Automatic Tuning of Whole Applications Using Direct Search and a Performance-based Transformation System *

Apan Qasem Ken Kennedy John Mellor-Crummey

Department of Computer Science
Rice University
Houston, TX
{qasem,ken,johnmc}@cs.rice.edu

Abstract. In many cases, simple analytical models used by traditional compilers are no longer able to yield effectively optimized code for complex programs because of the enormous complexity of processor architectures. A promising alternative approach for optimizing applications effectively has been the use of search-based empirical methods. The success of empirically tuned library generators such as ATLAS has shown that this strategy can be effective for domain-specific programs. However, to date there has been no general-purpose tool for effective empirical optimization of whole programs. The main obstacle to this approach has been the need for evaluating a prohibitively large number of alternative program variants. To address this problem, we have developed a prototype tool for automatic application tuning that uses loop-level performance feedback and a direct search strategy to guide search for the best set of optimization parameters. Experiments on four different architectures show that direct search can be an effective technique for finding good values for transformation parameters in a reasonable time.

1 Introduction

Over the last several decades, the complexity of microprocessor architectures has grown considerably. Today, microarchitectures are so complex that using static models it is difficult to predict how different features will interact during program execution. As a result, it has become increasingly difficult for compilers to choose the proper set of optimizations to yield the best program performance. Parameters for program transformations such as tiling and unrolling are sensitive not only to the input program but also to the underlying architecture. Moreover, many of these transformations interact with each other in complex ways. Because of all these complex interactions simple analytical models used by traditional compilers fail to generate fast code across architectures.

* This material is based on work supported by the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and/or 12783-001-05 49 from the Los Alamos National Laboratory.

As a result, programmers often have to manually transform their code to get the desired performance on a specific architecture. However, this process of manual tuning of programs is time consuming, tedious and error prone. Not to mention that repeated manual transformation of the code makes it unmaintainable. In recent years, a novel alternative to manual tuning has been the the use of empirically tuned libraries. In an empirical compilation system the parameters for program transformations are not chosen using static models. Instead, programs with different optimization parameters are executed on the target machine and the program variant that gives the best performance is selected. Empirically tuned libraries such as ATLAS, are known to produce better code than native compilers across a range of modern architectures and are recognized as viable alternatives to hand-transformation in their respective domains.

In spite of the success of empirically tuned libraries, to date there has been no general-purpose tool for automatically tuning whole programs using empirical methods. The principal bottleneck in this regard has been the time for evaluating a prohibitively large number of program variants. Empirically tuned library generators have the advantage of focusing on a specific problem domain. As such, they can use domain specific knowledge to prune the search space to reduce the number of program variants that must be considered. A recent study by Yotov et. al. [19] show that for specific domains, carefully constructed static models can give performance that is comparable to empirically tuned libraries. However, this is unlikely to be the case for tuning arbitrary whole programs since a tuning tool will have to consider a large number of transformations and it cannot make assumptions about the type of program it will encounter. Hence, for tuning arbitrary programs, an empirically-driven approach may be the only effective approach.

One of the reasons that empirically-based program tuning has been so costly is because as yet there has been no suitable search strategy for exploring the large and complex search space. Although genetic algorithms have been used with some success in the context of the phase-ordering problem [3, 9], they are not suitable when searching for unconstrained transformation parameters such as tiling sizes and unroll factors. In this paper, we describe a framework for empirically-based program tuning that uses direct search [6] to find optimal parameters for the two transformations: tiling [17] and unroll-and-jam [2]. Our goal has been to develop a general-purpose framework for empirically-based program tuning that can deliver performance without having to depend on domain-specific properties of programs. An important component of this tool is a search strategy that can explore the complex optimization space effectively and efficiently.

In the sections that follow, we discuss related work, give an overview of our framework for empirically-based program tuning, describe the direct search algorithm, present results of experiments using the framework, and finally discuss our conclusions and future plans.

2 Related Work

A number of empirically-tuned libraries have been successful in delivering high-performance in their respective domains. ATLAS [15] produces highly optimized BLAS routines by probing the underlying hardware for platform specific information and using a global search to find the best transformation parameters. SPIRAL [18] and FFTW [4] use empirical techniques and mathematical properties of signal processing algorithms to choose an optimal algorithm from a suite of algorithms. PhiPAC [1] uses a parameterized code generator that generates portable C code for matrix multiply that achieves close to peak performance on a range of architectures. The main distinction between our approach and these empirically tuned libraries is that we do not attempt to exploit properties of any specific domain. Our goal is to develop an empirical tuning strategy for general scientific applications.

There has been some work in employing empirically-based tuning methods in general purpose compilers. Cooper et. al. [3] use genetic algorithms to find the best sequence of compiler passes. Kulkarni et. al. [9] have described efficient ways of reducing the running time of the search algorithms. Our approach is different from theirs, in that we look at optimization parameters rather than transformation sequences.

Fursin et al. [5] describes using empirically-based methods to select the best unroll, blocking and padding factors. The two search strategies that they use are sequential and random search. The sequential search is augmented by selecting only those loops that dominate program execution and searching for the best parameters in separate phases. Their results showed that there was no significant advantage of using sequential search over random search. Previous work by Kisuki et. al. [8] showed that in finding transformation parameters, random search performs as well as other sophisticated techniques such as genetic algorithms and simulated annealing. Part of our motivation for this work has been to find a search strategy that would work better than random search.

Wolf et. al. [16] describes using static performance estimators to find the best combination of parameters for loop transformations such as fusion, unrolling and tiling. Although their approach is not strictly empirical it does show that choosing the best combination of high-level transformations can significantly improve a program's performance. The only limitation to their approach is that their performance improvement is bounded by the accuracy of static predictors. Given the increasing complexity of the processor architecture it is unlikely that any static predictor will give results that are as accurate as actually running the program on the target machine.

The OSE compiler organization described by Triantafyllis et. al. [14] is perhaps the most practical approach to adopting iterative compilation strategies to a real compilation system. They use static models available in Intel's high-level optimizer to prune the space that is searched by their search module which significantly reduces the compilation time. However, the parameters they consider have either boolean values that either enable or disable a transformation or they have a set of 4-5 legal values that control how aggressively a transformation is

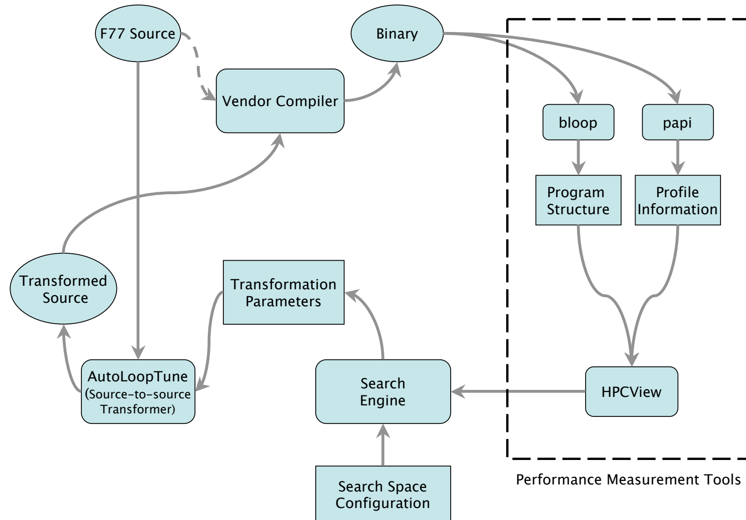

Fig. 1. Overview of Iterative Framework

applied. We consider parameters whose values can potentially be any integer and hence we have to deal with a much larger search space. Of course, the OSE approach to pruning the search space can be beneficial for our strategy as well and that is part of our future plans.

3 A Framework for Empirically-based Program Tuning

Figure 1 gives an overview of our iterative compilation framework. The major components of the framework include a source-to-source transformer (AutoLoopTune), a set of performance measurement tools, and a search module that uses the measurements to guide selection of program transformations. We use these tools in conjunction with a native compiler for an architecture that will perform low-level optimization of the source-level program variants generated by AutoLoopTune as it transforms them into machine code. On each pass through the framework, the search module generates a set of transformation parameters that are applied to the program by AutoLoopTune. The program is then compiled using the native compiler and run on the target machine. During execution, performance measurement tools collect metrics to feed to the search module. The search module will use these metrics in combination with results from previous passes to generate the next set of tuning parameters. This process continues until some pre-specified optimization time limit has been reached.

Although the structure of our framework is not dramatically different from that of other systems for empirical tuning, there are several key ideas that make

our framework unique. Among them is our use of loop-level performance measurements, selection of tuning parameters at the loop level and our ability to search for optimization parameters for independent code regions simultaneously. The rest of the section discusses the core components of our framework in some detail.

3.1 Transformation Tool

We implemented a source-to-source transformation tool (AutoLoopTune) [12] that is capable of performing a large class of high level transformations. The transformations supported by AutoLoopTune include tiling, unroll-and-jam, fusion, array contraction, and iteration space splicing. In designing AutoLoopTune the safety and profitability analyses have been kept completely separate from each other. This decoupled structure together with its ability to process source code directives at loop level granularity makes AutoLoopTune an ideal candidate for use with an iterative compilation system.

Currently, our search strategy uses only two of the transformations supported by AutoLoopTune, namely tiling and unroll-and-jam (inner loop unrolling included). In the future, we plan to use our search strategy to pick the best parameters for each of the transformations supported by AutoLoopTune. The parameters for tiling and unrolling are specified using directives in the source code. Each loop in the program can have a separate tiling and unrolling factor and any integer value for these parameters is considered legal. AutoLoopTune reads in a source file annotated with directives, applies the transformations and produces a transformed source file. An error message is sent to the search module if legality issues prevent AutoLoopTune from applying any of the specified transformations.

3.2 Performance Measurment Tools

We use tools from the HPCToolkit performance analysis toolkit [11] to gather loop-level performance metrics to guide tuning. HPCToolkit can collect metrics using hardware performance counters during a program's execution and then compute aggregate metrics for each loop in the program. HPCToolkit aggregates metrics at the loop level by analyzing an application's executable, recovering the control flow graphs (CFGs) for its procedures, applying interval analysis to recover information about loops in each CFG, and using symbol table information to determine the statements within each loop. This information is then delivered to the search engine for analysis.

There are two reasons for choosing HPCToolkit over a simpler performance measurement tool that measures total execution time. First, we wanted the ability to measure performance at the loop level. For most numerical applications execution time is concentrated around a few core loop nests. To get best results each loop nest needs to be tuned individually using different sets of transformation parameters. In many cases, these loopnests are independent from each other in the sense that the effect of applying the optimizations can be evaluated on

a loop nest by loop nest basis. Loop level performance measurements collected by HPCToolkit gives us the ability to treat each loop individually and tailor the transformation parameters accordingly. Our current prototype tool does not use the information obtained at the loop level. In the future, this prototype will be extended to use loop level execution metrics to guide separate instances of the search on individual loop nests that are independent.

The second reason for using HPCToolkit in our framework is to be able to collect performance metrics other than total running time. For instance, performance metrics such as cache misses and pipeline stalls indicate causes of inefficiency; for this reason, they may be better metrics to use for guiding the search. We are currently exploring ways to use these performance metrics to help guide our search for the best parameters.

3.3 Search Module

At the heart of our framework is the search engine which integrates all of the components. The search engine uses a pattern-based direct search technique [6] to search for transformation parameters. The current implementation of the search module uses whole program execution time to guide the search process. The search module starts off by reading in a configuration file that describes the search space. The search space is described in terms of the loops that are to be tiled and/or unrolled and the corresponding range of values for each parameter. Currently, we pick the applicable transformations and the range of values for each transformation parameter by hand. Loops that dominate a large portion of the execution time and contain some reuse are chosen for tiling and unrolling. The tiling range is chosen to be between 1 and some fraction of the loop upper bound whereas unroll factors lie between 1 and 30. This method of selecting transformations and parameter ranges is somewhat arbitrary and understandably not the best way of doing it. In the future, we plan to use dependence analysis to select the applicable transformations and static models to generate the range of values for each transformation parameter.

The search module uses the information in the configuration file to generate the initial set of parameters and annotates the source with the appropriate directives. After the program has been transformed by AutoLoopTune and the transformed program has been run, the performance results are fed to the search module. The search module uses this information to generate the next set of parameters. This process continues until the search converges to a local minima or the pre-specified compilation time has been reached.

Although, our current implementation uses whole program execution time to guide the search, in the future we plan to run multiple instances of the search on independent code regions of the same program. Running multiple instances of the search on a single program can potentially cut down the number of times the program is executed.

4 Direct Search

Direct search methods for nonlinear optimization have been used by computation scientists for over four decades [10]. The main feature of direct search that sets it apart from other optimization techniques is that the decision making process in direct search is based solely on function evaluation. So, unlike the quasi-Newton methods, direct search does not require any derivative information to find a direction of steepest descent. It was this particular feature that motivated us to apply direct search to our problem domain. In the following subsections we discuss the benefits and problems associated with using direct search for our problem and describe the algorithm used in our compilation system in some detail.

4.1 Why Direct Search?

It has been observed by researchers [3] that the search space for optimization parameters is large enough to make iterative approaches based on exhaustive search impractical. For example, if we are taking the iterative approach to finding the best compilation sequence for a compiler that supports 10 transformations - a modest number by today's standards - then an exhaustive search would have to examine 10^{10} possible sequences. Even in the limited case of a few transformations, the search space can be quite large if we consider transformations whose parameters can vary. For transformations like loop unrolling, the optimization parameter determines how many times loops are to be unrolled in the program. Hence, the values for these transformation parameters can potentially be any integer. Moreover, these transformations are usually most effective when we allow each loop to have a different parameter for each transformation. So, even for a small kernel like a 1000x1000 matrix multiply, if we are considering tiling of two loops with tile sizes from 1 to 100 and unrolling of one loop with unroll factor from 1 to 20, we end up with a search space that contains 100 x 100 x 20 or 200,000 points. Evaluating the kernel at all these points could take several days even on a reasonably fast microprocessor. Our goal is to use direct search to reap most of the benefits possible with empirically-based program tuning while only exploring a small fraction of the search space of transformation parameters.

It is not just the size that makes exploring the transformation search space difficult. Studies have shown that the search space is neither smooth nor continuous [3, 7]. Transformations like tiling and instruction scheduling are highly sensitive to the underlying architectutre. Moreover, many of these transformations interact with each other in complicated ways. As a result, the characteristics of the search space varies from program to program, from platform to platform and even from one input set to the other. Building accurate models for this search space has become extremely difficult. Hence, in the absence of such modeling, a derivative-free search method becomes a good choice to explore this optimization space.

Knijnenburg et. al. [8] conducted an experiment where they explored the search space of tiling sizes and unroll factors for matrix-multiply using several

different search techniques. The search methods included simulated annealing, pyramid search, window search and a random search. An interesting and somewhat surprising result of that experiment was that random search performed as well (and in some cases even better than) the other more sophisticated methods. The explanation for this is that all the other search techniques assume certain properties to hold for the space that is being explored. For example, simulated annealing uses a pre-computed value called *temperature* when exploring the search space. At any time during the search, the decision to move to a new point is based not only on the current function value but also on the value of the *temperature* parameter. However, if the current *temperature* is computed without detailed knowledge about the search space then it may not be useful in guiding the search in the best direction. Similarly, both pyramid search and window search depend on certain properties of the search space. In using direct search to explore the transformation search space we wanted to step away from the search methods that uses some form of modeling and use a method that relies solely on function evaluations.

Like most other search techniques direct search is not guaranteed to find the global minima. So, the result obtained through direct search may not always be as good as the one obtained through an exhaustive search. However, given the long compilation time associated with an exhaustive search this might not be that big a penalty. Another issue with direct search is that it is known to take a long time to converge when dealing with a large number of parameters. (usually more than 10). [13]. This can potentially increase the compilation time to a point where it is no longer feasible to use direct search in an iterative compilation system. However, for our problem domain we do not necessarily need the search to converge to a local minima. By keeping track of the best value found so far we can stop the search after a prespecified number of iterations. The experimental results from Section 5 suggest that this approach can be useful in finding good values even in cases when we do not wait for the search to converge to a local minima.

4.2 The Algorithm

To explore the search space of tiling and unrolling factors, we use a version of the pattern-based direct search method first proposed by Hookes and Jeeves [6]. We introduce the following terms to describe the algorithm.

- N denotes an $n-dimensional$ search space, where each dimension represents a transformation parameter that is being tuned
- $p = (p_1, p_2, ...p_n)$ denotes a point in the search space where p_i is the value of the ith parameter
- $f(p_1, p_2, ...p_n)$ denotes the execution time for the program compiled with transformation parameters $p_1, p_2, ...p_n$
- s denotes the step size, this value determines the size of the subspace that is explored during the exploratory moves

Table 1. Benchmarks

Program	Description	LOC
advect3d	advection kernel from NCOMMAS code for weather modeling	545
lud	1000 x 1000 lu decomposition based on matrix vector multiply	131
mm	1000 x 1000 matrix-matrix multiply	35
vpenta	NAS kernel benchmark program	145
swim	weather prediction program from SPEC 2000 FP benchmark suite	282
mgrid	multi grid solver programs from SPEC 2000 FP benchmark suite	344

The goal of the search algorithm is to find a point $(p_1, p_2, ...p_n)$ in N such that $f(p_1, p_2, ...p_n)$ is minimized. The algorithm proceeds by making a set of *exploratory moves* and *pattern moves*. The smaller *exploratory moves* identify a promising direction of movement from the current position. Once, this direction has been identified the search takes a larger jump in that direction (*pattern move*) and then explores that new location. This process continues until the *exploratory moves* fail to find a new promising direction. The major steps of the algorithm are sketched below:

Step 1: *Pick an initial base point p.* This is done by choosing the midpoint within the range for each parameter.

Step 2: *Make exploratory moves.* For each parameter p_i we first increment its value by step size s and evaluate the program at $p'(p_1, ...p_i + s, ..., p_n)$. If the execution time at p' is less than the current minimum then we set the value of parameter p_i to $(p_i + s)$ and move on to the next parameter. Otherwise we decrement the value of the parameter by s and evaluate the program at $p'(p_1, ...p_i - s, ..., p_n)$. If $f(p')$ is less than the current minimum then we set the value of parameter p_i to $(p_i - s)$. Otherwise the value of the parameter remains unchanged. Once, all the parameters have been explored we move to Step 3

Step 3: *Make pattern move.* The series of exploratory moves gives us a new point p' in N where we are likely to find a value that is less than the current minimum. The *pattern move* moves the base point in the direction of p', that is $p \leftarrow p' - p$. The execution time at this new point is evaluated. If this execution time is less than the current base point execution time then we go to Step 2. Otherwise we move to Step 4.

Step 4: *Reduce step size.* If we have reached the minimum step size then we move to Step 5. Otherwise, we reduce the step size by the step size reduction factor and go back to Step 2.

Step 5: Done.

5 Experimental Results

5.1 Benchmarks and Search Space

Table 1 lists the benchmarks that were used in our experiments. Among the benchmarks there are four kernels: `advect3d`, `lud`, `mm`, `vpenta` and two full ap-

Table 2. Search Space Properties

Benchmark	Loops Unrolled	Loops Blocked	Search Space Dimension	Points in Search Space
advect3d	1	1	10 x 100	1,000
lud	2	0	30 x 30	900
mm	1	2	200 x 10	2,000
vpenta	1	1	10 x 100	1,000
swim	1	4	10 x 100 x 100	100,000
mgrid	1	2	10 x 100 x 100	100,000

Table 3. Platforms

Processor	L1	L2	L3	Compiler
Alpha 21264A @ 667MHz	8 K	8 M	-	Compaq Fortran V5.5-1877
Itanium2 @ 900 MHz	16 K	256 K	1.5 M	Intel Fortran Compiler 7.1
SGI Origin R10K @ 195 MHz	32 K	1 M	-	MipsPro 7.3,
Pentium 4 @ 2 GHz	8 K	512 K	-	Intel Fortran Compiler 7.1

Table 4. Comparison of Speedup between Direct Search and Exhaustive Search on Itanium 2

Program	**Exhaustive Speedup**	**DS 30 Frac of Best**	**DS 60 Frac of Best**	**DS 90 Frac of Best**	**DS 120 Frac of Best**
advectd3d	1.23	96.75%	96.75%	96.75%	96.75%
lud	4.07	87.48%	100.00%	100.00%	100.00%
mm	1.22	95.90%	99.18%	100.00%	100.00%
vpenta	1.57	86.62%	98.08%	98.08%	98.08%
swim	1.27	96.06%	97.63%	97.63%	97.63%
mgrid	1.17	95.34%	97.89%	97.89%	97.89%

Table 5. Comparison of Tuning Time between Direct Search and Exhaustive Search on Itanium 2

Program	**Exhaustive Time**	**DS 30 Frac of Time**	**DS 60 Frac of Time**	**DS 90 Frac of Time**	**DS 120 Frac of Time**
advectd3d	8:27:11	2.64%	4.99%	9.77%	9.77%
lud	2:28:23	4.10%	6.08%	6.08%	6.08%
mm	4:08:34	1.96%	3.81%	5.71%	6.70%
vpenta	2:31:19	1.69%	3.45%	4.26%	5.02%
swim	664:28:09	0.10%	0.17%	0.29%	0.36%
mgrid	831:23:45	0.06%	0.13%	0.22%	0.22%

Table 6. Performance Improvement and Tuning Time on Itanium2 using Random Search

Program	DS 30		DS 60		DS 90		DS 120	
	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time
advectd3d	1.17	14:28	1.17	26:56	1.17	43:25	1.17	57:53
lud	3.47	5:06	3.91	8:11	4.03	13:17	4.03	16:23
mm	1.08	4:25	1.09	6:51	1.09	9:16	1.09	11:41
vpenta	1.49	1:25	1.55	2:53	1.57	4:20	1.57	5:46
swim	1.13	42:07	1.16	1:24:15	1.16	2:06:23	1.16	2:48:30
mgrid	1.00	43:30	1.02	1:27:00	1.02	2:10:30	1.02	2:54:11
Mean	1.56		1.65		1.67		1.67	

Table 7. Performance Improvement and Tuning Time on Itanium2 using Direct Search

Program	DS 30		DS 60		DS 90		DS 120	
	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time
advectd3d	1.19	13:23	1.19	25:20	1.19	49:34	1.19	49:34
lud	3.52	6:05	4.07	9:01	4.07	9:01	4.07	9:01
mm	1.17	4:52	1.21	9:28	1.22	14:11	1.22	16:39
vpenta	1.36	2:33	1.54	5:13	1.54	6:27	1.54	7:36
swim	1.22	41:07	1.24	1:09:11	1.24	1:56:34	1.24	2:25:02
mgrid	1.12	31:00	1.15	1:04:00	1.15	1:47:50	1.15	1:47:50
Mean	1.60		1.73		1.74		1.74	

Table 8. Performance Improvement and Tuning Time on SGI Origin R10000 using Direct Search

Program	DS 30		DS 60		DS 90		DS 120	
	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time
advect3d	1.00	41:23	1.05	1:21:32	1.05	2:05:02	1.05	4:17:24
lud	2.98	56:00	2.99	1:44:45	2.99	2:05:12	2.99	2:05:12
mm	1.54	19:10	1.58	56:04	1.59	1:22:30	1.59	1:55:45
vpenta	1.61	22:17	1.65	41:23	1.65	1:06:03	1.65	1:20:03
swim	1.04	3:30:19	1.04	5:30:56	1.05	8:12:46	1.05	8:12:46
mgrid	1.04	2:23:00	1.04	3:25:12	1.04	4:45:00	1.04	4:45:00
Mean	1.54		1.56		1.56		1.56	

Table 9. Performance Improvement and Tuning Time on Alpha 21164 using Direct Search

Program	**DS 30**		**DS 60**		**DS 90**		**DS 120**	
	Speedup	**Time**	**Speedup**	**Time**	**Speedup**	**Time**	**Speedup**	**Time**
advect3d	1.58	7:47	1.58	14:44	1.63	19:35	1.63	41:22
lud	1.07	11:02	1.25	17:28	1.25	21:25	1.25	21:25
mm	1.02	6:41	1.02	12:12	1.02	18:01	1.02	38:02
vpenta	1.41	8:28	1.41	16:00	1.42	21:16	1.42	23:29
swim	1.03	1:06:03	1.04	1:59:41	1.04	3:18:50	1.04	3:44:24
mgrid	1.17	58.29	1.17	1:40:21	1.17	2:24:34	1.17	3:01:04
Mean	1.21		1.25		1.26		1.26	

Table 10. Performance Improvement and Tuning Time on Pentium 4 using Direct Search

Program	**DS 30**		**DS 60**		**DS 90**		**DS 120**	
	Speedup	**Time**	**Speedup**	**Time**	**Speedup**	**Time**	**Speedup**	**Time**
advect3d	1.23	3:33	1.27	5:20	1.27	5:37	1.27	5:37
lud	1.52	6:43	1.53	10:33	1.53	10:33	1.53	10:33
mm	6.46	5:05	6.60	5:28	6.65	7:35	6.75	9:35
vpenta	5.75	2:06	5.75	4:00	5.75	14:01	5.75	14:01
swim	1.00	1:10:43	1.00	1:56:03	1.00	2:11:40	1.00	2:11:40
mgrid	1.05	1:15:32	1.05	2:02:07	1.05	2:02:30	1.05	2:02:30
Mean	2.84		2.87		2.88		2.89	

plications: `swim` and `mgrid`. Table 2 lists the applied transformations and the dimensions of the search space for each application. The total number of points in the search space is also listed. Each transformation parameter whose value is being searched by the search algorithm corresponds to a dimension of the search space for that application. As mentioned previously, the possible range of values for each transformation parameter is chosen by hand prior to performing the search. As an example, two loops were unrolled in `lud` and the maximum unroll factor considered for each loop was 30. Hence, for `lud` we have a two-dimensional search space with 900 points. For `swim` four loops were blocked and one loop was unrolled. The four loops that were blocked came from two different loop nests and in searching for the tiling parameters we only considered square tile sizes (i.e. the same tile size was used for loops in the same loopnest). For the unrolled loop the maximum unroll factor considered was 10 whereas the tiling sizes ranged between 1 and 100. Hence, for `swim` we have a three-dimensional search space consisting of 100,000 points.

5.2 Performance Across Architectures

A major argument for doing iterative compilation is its ability to deliver improved performance in an architecture independent manner. Hence, to determine the effectiveness of our approach we ran experiments on four different platforms which are listed in Table 3. On each platform, we first compiled the program with the vendor compiler with full optimization turned on. We ran this program to get the baseline execution time. We then used our direct search technique to search for the optimization parameters for the core loop nests of each program. When using the native compilers in our system we disabled tiling and unrolling in the native compiler whenever possible. This step was necessary since in some cases the native compiler would actually degrade performance when it *re-applied* transformations which had already been applied by our high-level transformer. In each case we allowed the search to continue untill 30, 60, 90 and 120 iterations. Tables 7–10 list performance improvement and tuning time for each platform. For each platform speedup that is reported is the speedup that is obtained over the fully optimized version of the native compiler. The total tuning time includes program evaluations, compilation and search analysis time. The time to do the high level transformations is not included in the total tuning time. Currently, we do not have an implementation of the high-level transformer on all our test platforms. So, for our experiments we generated all the program variants for the search beforehand and during the experiments the call to the source-to-source transformer was replaced by a unix `cp` operation. It should be noted however, that the time for doing high level transformations is not likely to add much to the total tuning time. For any moderate sized program the principal bottleneck is the execution time of the programs.

The results presented in Tables 7–10 show that our approach can yield significant performance improvements across a range of architectures. Overall, the biggest benefits were obtained on the Pentium whereas Alpha provided the least improvements. One explanation, for getting relatively less improvement on the

Alpha is that for the Compaq Fortran Compiler we were not able to selectively disable tiling and unrolling. So, to prevent the native compiler from re-applying tiling and unrolling, we compiled the transformed code with the `-O4` option whereas the baseline version was compiled with full optimizations turned on (`-O5` option). Using the `-O4` option turned off some other high-level transformations such as scalar replacement which may have inhibited the effectiveness of the unroll-and-jam transformation.

5.3 Comparison with Exhaustive Search

In order to evaluate the performance of the direct search strategy we wanted to compare the results against results from an exhaustive search. We ran a set of experiments on the Itanium 2 platform where we used exhaustive search to find the best tiling and unroll factors for the four kernels. Table 4 lists the speedup obtained using exhaustive search. Table 4 also lists speedup obtained from direct search as a percentage of the speedup obtained from exhaustive search. Table 5 lists the tuning time for the two search strategies in a similar fashion. When limiting direct search to 30 iterations about 93% of the performance is gained at about 1.7% of the cost. Increasing the number of iterations gets us closer to the best speedup obtained from exhaustive search. For `lud` and `mm` we are able to find the best solution within the search space after 60 and 90 iterations respectively. The results in Table 4 and 5 show that direct search can come very close to the performance of exhaustive search at only a fraction of the cost.

5.4 Comparsion with Random Search

We also wanted to find out how the direct search strategy compares with a randomized search strategy. Table 6 presents performance results and tuning time for a random search strategy on the Itanium 2 platform. The random search used the same search space as the direct search and at each iteration a random point within the search space was generated and evaluated. The search was allowed to continue until a predetermined number of evaluations. The minimum execution time from all evaluations was compared to the baseline execution time to obtain speedup information.

Comparing the results from Table 6 and Table 7, we see that on average direct search is able to achieve higher speedup than a random search. This improvement is not as significant as we had expected. For 30 evaluations the performance gains from direct search is only about 3.2% better than that of random search. In fact, for `vpenta` random search is able to find a better execution time after both 30 and 60 evaluations. However, direct search does perform significantly better on the two applications that have the largest search spaces. For `swim` the performance improvement is about 7% higher than that of random search whereas for `mgrid` the performance improvement is about 12% higher.

These results indicate that direct search is likely to perform better than random search as the search space gets larger. However, for smaller search spaces random search is almost as effective as direct search. This suggests that there

might be room for improving the direct search strategy for smaller search spaces. One approach of doing this might be to tune the step size parameter to particular search spaces. We plan to explore this issue in future.

5.5 Compilation Costs

The performance improvements that we obtain does come at a cost however. This cost comes in terms of longer tuning times. The total tuning time for the set of benchmarks ranges from a few minutes (kernels) to several hours (full applications). This ofcourse is a natural consequence of any iterative approach. The tuning time is mostly dominated by program execution time. As we can see from the results, increasing the number of iterations results in a proportional increase in tuning time. Also the two applications that have the longest running times suffer the longest tuning time as well.

Another observation to be made from the results is that performance benefits start to diminish rapidly as we run the search algorithm for longer iterations. For all platforms except Itanium2, 98% of the benefits are realized after 30 iterations. Even for Itanium2, going from 30 to 60 iterations yields a modest 10% extra improvement whereas the total tuning time is almost doubled. Interestingly, this behavior holds true for `swim` and `mgrid` whose search space is significantly larger than the search space of the kernels. One lesson in this might be that 30 iterations for the search algorithm is perhaps too many. More experiments will be needed if we wish to discover the right number of iterations for a given program. However, even in the long run we expect that in an iterative approach there will always be a trade-off between the performance gains and tuning time.

5.6 Summary

The results in this section show that direct search strategy is able to find suitable tile sizes and unroll factors by exploring only a small fraction of the search space. The results also support the notion that an iterative approach is able to deliver performance across architectures at the cost of extra compilation time.

There are several issues that still remain open. For our experiments the selection of loops and the generation of the search space was done by hand. The choice of loops to unroll or tile and the initial search space used by the search strategy has a strong impact on how well the search performs. Finding suitable ways to generate the search space automatically will require further research. Another issue that needs to be explored is the cost of tuning time. If the iterative approach is to be employed in a production environment we need to find out the level of performance it needs to deliver so that the compilation cost is amortized over many runs of the program.

6 Conclusions and Future Work

Given the enormous complexity of today's processor architecture, building exact architectural models have become an intractable task. As such, most optimizing

compilers use simplified models that have obvious shortcomings. An empirical method can overcome some of these shortcomings by searching for the best transformation parameters through iterative evaluation of the program. However, to do an effective job such a system requires a powerful search technique that can explore the complex search space to find good parameters values in reasonable time.

Our experimental results in Section 5 show that direct search can be an effective strategy for exploring the transformation parameter space. Its ability to discover good solutions in relatively few iterations make direct search a good choice for an iterative compilation system.

Although direct search proves to be an effective strategy in exploring the search space, there are several issues that still need to be addressed in applying this technique in an empirical tuning framework. In future, we plan to improve our strategy by using static models and architectural information to generate and prune the search space. A smaller and more representative search space is likely to improve the search results. The issue of long tuning times also needs to be addressed. We plan to explore ways of using training data sets during the tuning process to cut down the program execution time. Finally, the search itself might be made more effective by tailoring the step size parameter to specific search spaces.

References

1. J. Bilmes, K. Asanovic, C.-W. Chen, and J. Demmel. Optimizing matrix multiply using phipac: a portable high- performance ansi-c coding methodology. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.
2. S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.
3. K. Cooper, D. Subramanian, and L. Torczon. Adapative optimizing compilers for the 21st century. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001.
4. M. Frigo. A fast fourier transform compiler. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
5. G.G.Fursin, M.F.P.O'Boyle, and P.M.W.Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Fifteenth International Workshop on Languages and Compilers for Parallel Computing*, College Park, Maryland, July 2002.
6. R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. In *Journal of the ACM*, pages 212–229, 1961.
7. T. Kisuki and P. Knijnenburg. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.
8. P. Knijnenburg, T. Kisuki, and M. O. Boyle. Iterative compilation. In *Embedded Processor Design Challenges - System Architecture, Modeling and Simulation (SAMOS)*, Lecture Notes in Computer Science 2268, pages 171–187. Springer-Verlag, May 2002.

9. P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, Washington, DC, June 2004.
10. R. M. Lewis, V. Torczon, and M. W. Trosset. Direct search methods: then and now. *Journal of Computational and Applied Mathematics*, 124(1-2):191–207, Dec. 2000.
11. J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 2002. In press. *Special Issue with selected papers from the Los Alamos Computer Science Institute Symposium.*
12. A. Qasem, G. Jin, and J. Mellor-Crummey. Improving performance with integrated program transformations. Technical Report CS-TR03-419, Dept. of Computer Science, Rice University, Oct. 2003.
13. V. Torczon. *Multi-Directional Search: A Direct Search Algorithm for Parallel Machines*. PhD thesis, Dept. of Computer Science, Rice University, May 1989.
14. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, San Fransisco, CA, 2003.
15. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98: High Performance Networking and Computing*, Orlando, FL, Nov. 1998.
16. M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on MicroArchitecture*, pages 274–286, 1996.
17. M. J. Wolfe. Iteration space tiling for memory hierarchies, Dec. 1987. Extended version of a paper which appeared in *Proceedings of the Third SIAM Conference on Parallel Processing*.
18. J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP algorithms. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.
19. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.