Evaluating the Role of Optimization-Specific Search Heuristics in Effective Autotuning*

Jichi Guo[1] Qing Yi[1] Apan Qasem[2]

[1] University of Texas at San Antonio
[2] Texas State University

Abstract. The increasing complexities of modern architectures require compilers to extensively apply a large collection of architecture-sensitive optimizations, e.g., parallelization and memory locality optimizations, which interact with each other in unpredictable ways. The configuration space of these optimizations are exceedingly large, and heuristics for exploring the search space routinely end up settling for suboptimal solutions. We present a transformation-aware (TA) search algorithm which effectively combines optimization-specific heuristics (i.e., heuristics a compiler could use) to explore the configuration space of six optimizations, parallelization via OpenMP, cache blocking, array copying, unroll-and-jam, scalar replacement, and loop unrolling, to improve the multithreading, memory system and microprocessor performance of three linear algebra routines. We compare the effectiveness of the TA-search algorithm with generic search algorithms (e.g., hill climbing, simulated annealing) commonly adopted in autotuning research and empirically study the performance tradeoffs of various optimization-specific heuristics when used to speed up the search process.

1 Introduction

As modern computers become increasingly complex, it has become exceedingly difficult for compilers to statically predict the performance impact of architecture-sensitive optimizations such as OpenMP parallelization, tiling, unroll-and-jam, and loop unrolling, where the profitability of optimizations critically depends on detailed interactions of many architectural components. Empirical tuning has become a predominant approach to automatically achieve portable high performance on varying architectures. In particular, a large number of frameworks have focused on empirically exploring the optimization space of both domain-specific libraries[21, 3, 20, 8, 13] and general-purpose applications[11, 16, 5, 19, 2, 15, 22, 14], using a variety of different search algorithms[14, 17, 27, 19, 17, 4, 12].

This paper focuses on utilizing optimization-specific knowledge typically available within a compiler to better explore the search space in autotuning. We present a transformation-aware (TA) search algorithm which effectively integrates several optimization-specific heuristics to explore the configuration space

* This research is funded by the National Science Foundation under Grant No. 0833203 and No. 0747357 and by the Department of Energy under Grant No. DE-SC001770

of six architecture-sensitive optimizations, loop parallelization via OpenMP, blocking, array copying, loop unroll-and-jam, scalar replacement, and loop unrolling, to simultaneously improve the parallel, memory hierarchy, and microprocessor level performance for three linear algebra kernels. We evaluate the effectiveness of these heuristics by comparing them with three commonly adopted generic search algorithms, simulated annealing, hill climbing, and random search, from the PSEAT project[24]. Further, we conduct a series of experiments to study the performance tradeoffs of these heuristics when used to speed up the search.

Previous research has proposed many different search heuristics for efficient tuning. However, as yet, no strategy has emerged as a clear winner. On the one hand, optimization-specific search algorithms have used expert knowledge specific to an empirical tuning system to statically determine ranges of parameter values to try[21, 3, 20, 8, 13]. While effective, these heuristics are not easily applicable beyond their tuning frameworks. On the other hand, strategies based on intelligent search algorithms such as genetic algiorithms and simulated annealing[14, 17, 27, 18] use the *fitness* or objective function as the sole criteria for finding the best value. These strategies are more widely applicable but operate at a level that abstracts away many of the underlying search space characteristics. While model-driven search[19, 12, 4, 17] has investigated various model-based compiler heuristics to prune the search space before invoking generic algorithms, the generic search mostly remains oblivious of any compiler knowledge.

While a plethora of both optimization-specific and generic search algorithms exist, a comparison of their effectiveness has not been well studied. Further, the performance tradeoffs of various compiler heuristics are not well understood when using them to drive the exploration of an extremely large, complex, and randomly interacting multi-dimensional optimization space. Our TA search belongs to the optimization-specific category and is based on a number of heuristics commonly-adopted by compilers. Through this algorithm, 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 search time without sacrificing application performance. Our main contribution is an empirical study that provide insight to the answers of the following open questions.

- Whether optimization-specific search algorithms (e.g., our TA search) can be more effective than commonly adopted generic search algorithms. What are the advantages and disadvantages of these algorithms.
- 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. How sensitive are the various heuristics when facing unpredictable interactions between different optimizations.

Our experimental results show that our TA search algorithm outperforms three generic search algorithms in almost all cases. So our answers to the first two questions are a probable *yes*. However, we also find that interactions between different optimizations have routinely caused search heuristics to settle

for suboptimal solutions. In the following, we discuss related work, introduce more details about our transformation-aware search algorithm, and then present experimental results to better answer each of the above questions.

2 Related Work

Empirical tuning has been used to successfully achieve portable high performance for a number of domain-specific frameworks[21, 3, 20, 8, 13], where specialized kernel generators and search drivers are used to empirically determine the best configurations of manually orchestrated optimizations. Recent research on *iterative compilation* has adopted empirical tuning to modify the configurations of compiler optimizations based on performance feedbacks of the optimized code[11, 16, 5, 19, 2, 15, 22, 14, 7]. This paper focuses on the search portion of empirical tuning. In particular, we compare the efficiency of several search algorithms in exploring the configuration space of a large number of interacting performance optimizations. The search algorithms could be used both by domain-specific tuning systems and by general-purpose iterative compilers.

Previous autotuning research has adopted a wide variety of search algorithms, including both optimization-specific algorithms that are custom made for a tuning framework [21, 3, 20, 7] and generic algorithms that are oblivious of the optimizations being tuned[14, 17, 27, 18], combined with model-driven search where compiler models are used to prune the space before tuning[19, 17, 4, 12]. Seymour, You, and Dongarra[18] studied the relative efficiency of 6 different generic search algorithms: Orthogonal, Nelder-Mead Simplex, Genetic Algorithms, Simulated Annealing, Particle Swarm Optimization, and Random search, in terms of their abilities to find the best performance candidates under varying time limits. They found that random search performed fairly well, and that Particle Swarm Optimization has a slight edge over the other search algorithms. We focus on studying a transformation-aware search algorithm and comparing it with three more generic search algorithms, the direct search (a hill climber), simulated annealing, and random search. We investigate the effectiveness of several 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[7, 26] and general-purpose iterative compilation[4, 5, 2] to improve the efficiency of tuning. Similar to our work, Chen, Chame, and Hall[4] 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. In contrast, we integrate optimization-specific heuristics within a custom-built transformation-aware search algorithm instead of using generic search algorithms. Further, we evaluate the performance tradeoffs of these heuristics through extensive empirical studies.

Recent research has adopted predictive modeling from machine learning to statically build optimization models from a representative training set of pro-

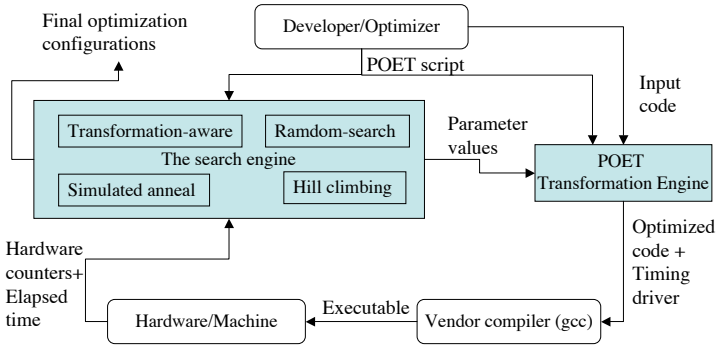


Fig. 1. The Tuning Infrastructure

grams[1, 6]. The learnt models are then used to automatically determine optimization configurations for future applications without any additional tuning. We share a similar goal in that we also try 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.

3 Tuning Infrastructure

Our tuning framework is shown in Figure 1 and includes two main components: the POET transformation engine and the optimization search engine. To use the framework, a developer needs to provide two inputs: the source code to be optimized and a POET transformation script that can be invoked to apply a collection of parameterized program transformations to optimize the input code.

POET is an interpreted program transformation language designed for parameterizing general-purpose compiler optimizations for auto-tuning[25]. The transformation engine in Figure 1 is essentially a POET language interpreter coupled with a set of transformation libraries and input/output language syntax descriptions. To optimize an input program, a POET script needs to be written which specifies 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 transformation engine.

The search engine takes a single input, the POET optimization script to tune, and orchestrates the whole tuning process by iteratively determining what parameter values to use to properly configure each optimization, invoking the transformation engine 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. Four different search algorithms are currently supported in the search engine, including our transformation-aware search and three generic search algorithms, simulated annealing, direct search (a hill climber), and random search, from the PEAT project[24].

The following first describes the optimization space currently supported by our search engine and then presents details of the search algorithms.

3.1 The Optimization Space

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).

When used as input for our search engine, each POET script applies the above optimizations in the order that they are discussed above. A set of standardized parameter declarations are included in the POET script and are automatically extracted by the search engine to be used as input to the various search algorithms to define the optimization search space.

3.2 The Transformation-Aware Search Algorithm

Figure 2 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. In contrast to the other generic search algorithms which look to find the maximal/minimal points in a multi-dimensional generic space, our TA search algorithm is optimization-specific in that it understands the meaning and context of each optimization parameter,

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 $cur\_tune = \emptyset$; /* Set up the current tuning space. */
For (each $config \in config\_res$ and each loop nest L being optimized by $config$):
$cur\_tune = cur\_tune \cup \{gen\_tune\_space(tuneParams(L), cur\_opt, config)\}$;
$cur\_config = gen\_first\_config(cur\_tune)$;
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 $config\_res$):
$config\_res = config\_res \cup \{(cur\_config, cur\_score)\}$;
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 ($cur\_config \neq null$): go to Step3.
Step6: cur_opt = next_entry($cur\_opt, opts$); /* try to tune the next optimization*/
If ($cur\_opt \neq null$): go to Sep 2;
Step7: return $config\_res$; /* return result */

Fig. 2. The transformation-aware search algorithm

and uses built-in heuristics to efficiently explore the configuration space. The search algorithm assumes full knowledge of how each POET parameter is used to control the optimizations in Section 3.1, and tunes their configuration space in a deliberate fashion, through the following steps.

First, 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. All the optimizations supported by the framework are then tuned independently one after another in a predetermined order. Note that the order of tuning individual optimization when collectively exploring their configuration space is contained entirely within the TA search algorithm and is different from the order of applying these optimizations in the POET scripts (discussed in Section 3.1). In Figure 2, the default tuning order is contained in the variable *opts* 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 impacts 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 4.3 evaluates the effectiveness of the above strategies (together with other varying heuristics).

Following the predetermined tuning order, the algorithm in Figure 2 iterates over steps 2-6 until all the optimizations have been tuned. In particular, 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. Step 2 of the algorithm 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 and 6 can skip optimization configurations that are known to have a negative impact based on previous experiments.

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 4.3 studies the effectiveness and performance tradeoffs of these heuristics.

3.3 The Generic Search Algorithms

Our search engine currently supports three generic search algorithms, direct search (a hill climber), simulated annealing [10], and random search, from the PSEAT project[24]. We provide brief descriptions of these algorithms below.

Direct Search. The variant of direct search implemented in our framework is the *pattern-based method*, originally proposed by Hooke and Jeeves [9]. This algorithm works on a discrete space and aims to find the optimal point in the search space using a method of steepest descent. The algorithm proceeds by making a set of *exploratory moves* and *pattern moves*. By visiting neighboring locations, the exploratory moves identify a promising direction of movement from the current position. The search then takes a leap in that direction (pattern move) and then explores neighbors of that new location. This process continues until the exploratory moves fail to find a new promising direction for some point. This point is returned as the optimal location.

Simulated Annealing Initially a random point is selected in the search space and its neighboring points are explored. At each step, the search moves to a point with the lowest value or depending on the current *temperature*, to a point with a higher value. The temperature is decreased over time and the search converges when no further moves are possible.

Random Search A random search picks random points within the search space and keeps track of the best value found at every step. Unlike the other search strategies described above, random search does not use any heuristics and it does not have any convergence criteria. The search is terminated after a pre-specified number of evaluations.

4 Experimental Results

The goal of this paper is to evaluate the effectiveness and performance tradeoffs of various optimization-specific heuristics when used in our transformation-aware search algorithm, described in Section 3.2. In particular, we aim to provide answers or insights to the following open questions.

- Whether the TA search algorithm can achieve better performance than other generic algorithms commonly used in autotuning research. What are the advantages and disadvantages of these search algorithms.
- How effective are the various strategies used in the TA search algorithm. What are their performance tradeoffs. In particular, whether tuning one optimization at a time can effectively explore the search space, and how should various optimizations be ordered.
- How sensitive are the various heuristics when facing complex interactions between different optimizations. What are the common patterns of interacting optimizations. What alternative heuristics can be used to improve the effectiveness of the TA search.

To quantify performance tradeoffs of various heuristics when used to explore the search space, we need to obtain the best performance possible through the optimizations we currently support. While an exhaustive search of the entire optimization space would extract the information, it is out of the question due to the astronomic amount of time required (in particular, it took our machine one week to explore about 30% of the whole optimization space for a vector-vector multiplication kernel). 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.

4.1 Experimental Design

We have evaluated all the search algorithms 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.

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*

- All of them are computationally intensive, and their efficiency can be improved significantly via the collection of source-to-source optimizations (e.g., loop optimizations) supported by our search engine.
- The three benchmarks vary widely in their computation/data-access rates. 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 a C/C++ optimizing compiler [23]. The optimizations applied to each benchmark are discussed in Section 3.1. We tuned each benchmark using both small (100^2) and large (1000^2) matrices. Table 1 shows the optimization search Space for each of the benchmark kernels.

We tuned each benchmark on two multi-core machines: a quad-core machine running Linux with two dual-core 3 GHz AMD Opteron Processors (1KB L1 cache), and an eight-core machine running MacOS with two quad-core 2.66 GHz Intel processors (32KB L1 cache). We used PAPI to collect values of hardware counters on the AMD Opteron. However, because PAPI does not support MacOS, we were not able to collect hardware counter values on the Intel Mac. 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. 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, the MFLOPS achieved, and the median of hardware counter values across multiple runs of invoking the targeting routine. For this paper, all the search algorithms used the reported MFLOPS as performance feedbacks from empirical evaluation.

4.2 Comparing Transformation-aware and Generic Search Algorithms

Figure 3 shows the best performance attained when using the four search algorithms, transformation-aware (*TA-default*), simulated annealing, direct search

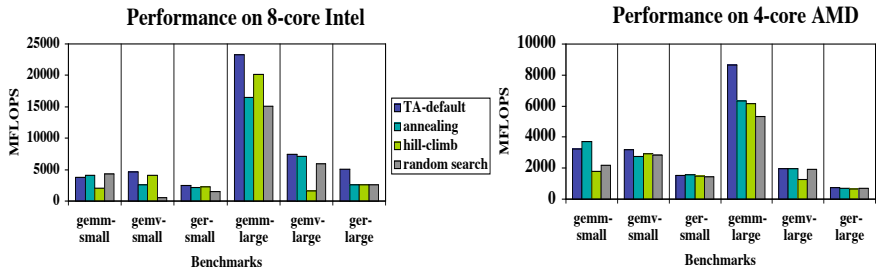


Fig. 3. Best performance achieved by different search algorithms

(hill-climb), and random search, to explore the optimization space of each benchmark. Table 2 shows the number of configurations tried by each search algorithm when using both small and large matrices for each benchmark. Note that all the generic algorithms (i.e., annealing, hill-climb, and random) have been forced to terminate after evaluating 500 different configurations of the optimizations. From Table 2, *annealing* is the only generic search algorithm that has converged within 400 iterations (both *hill-climb* and *random* search algorithms have tried the full 500 iterations). In contrast, all the TA searches have terminated before reaching 400 evaluations. As a result, the TA searches have generally taken much shorter tuning time than the other generic search algorithms.

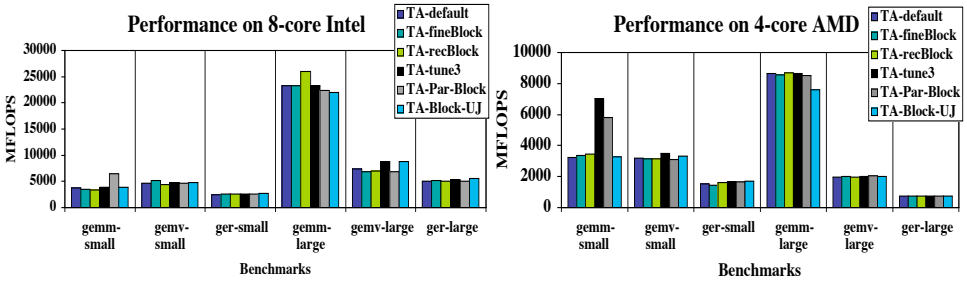
From Figure 3, the TA search algorithm has performed similarly as or better than all the other generic search algorithms in all cases except for *gemm* using small (100^2) matrices, where *TA-default* lags slightly behind *annealing*. The results on both machines are fairly consistent. The performance comparison among the three generic search algorithms are consonant with conclusions by Seymoure et. al[18]. In particular, random search has performed reasonably well except for *gemv* using small matrices, *annealing* has done mostly well except for *gemv-small* and *gemm-large*, and *hill-climb* has performed well in some cases but failed by a large margin in some others (e.g., *gemv-large* on the 8-core Intel).

Generic search algorithms are advantageous over optimization-specific ones as they are more general and readily applicable to arbitrary contiguous spaces. However, the lack of domain-specific knowledge about the search space could seriously detriment their effectiveness when exploring the enormously complex configuration space of heavily interacting optimizations. For example, since all the *annealing* searches have converged within 350 iterations, the low performance achieved is likely due to the search algorithm getting stuck at a local minimum. On the other hand, the low performance by *hill-climb* was due to the search spending too much within the same neighboring region (a histogram of visited points revealed this pattern).

Our TA search algorithm is based purely on optimization-specific heuristics. In particular, the algorithm completely ignores the local performance relations between neighboring points. It strictly follows statically-determined heuristics to explore the configuration space of each optimization in a pre-determined order, and the tuning space is dynamic adjusted only in-between the tuning of different optimizations. Therefore, the TA search is not affected by the local minima of the

# of evals	annealing			hill-climb			random		
small/large	gemm	gemv	ger	gemm	gemv	ger	gemm	gemv	ger
8-core Intel	350/350	350/350	350/350	500/500	500/500	500/500	500/500	500/500	500/500
4-core AMD	350/350	350/350	350/350	500/500	500/500	500/500	500/500	500/500	500/500
# 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



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;

Fig. 4. Best performance achieved by different TA search heuristics

search space. And as it can use expert knowledge about various optimizations to more effectively explore the search space (e.g., it can distinguish different dimensions), it usually needs to evaluate much fewer configurations than the generic search algorithms and is therefore more efficient.

4.3 Evaluatng Optimization-specific Heuristics

To reduce tuning time, our TA search algorithm uses three main heuristics, a statically determined tuning order for optimizations, significant pruning of the search space for cache blocking, and maintaining only a small constant number of top configurations after tuning each optimization. We have selected these strategies 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 selected after tuning each optimization before settling for the most cost-effective one).

While these optimization-specific heuristics are shown to be effective when compared with the three generic search algorithms in Figure 3, they are far from optimal. In particular, interactions between different optimizations have routinely triggered the TA algorithm to behave unexpectedly, which can be ob-

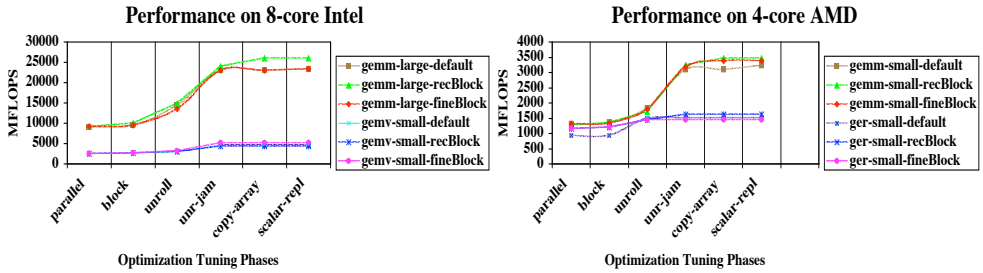


Fig. 5. Best performance achieved at different optimization tuning phases

served when alternative search strategies are used, shown in Figure 4 and discussed below. Table 2 shows the number of different optimization configurations tried when using the alternative search strategies.

Tuning the cache blocking optimization. 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 4, these heuristics are implemented by the *TA-default* search.

As shown in Figure 4, 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 individual blocking factors 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. Further, although we may miss optimization opportunities by dramatically reducing the number of different blocking factors to try, the risk 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 the alternative strategies (*TA-fineBlock* and *TA-recBlock*) are used 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 3.2.

In Figure 4, the alternative 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 5, 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 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 shown in Figure 5. The default tuning order is determined after experimenting with various alternative orderings. Although no particular order is optimal, the one in Figure 5 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 4 compares it with 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 4, 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 number of evaluations also increase significantly to find the better performance in these cases, shown in Table 2.

Interactions between 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 5 when different loop dimensions are given distinct blocking factors. Due to space constraints, Figure 4 shows only the interactions among parallelization, blocking, and unroll-and-jam.

In Figure 4, *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 Conclusions

This paper presents a transformation-aware search algorithm which integrates several optimization-specic heuristics to effectively explore the enormously complex configuration space of six interacting architecture-sensitive optimizations to simultaneously improve the parallel, memory hierarchy, and microprocessor level performance for a number of linear algebra kernels. We evaluate the effectiveness of these heuristics by comparing them with three commonly adopted generic search algorithms and conduct a series of experiments to study their performance tradeoffs when used to speed up the search process.

References

1. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.
2. N. Baradaran, J. Chame, C. Chen, P. Diniz, M. Hall, Y.-J. Lee, B. Liu, and R. Lucas. Eco: An empirical-based compilation and optimization system. In *International Parallel and Distributed Processing Symposium*, 2003.
3. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proc. the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.
4. C. Chen, J. Chame, and 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*, San Jose, CA, USA, March 2005.
5. A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160, New York, NY, USA, 2005. ACM.
6. C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and 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*, pages 78–88, New York, NY, USA, 2009. ACM.
7. B. Fraguela, Y. Voronenko, and M. Puschel. Automatic tuning of discrete fourier transforms driven by analytical modeling. In *PACT'09: Parallel Architectures and Compilation Techniques*, Raleigh,NC, Sept. 2009.
8. M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.
9. R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. In *J. ACM*, pages 212–229, 1961.

10. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.
11. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *Compilers for Parallel Computers*, pages 35–44, 2000.
12. P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 171–182, New York, NY, USA, 2004. ACM.
13. J. Moura, J. Johnson, R. Johnson, D. Padua, M. Puschel, and M. Veloso. Spiral: Automatic implementation of signal processing algorithms. In *Conference on High-Performance Embedded Computing*, MIT Lincoln Laboratories, Boston, MA, 2000.
14. Z. Pan and 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.
15. G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *SC*, Baltimore, MD, USA, November 2002.
16. A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th ACM International Conference on SuperComputing (ICS06)*, June 2006.
17. A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, 36(2):183–196, 2006.
18. K. Seymour, H. You, and J. Dongarra. Comparison of search heuristics for empirical code optimization. In *iWapt:International Workshop on Automatic Performance Tuning*, 2008.
19. A. Tiwari, C. Chen, J. Chame, M. Hall, and 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*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.
20. R. Vuduc, J. Demmel, and K. Yelick. OSKI: An interface for a self-optimizing library of sparse matrix kernels, 2005. `bebop.cs.berkeley.edu/oski`.
21. R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.
22. R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *34th International Conference on Parallel Processing*, pages 89–98, Oslo, Norway, 2005. IEEE Computer Society.
23. Q. Yi. Automated programmable code transformation for portable performance tuning. Technical Report CS-TR-2010-002, Computer Science, University of Texas at San Antonio, 2010.
24. Q. Yi and A. Qasem. Exploring the optimization space of dense linear algebra kernels. In *The 21th International Workshop on Languages and Compilers for Parallel Computing*, Edmonton, Alberta, Canada, Aug. 2008.
25. Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.
26. K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. A comparison of empirical and model-driven optimization. *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.
27. 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.