Tuning High Performance Kernels through Empirical Compilation *

R. Clint Whaley
Dept. of Computer Science
Florida State University
whaley@cs.fsu.edu

David B. Whalley
Dept. of Computer Science
Florida State University
whalley@cs.fsu.edu

Abstract

There are a few application areas which remain almost untouched by the historical and continuing advancement of compilation research. For the extremes of optimization required for high performance computing on one end, and embedded systems at the opposite end of the spectrum, many critical routines are still hand-tuned, often directly in assembly. At the same time, architecture implementations are performing an increasing number of compiler-like transformations in hardware, making it harder to predict the performance impact of a given series of optimizations applied at the ISA level. These issues, together with the rate of hardware evolution dictated by Moore's Law, make it almost impossible to keep key kernels running at peak efficiency. Automated empirical systems, where direct timings are used to guide optimization, have provided the most successful response to these challenges. This paper describes our approach to performing empirical optimization, which utilizes a low-level iterative compilation framework specialized for optimizing high performance computing kernels. We present results showing that this approach can not only provide speedups over traditional optimizing compilers, but can improve overall performance when compared to the best hand-tuned kernels selected by the empirical search of our well-known ATLAS package.

1 Introduction and Related Work

For high performance computing on one end, and embedded computing at the opposite extreme, many critical routines are still hand-tuned (often directly in assembly) in order to achieve the required efficiency. It is very rare indeed that the hand-tuner in question applies a technique that is unknown to the compilation community. In almost every case, this hypothetical hand-tuner is applying transformations that are well understood, but is either optimizing for an architecture upon which compilers are not yet well-tuned (and may never be well-tuned), or is applying the techniques in ways that compilers either cannot (due to imperfect analysis) or purposely avoid (for instance, because the technique causes a slowdown in many cases).

This problem is exacerbated in high performance computing, where compute needs mandate usage of parallel machines. If the serial kernels upon which the computation is based fail to achieve adequate efficiency, this inefficiency results in the need to utilize more processors to solve the problem, which in turn leads to greater bottlenecks due to increased parallelization costs (eg., greater communication and implementation time). Therefore, any serial weakness is greatly magnified when parallelization is considered, and thus it becomes even more critical that the computational kernels used by the HPC application achieve the greatest percentage of peak possible.

At the same time, Moore's law has ensured that not only does new hardware come out in release cycles too short for compiler writers to keep pace, it has also resulted in architectures that implement many compiler-like transformations in hardware (eg., dynamic scheduling, out-of-order execution, register renaming, etc.). Due to this trend, the ISA available to the compiler writer becomes more and more like a high-level language, and thus the close connection between the instructions issued by the compiler, and the actions performed by the machine, is lost. This phenomenon makes it increasingly difficult to know a priori if a given transformation will be helpful, and almost impossible to be sure when it is worth applying a transformation that yields benefits only in certain situations. The most extreme example of this is embodied in the x86 architecture, whose non-orthogonal CISC instruction set has, to the frustration of many compiler writers, become the most widely-used ISA in general-purpose computing.

Even when the ISA is kept relatively close to the actual hardware, building models for each machine that instantiate all of the characteristics that affect optimization at this level is almost intractable. Even if a model could be created that was sophisticated enough to account for the in-

*This research was supported in part by National Science Foundation grants CIA-0072043, CCR-0208892, and CCR-0812493.

teractions between all levels of cache, the pipelines of all relevant functional units, and all shared hardware resources required by a given operation, it is often the case that much of the required data is proprietary, or due to unforeseen resource interactions, unknown even to the hardware vendor.

These problems, taken together, have led to the implementation of empirically tuned packages, such as ATLAS [17, 16] and FFTW [5, 10]. The central idea behind these packages is that since it is difficult to predict a priori whether or by how much a given technique will improve performance, one should try a battery of known techniques on each performance-critical kernel, obtain accurate timings to assess the effect of each transformation of interest, and retain only those that result in measurable improvements for this *exact* system and kernel. This approach allows for a much greater degree of specialization than can be realistically achieved in any other fashion. For instance, it is not uncommon for empirical tuning of a given kernel on two basically identical systems, varying only in the type or size of cache supported, to produce tuned implementations with significantly different optimizational parameters, and it is almost always the case that varying the kernel results in widespread optimization differences.

These empirically tuned packages have succeeded in achieving high levels of performance on widely varying hardware, but in a sense they are still very limited compared to compilation technology. In particular, they are tied to particular operations within given libraries, and are therefore not of great assistance in optimizing other operations that nonetheless require similar levels of performance. It is therefore no surprise that the compiler community has begun to evaluate the scope for using empirical techniques in compilation. Empirical techniques have greater overhead than more conventional feedback-generating schemes such as profiling, and thus are poorly suited for general compilation (unless a more targeted and less empirical approach is used, as in [12]). Therefore, most research in empirical compilation concentrates on one of these narrow areas to which we previously alluded. Sometimes this leads to library-oriented compiler research, as in the well-known SPIRAL work [9, 11], and sometimes it is more compiler-centric, but strongly oriented to an area such as embedded systems [7, 13] or high performance computing [4, 2]. These efforts, which should help automate building the libraries that drive high performance computing, are thus tools that can be leveraged in even higher-level compilation efforts, such as [6].

1.1 Our Contribution

The long-range goal of our research is to eliminate the need for hand-tuning in our area of interest, high performance computing. Our approach to this research is to utilize an empirical and iterative compilation framework to first enhance, and then (in some cases) replace, the kernel-specific techniques presently used in ATLAS. This work addresses two key weaknesses in the approach we employed in developing ATLAS: (1) ATLAS's optimization are kernel specific, and (2) Oftentimes the native compiler prevents ATLAS from achieving full performance in a portable manner. Many of the aforementioned researchers are performing related research using other packages (eg., SPIRAL, FFTW, etc.), but our approach is the first of which we are aware to perform all transformations at a low level in the backend (many researchers instead generate code in high level languages, just as ATLAS does), and at the same time actually have the search as part of the compiler (many projects put the search in the library generator). Working at the low level allows us to exploit the extremely architecture-specific features (eg., SIMD vectorization, CISC instruction formats, pipeline resource limitations, etc.) that are required to squeeze the last few percent of performance from an ISA, while in keeping the search in the compiler, we hope to generalize it enough to tune almost any floating point kernel.

One of the key contributions of this research is in where we focus the work, which is guided by our experience in developing ATLAS. So many papers have discussed search techniques that many researchers have come to believe that fast searches are the primary barrier to solving the problems inherent in this level of tuning. Our own ATLAS work directly contradicts this, in that it still uses the simplest of search techniques, and yet is one of only a handful of such efforts that is both freely available as software and also produces reasonable results on real-world architectures. It is our experience that a simple but intelligently designed search (as described in Section 2.3) reduces the problem of search to a low order term, and thus it does not make sense to make it the focus of the work.

In this paper we discuss our iterative and empirical compilation framework, iFKO [15] (iterative Floating Point Kernel Optimizer). We provide an overview of how it operates, and the set of transformations it presently supports. In order to show that empirical tuning can be effective on even the most well-understood and easily analyzed operations, we first concentrate on the simplest kernels ATLAS provides, the Level 1 BLAS [8] (see Section 3.1 for details). One of the surprising contributions of this paper is that even on simple loops such as these, empirical application of even a modest set of well-known compiler transformations can lead to performance improvements greater than those supplied by the best native compiler. Due to its ubiquity and the extreme separation between its ISA and its actual hardware (which makes empirical techniques uniquely attractive), we have chosen the x86 as our initial architectural target. We show that even in this early stage of development, iFKO produces code on average better than the hand-tuned codes chosen by ATLAS's empirical search (we are not aware of any other group who has shown this to be the case on actual kernels on real-world machines, using a completely automated framework) ; in those few cases where iFKO fails to provide the fastest implementation, we describe the transformation that the hand-tuned code utilized to get the best kernel, so it is clear whether or not the technique can be eventually be generalized into our compilation framework.

2 iFKO Implementation Details

If the supported transforms fail to supply performance equivalent to that gained by hand-tuning, our target community will probably not employ an automated approach. Therefore, we must make the effect of each transformation, and the interaction between transformations, as optimal as possible, and so it is better to do a limited number of transformations very well than to support many transformations that do not fully realize their potential. With this narrow and deep focus in mind, Section 2.1 provides a high-level overview of the entire framework, Section 2.2 describes the optimizing compiler component, while Section 2.3 outlines the current search implementation.

2.1 Anatomy of an Iterative and Empirical Compiler

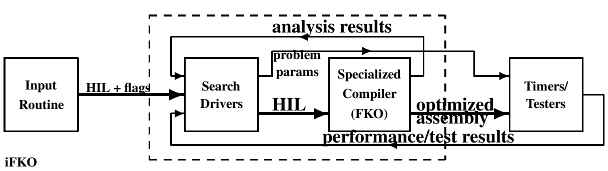


Figure 1. Overview of our Empirical and Iterative Compilation System

Figure 1 shows the basic outline of our empirical and iterative compilation system. Just as in a traditional compiler, iFKO is provided with a routine to be compiled, and perhaps some user-selected compiler flags. iFKO is composed of two components: (1) a collection of search drivers and (2) the compiler specialized for iterative empirical floating point kernel optimization (FKO).

The search first passes the input kernel to be optimized to FKO for analysis. FKO then provides feedback to the master search based on this analysis. The analysis phase together with any user input essentially establishes the optimization space to be searched, and the iterative tuning is then initiated. For each optimization of interest that takes an empirically tuned parameter (eg., the unrolling factor in loop unrolling), the search invokes FKO to perform the transformation, the timer to determine its effect on performance, and the tester to ensure that the answer is correct (unnecessary in theory, but useful in practice).

Input can be provided both by mark-up in the routine itself, and by flag selection from the user. These inputs can be used to place limits on the search, as well as to provide information specialized for an individual usage pattern (such as whether the operands are pre-loaded in cache, the size of the problem to time, etc.). Note that iFKO has intelligent defaults for these values, so such user direction is optional. The 'HIL' in Figure 1 stands for high-level intermediate language, and is the language (specialized for floating point kernel optimization) which FKO accepts as input.

2.2 Floating Point Kernel Optimizer (FKO)

The heart of this project is an optimizing compiler called FKO, which has been specialized for empirical optimization of floating point kernels. Our focus on these kernels affects not only our choice of optimizations, but also our input language. Unlike a traditional compiler, FKO must also communicate its analysis to the search program, as described in Section 2.2.2.

In general, one of the main strengths of empirical optimization is that all known techniques can by tried, even ones that may cause significant slowdown in some cases, since the search can keep only the successful optimizations. Therefore FKO eventually needs to have an incredibly diverse battery of transformations, which will result in an expanding optimization space, which in turn will require sophisticated search techniques. Compared with those available to an experienced hand-tuner (and thus to the set of techniques we would ultimately like to implement) FKO presently has a very tractable number of optimizations, which are split into two types. FKO has *fundamental* transformations, which are applied only one time and in a known order (thus easing the extra analysis required for some of these optimizations), and these techniques are outlined in Section 2.2.3. The second class consist of the *repeatable* transformations, discussed in Section 2.2.4, which may be applied multiple times and in almost any order. In order to present an overview of the entire work, we provide only a brief outline of each transformation here, due to space limitations.

2.2.1 Input Language (HIL)

Our input language is kept close to ANSI C in form, so that the task of kernel implementation is comparable to writing a reference implementation in languages such as ANSI C or Fortran 77 (common kernel languages). However, we wanted to keep it simple enough so that we can concentrate on back-end optimization, as well as to specialize it to some degree for our problem domain. Therefore, we provide an opportunity for user mark-up that can provide information that is normally discovered (if it can be determined at all) by extensive front-end analysis. For the simple operations surveyed in this paper, the only mark-up used was the identification of the loop upon which to base the iterative search (iFKO could optimize all inner loops this way, but this could potentially cause insupportable slowdown in tuning more complex kernels, and so we require that a loop be flagged as important before it is empirically tuned).

Although our input language resembles ANSI C, its usage rules are closer to Fortran 77, which has a more performance-centric design. For instance, aliasing of output arrays is disallowed unless annotated by mark-up. A detailed description of the input language is beyond the scope of this paper, but Section 3.2.1 shows example loops written in our HIL (high-level intermediate language) that correspond to the ANSI C loops surveyed in Section 3.1.

2.2.2 Analysis and Communication with the Search

Unlike a normal compiler, a compiler used in an iterative search needs to be able to communicate key aspects of its analysis of the code being optimized, as this strongly affects the optimization space to be searched. Currently, FKO reports architecture information such as the numbers of available cache levels and their line sizes. More importantly, it reports kernel-specific information such as the loop (if any) identified for tuning in the iterative search. For this loop, it then reports the maximum safe unrolling, and whether it can be SIMD vectorized. For each scalar and array accessed in the loop, the analysis further reports its type, sets and uses. Finally, the analysis returns a list of all scalars that are valid targets for accumulator expansion (see Section 2.2.3), and all arrays that are valid targets for prefetch (by default any array whose references increment with the loop, but the user can override this behavior, for instance for arrays known to be already in cache, using mark-up).

2.2.3 Fundamental Transformations

This section outlines the fundamental optimizations presently supported by FKO, in the order in which they are applied. For each such transformation, we list an abbreviation which is used in the paper to refer to this optimization. *SIMD Vectorization* [3] (**SV**): transforms the loop nest (when legal) from scalar instructions to vector instructions. This typically results in the same number of instructions in the loop, but its effect on loop control and computation done per iteration is similar to unrolling by the vector length (4 for single precision, 2 for double). *Loop Unrolling* [1] (**UR**): duplicates the loop body (avoiding repetitive index and pointer updates) N_u times. Since it is performed after SIMD vectorization, when vectorization is also applied the computational unrolling is actually $N_u * veclen$. *Optimize Loop Control* (**LC**): rearranges loop indexing (when possible) to avoid (on some architectures) unnecessary loop branch comparisons, or to exploit such features as specialized counter registers. *Accumulator Expansion* (**AE**): In order to avoid unnecessary pipeline stalls, **AE** uses a specialized version of scalar expansion [1] to break dependencies in scalars that are exclusively the targets of floating point adds within the loop.

The next fundamental transformation is *prefetch* (**PF**). This transformation can prefetch any/all/none of the arrays that are accessed within the loop, select the type of prefetch instruction to employ, vary the distance from the current iteration to fetch ahead, as well as provide various simple scheduling methodologies. Prefetches are scheduled within the unrolled loop because many architectures discard prefetches when they are issued while the bus is busy, and so they can be an exception to the general rule that modern x86 architectures are relatively insensitive to scheduling (due to their aggressive use of dynamic scheduling, out-of-order execution, register renaming, etc.). Note that prefetching one array can require multiple prefetch requests in the unrolled loop body, as each x86 prefetch instruction fetches only one cache line of data.

Our final fundamental transformation is *non-temporal writes* (**WNT**), which employs non-temporal writes on the specified output array. These are writes that contain a hint to the caching system that they should not be retained in the cache, though how this hint is used varies strongly by architecture.

2.2.4 Repeatable Transformations

Repeatable transformations can not only be applied multiple times, but are typically applied in a series (or optimization block) which is repeated while they are still successfully transforming the code. This is useful for synergistic optimizations (eg., register allocation and copy propagation). All of these operations may be applied to a scope (a set of basic blocks, typically a given loop nest or the entire function). When applied to each loop nest level in turn, scoping ensures that inner-loop resource needs are fully satisfied before outer loops are considered, which is critical in floating point code, where long-running loops are the typical case. Most of FKO's repeatable transformations are fairly standard (though in some we have added a few refinements to specialize them for floating point optimization), and are discussed in [1].

We support repeatable transformations for improving register usage and control flow. In register usage optimization, we support two types of register allocation and several forms of copy propagation. We also perform several peephole optimizations that exploit the fact that the x86 is not a true load/store architecture (relatively important when the

ISA has only eight registers, but the underlying hardware may have more than a hundred). Finally, we perform branch chaining, useless jump elimination, and useless label elimination, which, when applied together, merges basic blocks (critical after extensive loop unrolling).

2.3 Iterative Search

Finding the best values for N_T empirically tuned transformations consists of finding the points in an N_T dimensional space that maximize performance (thus the phrase "searching the optimization space"). There are several ways of performing this search, including simulated annealing and genetic algorithms. We currently use a much simpler technique, a modified line search. In a pure line search, the N_T-D problem is split into N_T separate 1-D searches, where the starting points in the space correspond to the initial search parameter selection (in our case, FKO defaults). Obviously, this approach results in a very poor search of the space by volume. However, since we understand the properties of these transformations, we are able to select reasonable start values for the search, and because we understand many of the interactions between optimizations, we are able to relax the strict 1-D searches to account for interdependencies (eg., when two transformations are known to strongly interact, do a restricted 2-D search). Application of this knowledge to the line search algorithm provides what is in essence a de facto expert system / search hybrid. In this sense, much of the high-level knowledge that influences model-driven compilation may be moved into the search (where heuristics and architectural assumptions are replaced with empirical probes, of course), rather than being lost as when the search is based purely on geometry. With these straightforward modifications, line searches are quite effective in practice (ATLAS, one of the most successful empirical projects, still uses a modified line search), even though they are completely inadequate in theory.

The present iterative search varies only the fundamental optimizations. Our search takes FKO's optimization defaults for its initial values. If we define L as the line size of the first prefetchable cache, and L_e as the number of elements of a given type in such a line (for example, if $L = 32$ bytes, L_e would be 4 for a double precision scalar, 8 for a single precision scalar, or 2 for a SIMD vector of either type), then the initial values for the search (and thus the defaults for FKO) are: **SV**='Yes', **WNT**='No', **PF**(type,dist) = ('prefetchnta', $2 \times L$), **UR**=L_e, **AE**='No'.

3 Experimental Results

This section presents and analyzes results on two of today's premier x86 implementations, and is organized in the following way: Section 3.1 outlines the floating point kernels that are being optimized, Section 3.2 discusses version and timing methodology information, and Section 3.3 discusses some key points about the presented results. When we have had to concentrate on one machine due to space limitations, we have retained the the timings for the Intel machine, as it supplies the fairest comparison against Intel's compiler, even though it is not our best platform. For instance, for the omitted in-L2 Opteron timings, the two best tuning mechanisms are iFKO followed by FKO, and icc-tuned kernels run on average at 68% of the speed of iFKO-tuned code.

3.1 Surveyed Routines

The general domain of this research is floating point kernels, but this paper focuses on the Level 1 BLAS. The Level 1 BLAS are vector-vector operations, most of which can be expressed in a single for-loop. These operations are so simple that it would seem unlikely that empirical optimization could offer much benefit over model-based compilation. One of the key contributions of this initial work is that we show that even on such well-understood and often-studied operations as these, empirical optimization can improve performance over standard optimizing compilers.

Most Level 1 BLAS have four different variants depending on type and precision of operands. There are two main types of interest, real and complex numbers, each of which has double and single precision. In this work, we concentrate on single and double precision real numbers. The Level 1 BLAS all operate on vectors, which can be contiguous or strided. Again, we focus on the most commonly used (and optimizable) case first, the contiguous vectors. For each routine, the BLAS API prefixes the routine name with a type/precision character, '`s`' meaning single precision real, and '`d`' for double precision real. Since `iamax` involves returning the index of the absolute value maximum in the vector, the API puts the precision prefix in this routine as the second character rather than the first (i.e., `isamax` or `idamax` rather than `ddot` or `sdot`). There are quite a few Level 1 BLAS, and so we study only the most commonly used of these routines, which are summarized in Table 1.

Note that the performance of the BLAS are usually reported in MFLOPS (millions of floating point operations per second), but that some of these routines actually do no floating point computation (eg., `copy`). Therefore, the FLOPs column gives the value we use in computing each routine's MFLOP rate.

3.2 Methodology and Version Information

All timings were done with ATLAS version 3.7.8, which we modified to enable vectorization by Intel's C compiler, icc. Most of the loops in ATLAS are written as '`for(i=N; i; i--)`' or '`for(i=0; i!=N; i++)`' and icc will not vectorize either form, regardless of what is in the loop. Once we experimentally determined that this loop formulation was preventing icc from vectorizing any of the target loops, we simply modified the source of the relevant routines to '`for(i=0; i<N; i++)`', which icc successfully vectorizes.

We report numbers for two very different high-end x86 architectures, the Intel Pentium4E and AMD Opteron. Further platform, compiler and flag information is summarized in Table 2 (for the profile build and use phases, the appropriate flags were suffixed to those shown Table 2.) The ATLAS Level 1 BLAS kernel timers were utilized to generate all performance results. However, we enabled ATLAS's assembly-coded walltimer that accesses hardware performance counters in order to get cycle-accurate results. Since walltime is prone to outside interference, each timing was repeated six times (on an unloaded machine), and the minimum was taken. Because these are actual timings (as opposed to simulations), there is still some fluctuation in performance numbers despite these precautions, and so we additionally ran each install three times and chose the best.

3.2.1 Input Routines

With the exception of `iamax`, the computational loops of the ANSI C reference implementations are precisely those given in Table 1. The input routines given to FKO were the direct translations of these routines from ANSI C to our HIL (i.e., high level optimizations were not applied to the source). For instance, Figure 6(a) shows the translation of the `dot` routine into our HIL. The exception to this strictly corresponding mapping is the `iamax` routine. Our HIL does not yet support scoped ifs, and so it was originally coded for all compilers (in the appropriate language) as shown in 6(b), which, absent code positioning transformations, is the most efficient way to implement the operation. However, this formulation of `iamax` depressed performance significantly for icc, while not noticeably improving gcc's performance, and so we utilized the more straightforward implementation for these compilers.

3.3 Analysis

Figures (2,3,4) report the percentage of the best observed performance provided by the following six methodologies: **gcc+ref**: Performance of ANSI C reference implementation compiled by gcc. **icc+ref**: Performance of ANSI C reference implementation compiled by icc. **icc+prof**: Performance of ANSI C reference implementation, using icc and profiling. Profiling was performed with tuning data identical to the data used in timing. **ATLAS**: The best kernel found by ATLAS's empirical search, installed with both icc and gcc. ATLAS empirically searches a series of implementations, which were laboriously written and hand-tuned using mixtures of assembly and ANSI C, and contain a multitude of both high and low-level optimizations (eg., software pipelining, prefetch, unrolling, scheduling, etc.). When ATLAS has selected a hand-tuned all-assembly kernel (as opposed to the more common ANSI C routine with some inline assembly for performing prefetch), the routine name is suffixed by a * (eg., `dcopy` becomes `dcopy*`). This is mainly of interest in that hand-tuning in assembly allows for more complete and lower-level optimization (eg. SIMD vectorization, exploitation of CISC ISA features, etc.). **FKO**: The performance of the kernel when compiled with FKO using default transformation parameters (i.e., no empirical search). **iFKO**: The performance of the kernel when iterative compilation is used to tune FKO's transformation parameters.

For each kernel, we find the mechanism that gave the best kernel performance, and all other results are divided by that number (eg. the method that resulted in the fastest kernel will be at 100%). This allows for the relative benefit of the various tuning mechanisms to be evaluated. This comparison is done for each studied kernel, and we add two summary columns. The second-to-last column (AVG) gives the average over all studied routines, and the last column (VAVG) gives the average for the operations where SIMD vectorization was successfully supplied; in practice, this means the average of all routines excluding `iamax`, which neither icc nor iFKO automatically vectorize.

On all studied architectures and contexts, iFKO provides the best performance on average, better even than the hand-tuned kernels found by ATLAS's own empirical search. However, in several individual hand-tuned cases, iFKO loses decidedly. Primarily, this occurs in `iamax`, where the hand-tuned assembly vectorizes the loop, but neither iFKO nor icc can do so automatically. It is a topic for further research to see if we can find a way to vectorize such loops generally and safely in the compiler. The only other routine where iFKO is significantly slower is in P4E/`dcopy`, where the hand-tuned assembly uses a technique called block fetch [14]. This technique can be performed generally and safely in a compiler, and we are planning to add it to FKO.

iFKO is slightly slower (just barely above clock resolution) on out-of-cache Opteron `scopy`, and this is due to FKO generating an extra integer operation per loop iteration. FKO presently does not exploit the opportunity to use x86 CISC indexing to index both arrays using a register, which avoids an additional pointer increment at the end of the loop. The summary here is that, given a few optimizations that we understand and plan to add, we would lose only on `iamax`, and further research is required to determine if we can find a general way to vectorize this operation as well (it seems almost certain that we can overcome this problem in a narrow way, for instance by having the user supply us with markup indicating how to address the dependency).

Table 3 shows the best parameters that were found by our empirical search for each platform/context. Section 2.2.3 defines the abbreviations used in the headings, and Section 2.3 provides the default values used by FKO. The prefetch parameters have both instruction type (INS) and distance in bytes (DST). For each type of prefetch instruction, the search chooses between those available on the machine, and they are reported using the following abbreviations: `tX`: SSE temporal prefetch to cache of level $X + 1$ (eg., `prefetcht0`, `prefetcht1`, etc.); `nta`: SSE non-temporal prefetch to the level of supported cache nearest the CPU (`prefetchnta`), or `w`: 3DNow! prefetch for write (`prefetchw`). Figure 7 shows, as a percentage of FKO's speed, the results of empirically tuning these parameters (i.e. the speedup of iFKO over FKO, *not over code in which a given transformation has not been applied*). For each BLAS kernel, we show a bar for each architecture (p4e/opt) and context (ic: in-L2 cache, oc: out of cache). Each bar shows the total speedup over FKO, and how much tuning each transformation parameter contributed to that speedup. For instance, on average over all operations, architectures and contexts, empirically tuning [WNT, PF DST, PF INS, UR, AE], provided speedups of [2, 26, 3, 2, 5]%, respectively, resulting in the empirically-tuned kernels on average running 1.38 times faster than our statically-tuned kernels. The prefetch results are of particular interest, in that they are relatively difficult to to model accurately, and provide the greatest speedup on average.

One of the most important observations from these tables is how variable these parameters are: they vary depending on operation, precision, architecture, *and* context. This suggests that any model that captures this complexity is going to have to be very sensitive indeed. Note that while empirical results such as these can be used to refine our understanding of relatively opaque interactions (eg., competing compiler and hardware transformations), which in turn allows for building better theoretical models, one of the great strengths of empirical tuning is that full understanding of why a given series of transformations yielded good speedup is not required in order to achieve that speedup.

In addition to the general variability, we can examine how parameters can change based on either the architecture or context of the operation. When we vary the architecture, of course, empirical methods shine, particularly when the compiler has not yet been (or will never be) fully tuned to the new platform (eg. Intel compiler on AMD platform). Here we see the strength of empirical tuning over even aggressive profiling: notice that for both `swap` and `axpy`, **icc+prof** is many times slower than than **icc+ref** in Figure 3. This is because non-temporal writes (**WNT**) can improve performance anytime the operand doesn't need to be retained in the cache on the P4E. On the AMD Opteron, however, non-temporal writes result in significant overhead unless the operand is write only (as in the Y vector of `copy`). Icc's profiling detects that the loop is long enough for cache retention not to be an issue, and blindly applies **WNT**, whereas the empirical tuning tries it, sees the slowdown, and therefore does not use it.

In addition to adapting to the architecture, empirical methods can be utilized to tune a kernel to the particular context in which it is being used. Figure 4 and Table 3 show such an example, where the adaptation is to having the operands in-L2-cache. This changes the optimization set fairly widely, including making prefetch much less important, and **WNT** a bad idea. Prefetch is still useful in keeping data in-cache in the face of conflicts, and so we see it provides greater benefit for the "noisier" (bus-wise) routines such as `swap`. In-cache, computational optimizations become much more important. One such is transformation is accumulator expansion (**AE**), which on the P4E accounts for an impressive 41% of `sasum` speedup in-cache, while only improving performance by 2% for out-of-cache.

Since all results discussed so far are relative to the best tuning method, it is easy to lose track of the relative performance of the individual operations. Therefore, Figure 5(a) shows the the speed of these operations in MFLOPS, computed as discussed in Section 3.1. Note MFLOPS is a measure of speed, so larger numbers indicate better performance. All timings in this figure deal only with iFKO (on average, the best optimizing technique). Basically, the more bus-bound an operation is, the worse the performance. For example, ASUM, which has only one input vector, and no output vectors, is always the fastest routine, with single precision (half the data load for same amount of FLOPs) always faster than double precision. Similarly, Figure 5(b) shows the speedup of the in-L2 cache P4E timings over the out-of-cache performance. One of the most interesting things about this graph is that it provides a very good measure of how bus-bound an operation is, even after prefetch is applied: If the kernel tuned for in-cache usage is only moderately faster than the kernel when tuned in out-of-cache timing, the main performance bottleneck is clearly not memory.

An interesting trend to notice in surveying these results in their entirety is that the more bus-bound an operation is, the less prefetch improves performance. The reason for this seeming paradox is in how prefetch works: prefetch is a latency-hiding technique that allows data to be fetched for later use while doing unrelated computation. If the bus is always busy serving computation requests, there is no time when the prefetch can be scheduled that doesn't interfere with an active read or write, and most architectures simply ignore prefetch instructions in this case. This is why operations such as `swap` or `axpy` get relatively modest benefit. Since prefetch optimization is one of our key strengths for out-of-cache usage on these routines, this is also why iFKO does much better on the Opteron than on the P4E (when compared against all tuning mechanisms, including icc): the Opteron, having a slower chip and faster memory access, is less bus bound, and so there is more room for empirical improvement using this key optimization.

4 Summary and Conclusions

We have shown how empirical optimization can help adapt to changes in operation, architecture, and context. We have discussed our approach to empirical compilation, and presented the framework we have developed. We have shown that even on simple, easily analyzed loops that many would expect to be fully optimized by existing compilers, empirical application of well-understood transformations provides clear performance improvements. Further, even though our current palette of optimizations is limited compared to that available to the hand-tuner, we have presented results showing that this more fully automated approach results in greater average performance improvement than that provided by ATLAS's hand-tuned (and empirically selected) Level 1 BLAS support. Note that our initial timings show iFKO already capable of improving even Level 3 BLAS performance more than icc or gcc, but due to the lack of outer-loop specialized transformations (which we plan to add) we are presently not competitive with the best Level 3 hand-tuned kernels. Therefore, as this framework matures, we strongly believe that it will serve to generalize empirical optimization of floating point kernels, and that it will vastly reduce the amount of hand-tuning that is required for high performance computing. Finally, it appears certain that an open source version of such a framework will be a key enabler of further research as well. For example, just as ATLAS was used to provide feedback into model-based approaches [18], iFKO will provide an ideal platform for tuning and further understanding the models used in traditional compilation, while a fully-featured FKO will provide a rich test bed for research on fast searches of optimization spaces.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[2] P. Diniz, Y.-J. Lee, M. Hall, and R. Lucas. A case study using empirical optimization for a large, engineering application. In *International Parallel and Distributed Processing Symposium*, 2004. CD-ROM Proceedings.

[3] F. Franchetti, S. Kral, J. Lorenz, and C. Ueberhuber. Efficient utilization of simd extensions. Accepted for putblication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

[4] M. Frigo. A Fast Fourier Transform Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99), Atlanta, GA*, 1999.

[5] 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.

[6] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. Accepted for publication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

[7] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *CPC2000*, pages 35–44, 2000.

[8] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[9] M. Pushel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Frenchetti, A. Cacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. Accepted for putblication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

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

[11] See page for details. SPIRAL homepage. `http://www.spiral.net/`.

[12] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization*, pages 204–215, 2003.

[13] P. van der Mark. Iterative compilation. Master's thesis, Leiden Institute of Advanced Computer Science, 1999.

[14] M. Wall. Using Block Prefetch for Optimized Memory Performance. Technical report, Advanced Micro Devices, 2002.

[15] R. C. Whaley. *Automated Empirical Optimization of High Performance Floating Point Kernels*. PhD thesis, Florida State University, December 2004.

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

[17] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (cmt-twww.netlib.org/lapack/lawns/lawn147.ps).

[18] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. A comparison of empirical and model-driven optimization. Accepted for publication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

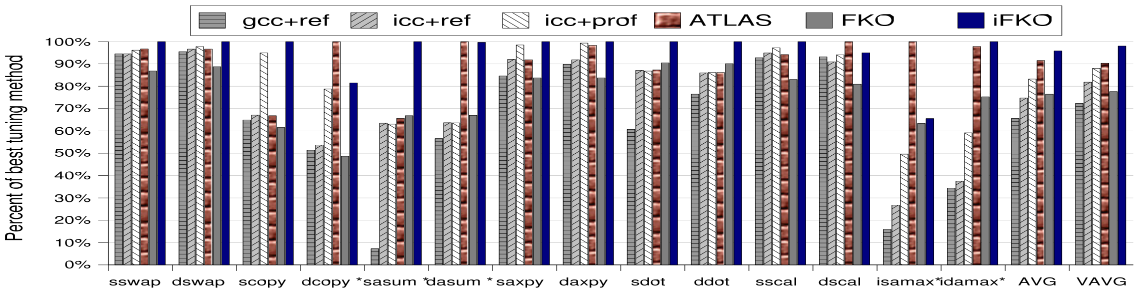


Figure 2. Relative speedups of various tuning methods on 2.8Ghz P4E, N=80000, out-of-cache

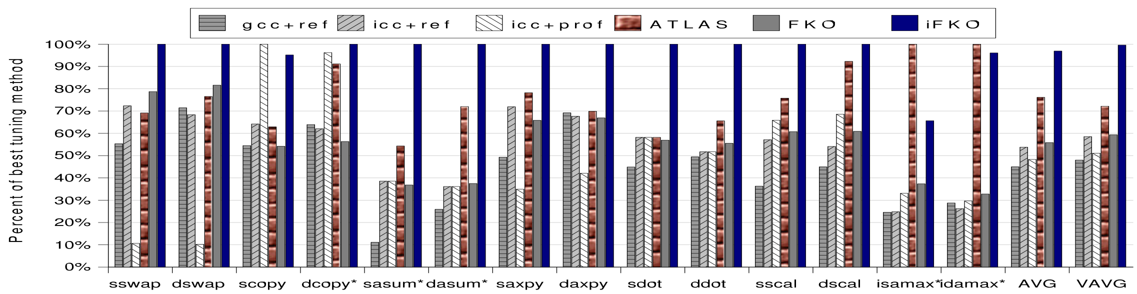


Figure 3. Relative speedups of various tuning methods on 1.6Ghz Opteron, N=80000, out-of-cache

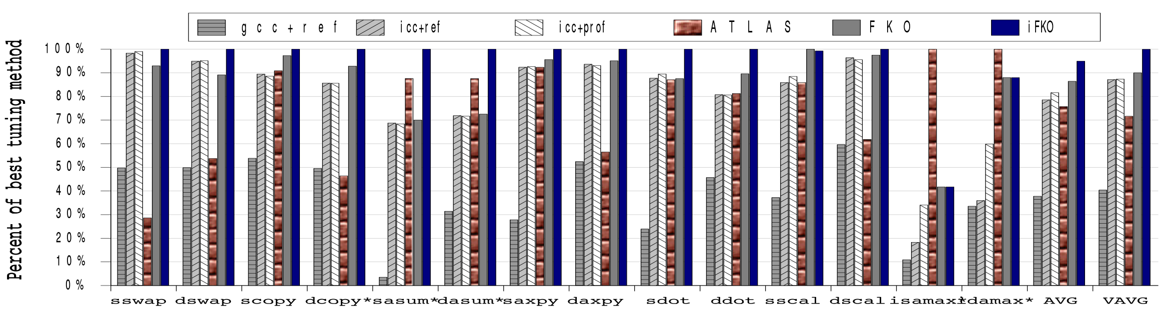
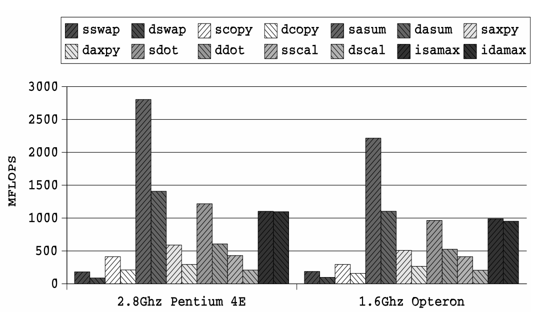
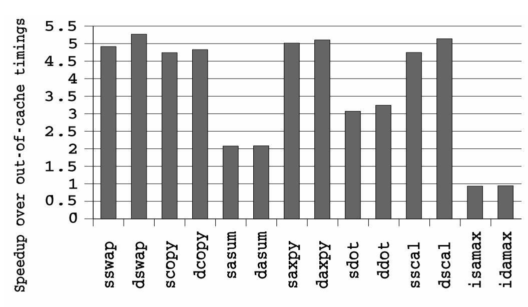


Figure 4. Relative speedups of various tuning methods on 2.8Ghz P4E, N=1024, in-L2 cache



(a) Out of cache



(b) P4E, in Level 2 cache

Figure 5. Relative BLAS performance results

NAME	Operation Summary	FLOPs
swap	`for (i=0; i < N; i++)` `{tmp=y[i]; y[i]=x[i]; x[i]=tmp}`	N
scal	`for (i=0; i < N; i++) y[i] *= alpha;`	N
copy	`for (i=0; i < N; i++) y[i] = x[i];`	N
axpy	`for (i=0; i < N; i++)` `y[i] += alpha * x[i];`	$2N$
dot	`for (dot=0.0,i=0; i < N; i++)` `dot += y[i] * x[i];`	$2N$
asum	`for (sum=0.0,i=0; i < N; i++) sum += fabs(x[i])`	$2N$
iamax	`for (imax=0,maxval=fabs(x[0]), i=1; i<N; i++){` `if (fabs(x[i]) > maxval)` `{ imax = i; maxval = fabs(x[i]); }` `}`	$2N$

Table 1. Level 1 BLAS summary

PLATFORM	COMP	FLAGS
2.8 Ghz P4E (Pentium 4E)	icc 8.0	-xP -O3 -mp1 -static
	gcc 3.3.2	-fomit-frame-pointer -O3 -funroll-all-loops
1.6 Ghz Opt (Opteron)	icc 8.0	-xW -O3 -mp1 -static
	gcc 3.3.2	-fomit-frame-pointer -O3 -O -mfpmath=387 -m64

Table 2: Compiler and flag information by platform

```
LOOP i = 0, N
LOOP_BODY
   x = X[0];
   y = Y[0];
   dot += x * y;
   X += 1;
   Y += 1;
LOOP_END
```

(a) `dot` loop

```
   LOOP i = N, 0, -1
   LOOP_BODY
      x = X[0];
      x = ABS x;
      IF (x > amax)
         GOTO NEWMAX;
ENDOFLOOP:
         X += 1;
   LOOP_END
   RETURN imax

NEWMAX:
   amax = x
   imax = N-i
   GOTO ENDOFLOOP;
```

(b) `amax` loop

Figure 6: Relevant portion of HIL implementation

	P4E, out-of-cache				Opteron, out-of-cache				P4E, in-L2 cache			
BLAS	**SV**: **WNT**	**PF** X INS:DST	**PF** Y INS:DST	**UR**: **AE**	**SV**: **WNT**	**PF** X INS:DST	**PF** Y INS:DST	**UR**: **AE**	**SV**: **WNT**	**PF** X INS:DST	**PF** Y INS:DST	**UR**: **AE**
sswap	Y:Y	none:0	nta:1920	1:0	Y:N	nta:1536	nta:1024	1:0	Y:N	t0:512	t0:1152	2:0
dswap	Y:Y	none:0	nta:1024	4:0	Y:N	nta:960	nta:512	1:0	Y:N	none:0	nta:1792	4:0
scopy	Y:Y	none:0	none:0	2:0	Y:Y	none:0	none:0	1:0	Y:N	nta:1408	nta:1536	2:0
dcopy	Y:Y	none:0	none:0	2:0	Y:Y	none:0	none:0	1:0	Y:N	nta:1408	nta:128	2:0
sasum	Y:N	nta:1024	n/a:0	5:5	Y:N	t0:1664	n/a:0	4:4	Y:N	nt0:896	n/a:0	16:2
dasum	Y:N	nta:1024	n/a:0	5:5	Y:N	nta:1920	n/a:0	4:4	Y:N	nta:1536	n/a:0	16:2
saxpy	Y:Y	t0:640	t0:1152	1:0	Y:N	nta:1984	nta:2048	4:0	Y:N	nta:512	nta:512	32:0
daxpy	Y:Y	nta:1152	nta:256	1:0	Y:N	nta:832	nta:448	1:0	Y:N	t0:1152	t0:384	32:0
sdot	Y:N	nta:1536	nta:384	3:3	Y:N	nta:1984	nta:1088	2:2	Y:N	nta:512	nta:768	64:4
ddot	Y:N	nta:1152	nta:384	3:3	Y:N	t0:1536	t0:576	3:3	Y:N	nta:1664	nta:1536	64:4
sscal	Y:Y	nta:1024	n/a:0	1:0	Y:N	nta:640	n/a:0	1:0	Y:N	nta:768	n/a:0	8:0
dscal	Y:Y	nta:1536	n/a:0	1:0	Y:N	nta:1536	n/a:0	1:0	Y:N	nta:1664	n/a:0	8:0
isamax	N:N	t0:768	n/a:0	8:0	N:N	nta:768	n/a:0	16:0	N:N	nta:56	n/a:0	32:0
idamax	N:N	nta:1280	n/a:0	8:0	N:N	nta:1920	n/a:0	32:0	N:N	t0:128	n/a:0	32:0

Table 3. Transformation parameters by architecture and context

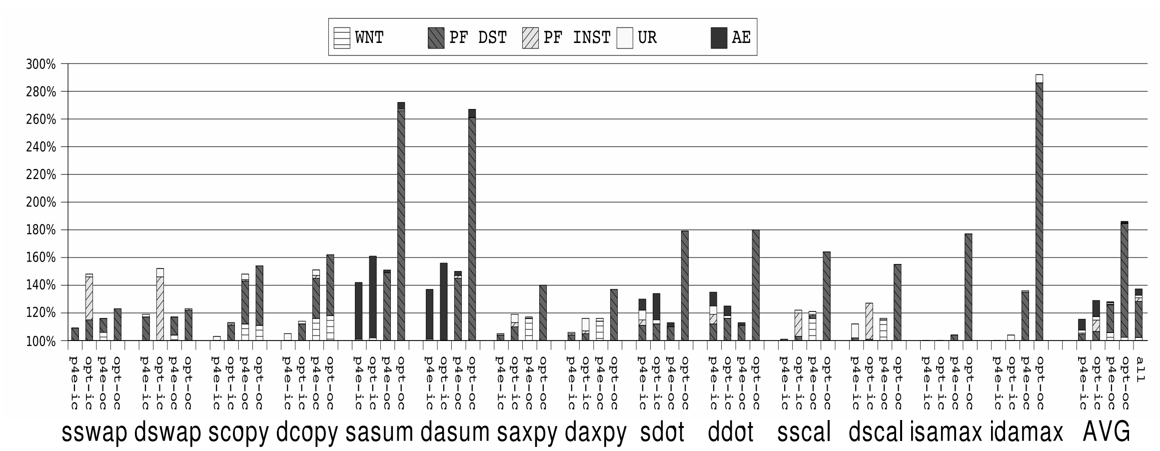


Figure 7. Percent of FKO performance by transform due to empirical search