uses C/C++ pragma to guide the application of a predefined collection of compiler optimizations. Instead of asking developers or search engines to guide transformations applied by a compiler, we adapt an optimizing compiler to output its optimization transformations to be perused by developers or independent search engines. The degree of parameterization in our auto-generated POET scripts is much more extensive than that supported by existing other approaches.

Previous autotuning research has adopted a wide variety of search algorithms, including both optimization-specific algorithms that are custom made for a tuning framework [12, 13, 14, 19] and generic algorithms that are oblivious of the optimizations being tuned [20, 2, 21, 22], combined with model-driven search where compiler models are used to prune the space before tuning [23, 2, 24, 25]. Seymour, You, and Dongarra [22] studied the relative efficiency of 6 different generic search algorithms in terms of their abilities to find the best performance candidates under varying time limits. We focus on studying a transformation-aware search algorithm and investigate the effectiveness of various optimization-specific heuristics when used to explore the configuration space of a large number of optimizations that interact with each other in unpredictable ways, using knowledge typically available within a compiler.

Static performance models have been used in both domain-specific tuning frameworks [19, 26] and general-purpose iterative compilation [24, 27, 3] to improve the efficiency of tuning. Chen, Chame, and Hall [24] used models within a compiler to prune the search space before using generic search algorithms to tune memory optimizations such as tiling, unroll-and-jam, array copying, and scalar replacement. Recent research has adopted predictive modeling from machine learning to statically build optimization models from a representative training set of programs [28, 29]. The learned models are then used to automatically determine optimization configurations for future applications without any additional tuning. We aim to identify important patterns of interacting optimizations to effectively prune the search space, but our focus is on identifying the performance tradeoffs of various optimization-specific heuristics.

6. Conclusions and Future Work

This paper presents a modular framework to support extensive parameterization and tuning of architecture-sensitive optimizations. Within out framework, optimization decisions by compilers are output as parameterized scripts in a program transformation language, POET, so that developers have full programmable control of the auto-generated scripts, and independent search engines can be used to explore the optimization configuration space. We have used our framework to apply 6 highly interactive optimizations, parallelization via OpenMP, cache blocking, array copying, unroll-and-jam, scalar replacement, and loop unrolling, and study the effectiveness of a number of commonly-adopted compiler heuristics in exploring their configuration space. Our framework can be applied to similarly optimize other regular scientific kernels such as triangular dense matrix solvers and stencil computations. To support irregular applications such as sparse-matrix computations and graph algorithms, a different set of optimizations need to be integrated within the ROSE analysis and the POET transformation engine, which belong to our future work.

[1] T. Kisuki, P. Knijnenburg, M. O'Boyle, H. Wijsho, Iterative compilation in program optimization, in: Compilers for Parallel Computers, 2000, pp. 35–44.
URL `citeseer.nj.nec.com/kisuki00iterative.html`

[2] A. Qasem, K. Kennedy, J. Mellor-Crummey, Automatic tuning of whole applications using direct search and a performance-based transformation system, The Journal of Supercomputing 36 (2) (2006) 183–196.

[3] N. Baradaran, J. Chame, C. Chen, P. Diniz, M. Hall, Y.-J. Lee, B. Liu, R. Lucas, Eco: An empirical-based compilation and optimization system, in: International Parallel and Distributed Processing Symposium, 2003.

[4] G. Pike, P. Hilfinger, Better tiling and array contraction for compiling scientific programs, in: SC, Baltimore, MD, USA, 2002.

[5] Z. Pan, R. Eigenmann, Fast automatic procedure-level performance tuning, in: Proc. Parallel Architectures and Compilation Techniques, 2006.

[6] Q. Yi, D. Quinlan, Applying loop optimizations to object-oriented abstractions through general classification of array semantics, in: The 17th International Workshop on Languages and Compilers for Parallel Computing, West Lafayette, Indiana, USA, 2004.

[7] Q. Yi, Automated programmable control and parameterization of compiler optimizations, in: CGO: ACM/IEEE International Symposium on Code Generation and Optimization, 2011.

[8] Q. Yi, K. Seymour, H. You, R. Vuduc, D. Quinlan, POET: Parameterized optimizations for empirical tuning, in: Workshop on Performance Optimization for High-Level Languages and Libraries, 2007.

[9] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by simulated annealing, Science, Number 4598, 13 May 1983 220, 4598 (1983) 671–680.

[10] Q. Yi, C. Whaley, Automated transformation for performance-critical kernels, in: ACM SIGPLAN Symposium on Library-Centric Software Design, 2007.

[11] S. F. Rahman, J. Guo, Q. Yi, Automated empirical tuning of scientific codes for performance and power consumption, in: HIPEAC:High-Performance and Embedded Architectures and Compilers, Heraklion, Greece, 2011.

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

[13] J. Bilmes, K. Asanovic, C.-W. Chin, J. Demmel, Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology, in: Proc. the 11th international conference on Supercomputing, ACM Press, New York, NY, USA, 1997, pp. 340–347. doi:http://doi.acm.org/10.1145/263580.263662.

[14] R. Vuduc, J. Demmel, K. Yelick, OSKI: An interface for a self-optimizing library of sparse matrix kernels, `bebop.cs.berkeley.edu/oski` (2005).

[15] M. Frigo, S. Johnson, FFTW: An Adaptive Software Architecture for the FFT, in: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 3, 1998, p. 1381.

[16] 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, N. Rizzolo, SPIRAL: Code generation for DSP transforms, IEEE special issue on Program Generation, Optimization, and Adaptation 93 (2).

[17] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, M. M. Khan, Loop transformation recipes for code generation and auto-tuning, in: LCPC, 2009.

[18] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, K. Pingali, A language for the compact representation of multiple program versions, in: LCPC, 2005.

[19] B. Fraguela, Y. Voronenko, M. Puschel, Automatic tuning of discrete fourier transforms driven by analytical modeling, in: PACT'09: Parallel Architectures and Compilation Techniques, Raleigh,NC, 2009.

[20] Z. Pan, R. Eigenmann, Fast and effective orchestration of compiler optimizations for automatic performance tuning, in: The 4th Annual International Symposium on Code Generation and Optimization, 2006.

[21] H. You, K. Seymour, J. Dongarra, An effective empirical search method for automatic software tuning, Tech. Rep. ICL-UT-05-02, Dept. of Computer Science, University of Tennessee (May 2005).

[22] K. Seymour, H. You, J. Dongarra, Comparison of search heuristics for empirical code optimization, in: iWapt:International Workshop on Automatic Performance Tuning, 2008.

[23] A. Tiwari, C. Chen, J. Chame, M. Hall, J. K. Hollingsworth, A scalable auto-tuning framework for compiler optimization, in: IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–12. doi:http://dx.doi.org/10.1109/IPDPS.2009.5161054.

[24] C. Chen, J. Chame, M. Hall, Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy, in: International Symposium on Code Generation and Optimization, 2005.

[25] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, D. Jones, Fast searches for effective optimization phase sequences, in: PLDI: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, ACM, New York, NY, USA, 2004, pp. 171–182. doi:http://doi.acm.org/10.1145/996841.996863.

[26] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, A comparison of empirical and model-driven optimization, IEEE special issue on Program Generation, Optimization, and Adaptation.

[27] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, N. Vasilache, Facilitating the search for compositions of program transformations, in: ICS '05: Proceedings of the 19th annual international conference on Supercomputing, ACM, New York, NY, USA, 2005, pp. 151–160. doi:http://doi.acm.org/10.1145/1088149.1088169.

[28] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, C. K. I. Williams, Using machine learning to focus iterative optimization, in: CGO: Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, Washington, DC, USA, 2006, pp. 295–305. doi:http://dx.doi.org/10.1109/CGO.2006.37.

[29] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, M. F. P. O'Boyle, Portable compiler optimisation across embedded programs and microarchitectures using machine learning, in: MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, ACM, New York, NY, USA, 2009, pp. 78–88. doi:http://doi.acm.org/10.1145/1669112.1669124.