AUTOMATED EMPIRICAL OPTIMIZATION OF HIGH PERFORMANCE FLOATING POINT KERNELS

Name: R. Clint Whaley
Department: Department of Computer Science
Major Professor: David Whalley
Degree: Doctor of Philosophy
Term Degree Awarded: Fall, 2004

Using traditional methodologies and tools, the problem of keeping performance-critical kernels at high efficiency on hardware evolving at the incredible rates dictated by Moore's Law is almost intractable. On product lines where ISA compatibility is maintained through several generations of architecture, the growing gap between the machine that the software sees and the actual hardware exacerbates this problem considerably, as do the evolving software layers between the application in question and the ISA. To address this problem, we have utilized a relatively new technique, which we call AEOS (Automated Empirical Optimization of Software). In this paper, we describe the AEOS systems we have researched, implemented and tested. The first of these is ATLAS (Automatically Tuned Linear Algebra Software), which empirically optimizes key linear algebra kernels to arbitrary cache-based machines. Our latest research effort is instantiated in the iFKO (iterative Floating Point Kernel Optimizer) project, whose aim is to perform empirical optimization of relatively arbitrary kernels using a low-level iterative and empirical compilation framework.

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS & SCIENCES

AUTOMATED EMPIRICAL OPTIMIZATION OF HIGH PERFORMANCE FLOATING POINT KERNELS

By

R. CLINT WHALEY

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Fall Semester, 2004

The members of the Committee approve the dissertation of R. Clint Whaley defended on November 2, 2004.

David Whalley
Professor Directing Dissertation

Gordon Erlebacher
Outside Committee Member

Theodore Baker
Committee Member

Michael Mascagni
Committee Member

Xin Yuan
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Using traditional methodologies and tools, the problem of keeping performance-critical kernels at high efficiency on hardware evolving at the incredible rates dictated by Moore's Law is almost intractable. On product lines where ISA compatibility is maintained through several generations of architecture, the growing gap between the machine that the software sees and the actual hardware exacerbates this problem considerably, as do the evolving software layers between the application in question and the ISA. To address this problem, we have utilized a relatively new technique, which we call AEOS (Automated Empirical Optimization of Software). In this paper, we describe the AEOS systems we have researched, implemented and tested. The first of these is ATLAS (Automatically Tuned Linear Algebra Software), which empirically optimizes key linear algebra kernels to arbitrary cache-based machines. Our latest research effort is instantiated in the iFKO (iterative Floating Point Kernel Optimizer) project, whose aim is to perform empirical optimization of relatively arbitrary kernels using a low-level iterative and empirical compilation framework.

CHAPTER 1

INTRODUCTION

The ultimate goal of this research is to provide compute kernels for the high performance computing (HPC) community that run at near-peak efficiency, even as architectures evolve at the frantic pace dictated by Moore's Law. If a kernel's performance is to be made at all robust, it must be both *portable*, and of even greater importance these days, *persistent*. We use these terms to separate two linked, but slightly different forms of robustness. The platform on which a kernel must run can change in two different ways: the machine ISA (Instruction Set Architecture) can remain constant even as the hardware implementing that ISA varies, or the ISA can change. When a kernel maintains its efficiency on a given ISA as the underlying hardware changes, we say it is *persistent*, while a *portably* optimal code achieves high efficiency even as the ISA and machine are changed.

Before the results of this research can be evaluated, it is important to demonstrate that there truly is a problem that needs to be solved, and thus the bulk of this introduction is dedicated to demonstrating why we have undertaken this line of research. Therefore, Section 1.1 overviews the need for highly tuned kernels in HPC, Section 1.2 discusses the traditional approaches to this problem, and gives the reasons why they are inadequate in practice, which in turn motivates the application of empirical techniques (the subject of this research), as discussed in Section 1.2.3. After this motivation, Section 1.3 provides a brief history of this research, and Section 1.4 describes the organization of the remainder of the paper.

1.1 Importance of Kernel Optimization for HPC

High performance computing is differentiated from general computing by its voracious appetite for computing resources. Despite hardware performance that has been steadily improving according to Moore's Law, this is as true today as it was a decade ago. Scientific modeling provides an illustration of this phenomenon. In many of these applications, computational power is the main constraint preventing the scientist from modeling more complex problems, which would then more closely match reality. As more computational power becomes available, the scientist typically increases the complexity/accuracy of the model until the limits of the computational power are reached. Therefore, since many applications have no practical limit of "enough" accuracy, it is important that each generation of increasingly powerful computers have well optimized computational kernels, which in turn allow for efficient execution of the higher-level applications that use them.

1.2 Problems with Traditional HPC Kernel Production Methods

The traditional path to achieving high performance in HPC involves compilation research combined with library production. General purpose compilers do not, in practice, achieve the very high percentages of peak on the complex kernels demanded by HPC applications (the reasons for this are outlined in Section 1.2.2). Therefore, since a user cannot write an arbitrary code and expect it to run at the extreme efficiencies demanded by HPC applications, the community has responded by emphasizing library production. In particular, APIs for reusable performance kernels are standardized, allowing these kernels to be hand-tuned by teams of experts for a given platform. Once these standard kernels are available for the platform of interest, higher-level applications that leverage them can run at high efficiencies without extensive additional tuning.

Both hand-tuning of kernels and traditional compilation have severe drawbacks when employed for performance-critical kernel production, as discussed in the following sections. Since traditional compilation shares many of the same drawbacks as hand-tuning, as well as having its own unique problems, we discuss hand-tuning first.

1.2.1 Shortcomings of Hand-tuning

Hand-tuning performance-critical kernels for each architecture of interest suffers from two main drawbacks: First, creating software that realizes near peak rates of execution requires detailed knowledge of a complex set of interrelated factors, including the operation being optimized, the target architecture(s), and all the intervening software layers. Even when the implementer possesses such broad understanding, the interactions between various hardware/software layers guarantee that significant empirical tuning of the initial kernel will be required. Therefore, optimizing even the simplest of real-world operations for high performance usually requires a sustained effort from the most technically advanced programmers, which are in critically short supply. Second, even when the requisite programming talent is available, hand-tuning such codes is a time consuming task, so that far too often, when the optimized libraries are finally ready to come on line, the generation of hardware for which they are optimized is well on its way towards obsolescence. This difficulty of keeping software highly optimized in the face of hardware change is a persistent problem for both hand-tuning and compilers.

1.2.2 Shortcomings of Traditional Compilation

The most fundamental reason traditional compilers do not achieve the high percentages of peak required by HPC kernels is that it is not what they are designed to do. Optimizing general-purpose code to this extreme degree would

be counter-productive: It would require substantially greater time to develop the compiler itself, would have almost no effect on overall performance for most codes, and would almost certainly increase compilation times to a degree intolerable for general use.

Even if the compiler were written with this kind of extreme optimization in mind, traditional compilation techniques would clearly need to be supplemented in some way. Traditionally, compilers perform transformations based on models that attempt to capture the relevant details of the underlying architecture. This approach works well for general purpose computing, but the model needs to be much more detailed to extract near-peak levels of performance: indeed, it needs to be so detailed that in practice producing such a model would be almost intractable. Even if a model could be created that was sophisticated enough to account for the interactions between all levels of cache, the pipelines of all relevant functional units, and all shared hardware resources required by a given operation (and it could do such detailed front-end analysis that all required kernel-specific information was extracted from the code), it is often the case that much of the data required to build such a model is unknown, either because the hardware vendor considers it proprietary, or because even the designers are unable to predict performance due to unforeseen resource interactions. Therefore, models require significant hand-tuning to each supported architecture; this is true even in general computing, and the cost would clearly go up dramatically for a kernel-oriented compiler.

Therefore, to obtain near-peak efficiency for kernels using traditional compilation, the models must be constantly modified to keep up with hardware being released at the rate dictated by Moore's Law. Since compilers are generally very complex applications, this is just as untenable in the long run as the hand-tuned kernel optimization discussed in the previous section. Moore's Law also provides a secondary effect that makes model-based approaches even more problematic for this type of tuning. Since software evolves at a much slower rate than Moore's Law, hardware architects

must retain ISA compatibility whenever possible, which can lead to mismatches between an ISA and the underlying hardware. At the same time, the additional circuits that can be economically added to a wafer have resulted in architectures that perform an increasing number of 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 trend 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.

Another problem with model-based approaches is how to allocate resources in competing optimization phases. Here heuristics must be employed, which may leave significant gaps in optimization (eg., reserve several registers when doing register assignment so that software pipelining can be performed later, etc.). Finally, even if all these challenges could be surmounted, kernel- and context-specific issues provide a further barrier to achieving high performance. For instance, tuning a bus-bound operation requires a different set of priorities than tuning a kernel that is primarily cpu-bound, the types and number of operands strongly affect the correct optimization scheme, etc. Profiling can discover some of this information, but quite a few relevant details cannot be realistically discovered even when the most aggressive traditional techniques are employed. If such perfect analysis were available, however, these complicated architectural models would still need to be split into many subcases to reflect varying usage patterns, worsening an already insupportable maintenance problem.

1.2.3 Addressing Optimization Challenges through Empirical Techniques

These problems, taken together, led to the implementation of empirically tuned library generators such as PHiPAC [1], FFTW [2, 3, 4] (these and other packages are discussed in the related work sections of Chapters 3 and 4) and our own ATLAS [5, 6, 7, 8, 9, 10] (discussed in Chapter 3). 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. Thus, the need to understand the architecture in detail is removed: we are probing the system as it stands, just as the empirical technique of the scientific method probes the natural world, and just as the scientific method discards disprovable theories, we do not retain transformations that do not result in sufficient speedup.

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. Chapters 4 and 5 outline our own empirical compiler research.

1.3 History of Research

The author began this line of research while a he was a full-time researcher at the University of Tennessee, Knoxville (UTK). In this initial work, the emphasis was on achieving *portably* optimal code for a restricted set of linear algebra routines. This research culminated in the ATLAS [6, 7, 8, 9, 10] project. After ATLAS proved so successful in optimizing a given set of operations, the obvious question was how it could be generalized, and this has led to the research we have conducted here at Florida State University (FSU) on iFKO (iterative Floating Point Kernel Optimizer). This work is aimed at generalizing empirical optimization to arbitrary floating point kernels, and concentrates on achieving *persistently* optimal code.

The ATLAS work is included in this dissertation for two main reasons. Most importantly, it combined with iFKO comprise the author's contribution to the emerging field of empirical optimization, and this first effort both supports and lends direction to our current research. Secondly, while the majority of the ATLAS framework was indeed developed at UTK, significant ATLAS work was done here at FSU as well, including FSU grant #1327-592-45, and the publications of [10, 11]. Work on the grant led particularly to some intensive assembly hand-tuning to exploit SIMD vectorization, and this and related efforts strongly influenced the design and implementation of iFKO.

1.4 Organization of Paper

The remainder of this paper is organized in the following way: Chapter 2 introduces the terminology used to describe these empirical techniques, Chapter 3 overviews our ATLAS work, Chapter 4 uses this foundation to motivate and explain

our empirical compilation research, Chapter 5 describes the current implementation of our empirical compilation framework, and Chapter 6 provides experimental results. Note that the ATLAS and iFKO chapters contain their own related work section, as these efforts are distinct in both approach and time. Finally, Chapter 7 summarizes our findings and contributions as well as discussing areas for future work.

CHAPTER 2

BASIC DEFINITIONS – AEOS

Many groups have begun to utilize automated and empirical approaches to optimization, resulting in a plethora of differing terminologies, including "self-tuning libraries", "adaptive software", "empirical compilation", "iterative compilation", etc. While these approaches differ strongly in details, in order to fall into the classification related to our research they must have some commonalities:

1. The search must be *automated* in some way, so that an expert hand-tuner is not required.

2. The decision of whether a transformation is useful or not must be *empirical*, in that an actual timing measurement on the specific architecture in question is performed, as opposed to the traditional application of transformations using static heuristics or profile counts

3. These methods must have some way to vary/adapt the software being tuned.

With these broad outlines in mind, we lump all such empirical tunings under the acronym AEOS, or Automated Empirical Optimization of Software, and Section 2.1 outlines the requirements of such systems, while Section 2.2 discusses the studied methods of software adaptation.

2.1 Basic AEOS Requirements

The basic requirements for supporting high performance kernel optimization using AEOS methodologies are:

- *Isolation of performance-critical routines:* Just as with traditional libraries, the performance-critical sections of code must be isolated (usually into subroutines, which dictates the need for an standardized API).

- *A method of adapting software to differing environments:* Since AEOS depends on iteratively trying differing ways of performing the performance-critical operation, the author must be able to provide implementations that instantiate a wide range of optimizations. This may be done very simply, for instance by having parameters in a fixed code which, when varied, correspond to differing cache sizes, etc., or it may be done much more generally, for instance by supplying a highly parameterized source generator which can produce an almost infinite number of implementations. No matter how general the adaptation strategy, there will be limitations or built-in assumptions about the required architecture which should be identified in order to estimate the probable boundaries on the code's flexibility. Section 2.2 discusses software adaptation methods in further detail.

- *Robust, context-sensitive timers:* Since timings are used to select the best code, it becomes very important that these timings be accurate. Since few users can guarantee single-user access, the timers must be robust enough to produce reliable timings even on heavily loaded machines. Furthermore, the timers need to replicate as closely as possible the way in which the given operation will be used. For instance, if the routine will normally be called with cold caches, cache flushing will be required. If the routine will typically be called with a given level of cache preloaded, while others are not, that too should be taken into account. If there is no known machine state, timers allowing for many different states, which the user can vary, should be created.

- *Appropriate search heuristic* The final requirement is a search heuristic which automates the search for the most optimal available implementation. For a simple method of code adaptation, such as supplying a fixed number of hand-tuned implementations, a simple linear search will suffice. However, when using sophisticated source generators with literally hundreds of thousands of ways of doing an operation, a similarly sophisticated search heuristic must be employed in order to prune the search tree as rapidly as possible, so that the optimal cases are both found and found quickly (obviously, few users will tolerate heavily parameterized search times with exponential growth). If the search takes longer than a handful of minutes, it needs to be robust enough to not require a complete restart if hardware or software failure interrupts the original search.

2.2 Methods of Software Adaptation

We employ three different methods of software adaptation. The first is widely used in programming in general, and it involves parameterizing characteristics which vary from machine to machine. In linear algebra, the most important of such parameters is probably the blocking factor used in blocked algorithms, which, when varied, varies the data cache utilization. In general, parameterizing as many levels of data cache as the algorithm can support can provide remarkable speedups. With an AEOS approach, such parameters can be compile-time variables, and thus not cause a runtime slowdown. We call this method *parameterized adaptation*.

Not all important architectural variables can be handled by parameterized adaptation (simple examples include instruction cache size, choice of combined or separate multiply and add instructions, length of floating point and fetch pipelines, etc), since varying them actually requires changing the underlying source code. This then brings in the need for the second method of software adaptation, *source code*

adaptation, which involves actually generating differing implementations of the same operation.

There are at least two different ways to do source code adaptation. Perhaps the simplest approach is for the designer to supply various hand-tuned implementations, and then the search heuristic may be as simple as trying each implementation in turn until the best is found. At first glance, one might suspect that supplying these multiple implementations would make even this approach to source code adaptation much more difficult than the traditional hand-tuning of libraries. However, traditional hand-tuning is not just the mere application of known techniques it may appear when examined casually. Knowing the size and properties of your level 1 cache is not sufficient to choose the best blocking factor, for instance, as this depends on a host of interlocking factors which defy often a priori understanding in the real world. Therefore, it is common in hand-tuned optimizations to utilize the known characteristics of the machine to narrow the search, but then the programmer writes various implementations and chooses the best.

For the simplest AEOS implementation, this process remains the same, but the programmer adds a search and timing layer to accomplish what would otherwise be done by hand. In the simplest cases, the time to write this layer may not be much if any more than the time the implementer would have spent doing the same process in a less formal way by hand, while at the same time capturing at least some of the flexibility inherent in AEOS-centric design. We will refer to this source code adaptation technique as *multiple implementation*. Due to its obvious simplicity, this method is highly parallelizable, in the sense that multiple authors can meaningfully contribute without having to understand the entire package. In particular, various specialists on given architectures can provide hand-tuned routines without needing to understand other architectures, the higher level codes (e.g. timers, search heuristics, higher-level routine which utilize these basic kernels, etc). This makes multiple

implementation a very good approach if the user base is large and skilled enough to support an open source initiative along the lines of, for example, Linux.

The second method of source code adaptation is *source generation*. In source generation, a source generator (i.e., a program that writes other programs) is produced. This source generator takes as parameters the various source code adaptations to be made. As before, simple examples include instruction cache size, choice of combined or separate multiply and add instructions, length of floating point and fetch pipelines, and so on. Depending on the parameters, the source generator produces a routine with the requisite characteristics. The great strength of source generators is their ultimate flexibility, which can allow for far greater tunings than could be produced by all but the best hand-coders. However, generator complexity tends to go up along with flexibility, so that these programs rapidly become almost insurmountable barriers to outside contribution.

In our own past efforts, we have therefore combined these two methods of source adaptation, where a kernel-specific source generator is provided for maximal architectural portability. Multiple implementation is utilized to encourage outside contribution, and allows for extreme architectural specialization via assembly implementations.

Source generators that generate high-level (and thus portable) languages such as FORTRAN or ANSI C (as opposed to low-level and non-portable languages such as assembly) have the advantage of being able to optimize a given operation for any architecture which possesses the requisite compiler. However, such source generators are specific to the kernel being tuned, and thus we can say they are *architecture/platform independent*, but *routine/operation specific*. Multiple implementation is obviously routine specific as well, and is architecture dependent (assembly) or independent (high level languages) depending on the implementation language.

Therefore, our past efforts have resulted in a AEOS-enabled library that is largely platform independent, but operation specific. In our iFKO work, we generalize these

techniques using a third method of software adaptation, which will be more platform specific, but routine independent. In addition, to augment our present strengths, we believe it is important have a mechanism to exploit particular architectural features not necessarily available in high level languages such as ANSI C. This was accomplished using an *iterative and empirical compiler*, hereafter shortened to *empirical compiler*.

2.2.1 Summary of Software Adaptation Methods

In summary, we use three tools in order to perform the required software adaptation (we hereafter treat multiple implementation and source generation as separate techniques, even though they are sub-classes of source code adaptation), and their strengths and weaknesses are summarized in Table 2.1. All three of these methodologies can be further augmented by parameterized adaptation.

Table 2.1: Summary of software adaptation techniques

ADAPTATION METHOD	PLATFORM INDEP.	ROUTINE INDEP.	OUTSIDE CONTRIB.	AUTOMATIC ADAPTABILITY
Multiple Implementation	YES	NO	EASY	LOW
Source Generator	YES	NO	DIFFICULT	HIGH
Empirical Compiler	NO	YES	DIFFICULT	INTERMEDIATE

We have previously discussed all of the columns of this table except the last, automatic adaptability. This column gives an indication on how likely the method is to provide good performance as the package is moved to differing architectures. Multiple implementation has the lowest adaptability, since users rarely write implementations for architectures they are not using. There is still some adaptability, and the closer the new architecture is to one of the ones previously seen, the better

multiple implementation will perform. However, multiple implementation is very likely to provide poor performance in the face of fundamental architectural change. In contrast, a general source generator, which can be built to be very flexible indeed, is very likely to be able to adapt to all but the most extreme changes in architecture.

The empirical compiler is given an intermediate adaptability rating. In order to use the full capabilities of the compiler, the backend must be ported, which is an obvious constraint on adaptability. More specifically, an empirical compiler should adapt well to varying architectures that implement a given ISA (i.e., it delivers persistent optimization), but the backend must be ported to all ISAs of interest in order to adapt to varying ISAs.

This table also provides the basis for understanding why all three mechanisms are desirable. The source generator is the most flexible in overall adaptability, multiple implementation allows for outside contribution and hand tuning, and an empirical compiler provides the opportunity to tune a wider array of kernels.

Because an automated search can try many more techniques than even the most motivated hand-tuner, we believe iFKO will ultimately make hand-tuning on supported platforms unnecessary. While iFKO's transformation palette is incomplete, however, multiple implementation (where iFKO is considered just another compiler) can be used to provide so-far unsupported optimizations. Further, multiple implementation provides an easy way to quickly try various optimization strategies, in order to find transformations worth adding to iFKO.

CHAPTER 3

FOUNDATIONAL WORK – ATLAS

ATLAS is the project from which our current understanding of AEOS methodologies grew, and now provides a test bed for their further development and testing. The initial goal of ATLAS was to provide a portably efficient implementation of the BLAS[12, 13, 14, 15, 16]. ATLAS now provides at least some level of support for all of the BLAS, and the first tentative extensions beyond this one API have been taken (for example, the most recent ATLAS release contained some higher level routines from the LAPACK [17] API). Since the BLAS represent the kernels which are empirically tuned, this paper will concentrate on ATLAS's BLAS support.

The BLAS (Basic Linear Algebra Subroutines) are building block routines for performing basic vector and matrix operations. The BLAS are divided into three levels: Level 1 BLAS do vector-vector operations, Level 2 BLAS do matrix-vector operations, and the Level 3 BLAS do matrix-matrix operations. The performance gains from optimized implementations is strongly affected by the level of the BLAS.

In the Level 1 BLAS, no memory reuse is possible, and therefore many Level 1 BLAS are completely memory-bound if they do not operate on in-cache data. For some Level 1 BLAS, prefetch and related techniques can still produce impressive speedups; however, some operations are so memory-bound that the bus is always saturated regardless of prefetch arrangements, so that out-of-cache speedups are essentially unrealizable. Even in these routines, however, it is important to perform all applicable computational optimizations, as inadequate computational optimization may cause additional delay in issuing the critical fetch operations.

In the Level 2 BLAS, memory blocking can allow for reuse of the vector operands, but not, in general, of the matrix operand (the exception is that some matrix types, for instance symmetric or Hermitian, can effectively use each matrix operand twice). Reducing the vector operands from $O(N^2)$ to $O(N)$ represents considerable savings over naive code, but due to the irreducible matrix costs, the memory load remains of the same order ($O(N^2)$) as the operation count. Therefore, the Level 2 BLAS can enjoy modest speedup (say, roughly in the range of 10-300% for out-of-cache timings), both because memory blocking is effective, and because the loops are complex enough that more compilers begin having problems doing the floating point optimizations automatically.

Finally, the Level 3 BLAS can display orders of magnitude speedups. To simplify greatly, these operations can be blocked such that the natural $O(N^3)$ fetch costs become essentially $O(N^2)$. Further, the triply-nested loops used here are almost always too complex for the compiler to figure out without hints from the programmer (eg, some explicit loop unrolling), and thus the $O(N^3)$ computation cost can be greatly optimized as well.

The following sections discuss our handling of all BLAS levels in ATLAS. Because of the amount of effort required to provide high-quality AEOS software, it becomes critical to find the smallest possible kernels which can be leveraged to supply all required functionality. Thus, each section describes the low level performance kernels, the techniques used to create them, and how these kernels are utilized to produce all required functionality.

3.1 Limits of ATLAS's Approach

As previously mentioned, any AEOS approach is bound to have some restrictions on its adaptability. ATLAS is no exception, and the following assumptions need to hold true for ATLAS to perform well:

1. *Adequate ANSI C compiler*: ATLAS is written entirely in ANSI/ISO C, with the exception of the FORTRAN 77 interface codes (which are simple wrappers written in ANSI FORTRAN 77, calling the C internals for computation). ATLAS does not require an excellent compiler, since it uses source generation to perform many optimizations typically done by compilers. However, too-aggressive compilers can transform already optimal code into suboptimal code, if flags do not exist to turn off certain compiler optimizations. On the other hand, compilers without the ability to effectively use the underlying ISA (eg., inability to utilize registers, even when the C code calls for them), will yield poor results as well.

2. *Hierarchical memory*: ATLAS assumes a hierarchical memory is present. Best results will be obtained when both registers and at least an L1 data cache are present.

Of these two, an adequate C compiler is the most important restriction. Even lack of hierarchical memory would at worst turn some of ATLAS's blocking and register usage into overheads. Even with this handicap, ATLAS's source adaptation may still yield enough performance to provide an adequate BLAS. If the ANSI C compiler is poor enough, however, this can result in the computational portion of the algorithms being effectively unoptimized. Since the computational optimizations are the dominant cost of a blocked Level 3 BLAS, this can produce extremely poor results. Note that multiple implementation, with its support for assembly as well as ANSI C, can be used to get around even this restriction. If the machine in question does not share an ISA with a previously seen machine, however, we will be back to the familiar problem of having optimization wait on hand-tuning.

3.2 AEOS Tuning for the Level 3 BLAS in ATLAS

All thirty routines of the Level 3 BLAS (for each real data type there are six Level 3 BLAS, and nine routines for each complex data type) can be efficiently implemented given an efficient matrix-matrix multiply (for details on how this is done, [10] discusses ATLAS's particular implementation, and other approaches are given in [18, 19, 20, 21]). Thus the main performance kernel is general matrix matrix multiply (hereafter shortened to matmul, or the BLAS matmul routine name, GEMM). As subsequent sections show, however, GEMM itself is further narrowed down to an even smaller kernel before source generation takes place.

The BLAS supply a routine GEMM, which performs a general matrix-matrix multiplication of the form $C \leftarrow \alpha op(A)op(B) + \beta C$, where $op(X) = X$ or X^T. C is an $M \times N$ matrix, and $op(A)$ and $op(B)$ are matrices of size $M \times K$ and $K \times N$, respectively.

In general, the arrays A, B, and C will be too large to fit into cache. Using a block-partitioned algorithm for matrix multiply, it is still possible to arrange for the operations to be performed with data for the most part in cache by dividing the matrix into blocks. For additional details see [22].

Using this BLAS routine, the rest of the Level 3 BLAS can be efficiently supported, so GEMM is the Level 3 BLAS computational kernel. In ATLAS, this BLAS-level GEMM is written as a series of high level codes which use compile- or run-time variables to adapt to cache levels. These high-level codes get most of their adaptation from a lower-level kernel (discussed in Section 3.2.2), which is adapted to the architecture using parameterized adaptation, multiple implementation, *and* source generation.

3.2.1 Building the General Matrix Multiply from the L1 Cache-contained Multiply

This section describes the non-generated code, whose only variance across platforms come from parameterization. These codes are used to form the BLAS's general matrix-matrix multiply using a L1 cache-contained matmul (hereafter referred to as the L1 matmul).

Section 3.2.2 describes the L1 matmul and its generator in detail. For our present discussion, it is enough to know that ATLAS has at its disposal highly optimized routines for doing matrix multiplies whose dimensions are chosen such that cache blocking is not required (i.e., the hand-written code discussed in this section deals with cache blocking; the generated code assumes things fit into cache).

When the user calls GEMM, ATLAS must decide whether the problem is large enough to tolerate copying the input matrices A and B. If the matrices are large enough to support this $O(N^2)$ overhead, ATLAS will copy A and B into block-major format. ATLAS's block-major format breaks up the input matrices into contiguous blocks of a fixed size N_B, where N_B is chosen as discussed in Section 3.2.2 in order to maximize L1 cache reuse. Once in block-major format, the blocks are contiguous, which eliminates TLB problems, minimizes cache thrashing and maximizes cache line use. It also allows ATLAS to apply alpha (if alpha is not already one) to the smaller of A or B, thus minimizing this cost as well. Finally, the package can use the copy to transform the problem to a particular transpose setting, which for load and indexing optimization, is set so A is copied to transposed form, and B is in normal (non-transposed) form. This means our L1-cache contained code is of the form $C \leftarrow A^T B$, $C \leftarrow A^T B + C$, and $C \leftarrow A^T B + \beta C$, *where all dimensions, including the non-contiguous stride, are known to be* N_B. Knowing all of the dimensions of the loops allows for arbitrary unrollings (i.e., if the instruction cache could support it, ATLAS could unroll all loops completely, so that the L1 cache-contained multiply had no loops at all). Further, when the source generator knows the leading dimension

of the matrices (i.e., the row stride), all indexing can be done explicitly, without the need for expensive integer or pointer computations.

If the matrices are too small, the $O(N^2)$ data copy cost can actually dominate the algorithm cost, even though the computation cost is $O(N^3)$. For these matrices, ATLAS will call an L1 matmul which operates on non-copied matrices (i.e. directly on the user's operands). The non-copy L1 matmul will generally not be as efficient as the copy L1 matmul; at this problem size the main performance bottleneck is memory, and so the lack of computational efficiency (mainly due to the additional pointer arithmetic required in order to support the user-supplied leading dimension) will likely only show up on in-cache operations.

The choice of when a copy is dictated and when it is prohibitively expensive is an AEOS parameter; it turns out that this crossover point depends strongly both on the particular architecture, the matmul kernel selected, and the shape of the operands (matrix shape effectively sets limits on which matrix dimensions can enjoy cache reuse). To handle this problem, ATLAS simply compares the speed of the copy and non-copy L1 matmul for variously shaped matrices, varying the problem size until the copying provides a speedup (on some platforms, and with some shapes, this point is never reached). These crossover points are determined at install time, and then used to make this decision at runtime. Because it is the dominant case, this paper describes only the copied matmul algorithm in detail.

There are presently two algorithms for performing the general matrix-matrix multiply. The two algorithms correspond to different orderings of the loops; i.e., is the outer loop over M (over the rows of A), and thus the second loop is over N (over the columns of B), or is this order reversed. The dimension common to A and B (i.e., the K loop) is currently always the innermost loop.

Let us define the input matrix looped over by the outer loop as the outer or outermost matrix; the other input matrix will therefore be the inner or innermost matrix. Both algorithms have the option of writing the result of the L1 matmul

directly to the matrix, or to an output temporary $\hat{C}$. The advantages to writing to $\hat{C}$ rather than C are:

1. Address alignment may be controlled (i.e., the code can ensure during the malloc that $\hat{C}$ begins on a cache-line boundary).

2. Data is contiguous, eliminating possibility of unnecessary cache-thrashing due to ill-chosen leading dimension (assuming a non-write-through cache).

The disadvantage of using $\hat{C}$ is that an additional write to C is required after the L1 matmul operations have completed. This cost is minimal if GEMM makes many calls to the L1 matmul (each of which writes to either C or $\hat{C}$), but can add significantly to the overhead when this is not the case. In particular, an important application of matrix multiply is the rank-K update, where the write to the output matrix C can be a significant portion of the cost of the algorithm. For the rank-K update, writing to $\hat{C}$ essentially doubles the write cost, which is clearly unacceptable. The routines therefore employ a heuristic to determine if the number of times the L1 matmul will be called in the K loop is large enough to justify using $\hat{C}$, otherwise the answer is written directly to C.

Regardless of which matrix is outermost, both algorithms try to allocate enough space to store the $N_B \times N_B$ output temporary, $\hat{C}$ (if needed), 1 panel of the outermost matrix, and the entire inner matrix. If this fails, the algorithms attempt to allocate smaller work arrays, the smallest acceptable workspace being enough space to hold $\hat{C}$, and 1 panel from both A and B. The minimum workspace required by these routines is therefore $2KN_B$, if writing directly to C, and ${N_B}^2 + 2KN_B$ if not. If this amount of workspace cannot be allocated, the previously mentioned non-copy code is called instead.

If there is enough space to copy the entire innermost matrix, there are several benefits to doing so:

- Each matrix is copied only one time.

- If all of the workspaces fit into L2 cache, the algorithm enjoys complete L2 reuse on the innermost matrix.

- Data copying is limited to the outermost loop, protecting the inner loops from unneeded cache thrashing.

Of course, even if the allocation succeeds, using too much memory might result in unneeded swapping. Therefore, the user can set a maximal amount of workspace that ATLAS is allowed to have, and ATLAS will not try to copy the innermost matrix if this maximum workspace requirement is exceeded.

If enough space for a copy of the entire innermost matrix is not allocated, the innermost matrix will be entirely copied for each panel of the outermost matrix (i.e., if A is our outermost matrix, ATLAS will copy B $\lceil M/N_B \rceil$ times). Further, our usable L2 cache is reduced (the copy of a panel of the innermost matrix will take up twice the panel's size in L2 cache; the same is true of the outermost panel copy, but that will only be seen the first time through the secondary loop).

Regardless of which looping structure or allocation procedure used, the inner loop is always along K. Therefore, the operation done in the inner loop by both routines is the same, and it is shown in Figure 3.1.

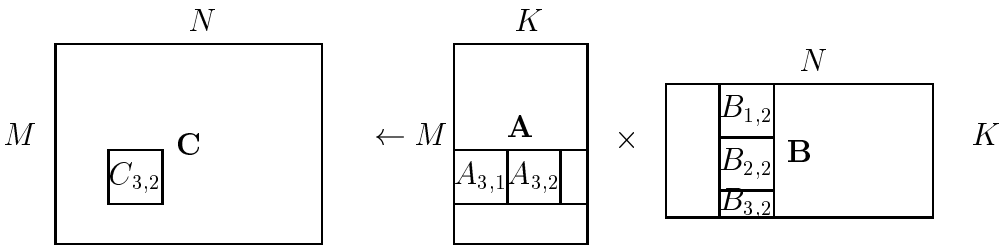


Figure 3.1: One step of matrix-matrix multiply

If GEMM is writing to $\hat{C}$, the following actions are performed in order to calculate the $N_B \times N_B$ block $C_{i,j}$, where i and j are in the range $0 \leq i < \lceil M/N_B \rceil$, $0 \leq j < \lceil N/N_B \rceil$:

1. Call L1 matmul of the form $C \leftarrow AB$ to multiply block 0 of the row panel i of A with block 0 of the column panel j of B.

2. Call L1 matmul of form $C \leftarrow AB + C$ to multiply block k of the row panel i of A with block k of the column panel j of B, $\forall k, 1 \leq k < \lceil K/N_B \rceil$. The L1 matmul is performing the operation $C \leftarrow AB + C$, so as expected this results in multiplying the row panel of A with the column panel of B.

3. $\hat{C}$ now holds the product of the row panel of A with the column panel of B, so ATLAS now performs the block write-back operation $C_{i,j} \leftarrow \hat{C}_{i,j} + \beta C_{i,j}$.

If ATLAS is writing directly to C, this action becomes:

1. Call L1 matmul of the correct form based on user-defined β (eg. if $\beta == -1$, use $C \leftarrow AB - C$) to multiply block 0 of the row panel i of A with block 0 of the column panel j of B.

2. Call L1 matmul of form $C \leftarrow AB + C$ to multiply block k of the row panel i of A with block k of the column panel j of B, $\forall k, 1 \leq k < \lceil K/N_B \rceil$.

Building from this inner loop, ATLAS has differing loop orderings which provide two algorithms for the full matmul. Figures 3.2 and 3.3 give the pseudo-code for these two algorithms, assuming the write is directly to C (writing to $\hat{C}$ is only trivially different). For simplicity, this pseudo-code skips the cleanup necessary for cases where dimensions do not evenly divide N_B. The matrix copies are shown as if coming from the notranspose, notranspose case. If they do not, only the array access on the copy changes.

```
work = allocate((M+NB)*K)
if (allocated(work)) then
   PARTIAL_MATRIX = .FALSE.
   copy A into block major format
else
   PARTIAL_MATRIX = .TRUE.
   work = allocate(NB*2*K)
   if (.NOT.allocated(work)) call small_case_code
   return
end if
NBNB = NB * NB
do j = 1, N, NB
   Bwork = ALPHA*B(:,J:J+NB-1); Bwork in block major format
   do i = 1, M, NB
      if (PARTIAL_MATRIX) Awork = A(i:i+NB-1,:); Awork in block major format
      ON_CHIP_MATMUL(Awork(1:NB*NB), Bwork(1:NB*NB), BETA, C(i:i+NB-1, j:j+NB-1), ldc)
      do k = 2, K, NB
         ON_CHIP_MATMUL(Awork((k-1)*NBNB+1:k*NBNB), Bwork((k-1)*NBNB+1:k*NBNB),
                        1.0, C(i:i+NB-1, j:j+NB-1), ldc)
      end do
   end do
end do
```

Figure 3.2: General matrix multiplication with A as innermost matrix

```
work = allocate(N*K + NB*K)
if (allocated(work)) then
   PARTIAL_MATRIX = .FALSE.
   copy B into block major format
else
   PARTIAL_MATRIX = .TRUE.
   work = allocate(NB*2*K)
   if (.NOT.allocated(work)) call small_case_code
   return
end if
NBNB = NB * NB
do i = 1, M, NB
      Awork = ALPHA*A(i:i+NB-1,:); Awork in block major format
   do j = 1, N, NB
      if (PARTIAL_MATRIX) Bwork = B(:,J:J+NB-1); Bwork in block major format
      ON_CHIP_MATMUL(Awork(1:NBNB), Bwork(1:NBNB), BETA,
                     Cwork(i:i+NB-1, j:j+NB-1), ldc)
      do k = 2, K, NB
         ON_CHIP_MATMUL(Awork((k-1)*NBNB+1:k*NBNB), Bwork((k-1)*NBNB+1:k*NBNB),
                        1.0, Cwork(i:i+NB-1, j:j+NB-1), ldc)
      end do
   end do
end do
```

Figure 3.3: General matrix multiplication with B as innermost matrix

3.2.1.1 Choosing the Correct Looping Structure

When the call to the matrix multiply is made, the routine must decide which loop structure to call (i.e., which matrix to put as outermost). If the matrices are of different size, L2 cache reuse can be encouraged by deciding the looping structure based on the following criteria:

1. If either matrix will fit completely into the usable L2 cache, put it as the innermost matrix (algorithm gets L2 cache reuse on the entire inner matrix).

2. If neither matrix fits completely into L2 cache, put largest matrix as the outermost matrix (algorithm gets L2 cache reuse on the panel of the outer matrix, if it fits in cache, and memory usage is minimized).

The size of the usable L2 cache is not directly known by ATLAS (although the AEOS variable `CacheEdge` described in Section 3.2.1.2 will often serve the same purpose) and so these criteria are not presently used for this selection. Rather, in order to minimize workspace, and maximize the chance that condition one above occurs, the smallest matrix will always be used as the innermost matrix. If both matrices are the same size, A is selected as the innermost matrix (this implies a better access pattern for C).

3.2.1.2 Blocking for Higher Levels of Cache

Note that this paper defines the Level 1 (L1) cache as the "lowest" level of cache: the one closest to the processor. Subsequent levels are "higher": further from the processor and thus usually larger and slower. Typically, L1 caches are relatively small (eg., 8-32KB), employ least recently used replacement policies, have separate data and instruction caches, and are often non-associative and write-through. Higher levels of cache are more often write-back, with varying degrees of associativity, differing replacement polices, and often contain both instruction and data.

ATLAS detects the actual size of the L1 data cache. However, due to the wide variance in high level cache behaviors, in particular the difficulty of determining how

much of such caches are usable after line conflicts and data/instruction partitioning is done, ATLAS does not presently detect and use a explicit Level 2 cache size as such. Rather, ATLAS employs a empirically determined value called `CacheEdge`, which represents the amount of the cache that is usable by ATLAS for its particular kind of blocking.

Explicit cache blocking for the selected level of cache is only required when the cache size is insufficient to hold the two input panels and the $N_B \times N_B$ piece of C. This means that users will have optimal results for many problem sizes without employing `CacheEdge`. This is expressed formally below. Notice that conditions 1 and 2 below do not require explicit cache blocking, so the user gets this result even if `CacheEdge` is not set.

Therefore, the explicit cache blocking strategy discussed in case 4 below assumes that the panels of A and B overflow a particular level of cache. In this case, the problem can be easily partitioned along the K dimension of the input matrices such that the panels of the partitioned matrices A_p and B_p *will* fit into the cache. This means that we get cache reuse on the input matrices, at the cost of writing C additional times.

It is easily shown that the footprint of the algorithm computing a $N_B \times N_B$ section of C in cache is roughly $2KN_B + {N_B}^2$, where $2KN_B$ stores the panels from A and B, and the section of C is of size ${N_B}^2$. If the above expression is set equal to `CacheEdge`, and solved for K, it will yield the maximal K (call this quantity K_m) which will, assuming the inner matrix was copied up front, allow for reusing the outer matrix panel N/N_B times. This partitioning transforms the original matrix multiply into $\lceil K/K_m \rceil$ rank-K_m updates.

Since the correct value of `CacheEdge` is not known a priori, ATLAS empirically determines it at install time by using large matrices (whose panel sizes can be expected to overflow the cache, and thus bring up the need for explicit, rather than implicit, L2 or higher blocking), and simply trying various settings. Extremely large

caches will probably not be detected in this manner (i.e., if the user cannot allocate enough memory to cause a panel to overflow the cache, the large cache will not be detected), and some higher-level caches provide relatively small benefits and so may not be detected, in which case `CacheEdge` is set to a 4 MB (this is large enough not to depress performance even for very large problems on systems without L2 caches, and it results in less memory usage).

Assuming that matrix A is the innermost matrix, and we are discussing cache level L, of size S_L, and that main memory is classified as a level of "cache" greater than L, there are four possible states (depending on cache and problem size, and whether `CacheEdge` is set) which ATLAS may be in. These states and their associated memory access costs are:

1. If the entire inner matrix, a panel of the outer matrix, and the $N_B \times N_B$ section of C fits into the cache (eg. $MK + KN_B + {N_B}^2 \leq S_L$):
   - $K(M+N)+MN$ reads (of A, B and C, respectively) from higher level(s) cache
   - $\frac{MNK}{N_B}$ writes to first level of non-write-through cache; higher levels of cache receive only the final MN writes
2. If the cache cannot satisfy the memory requirements of 1, it may still be large enough to accommodate the two active input panels, along with the relevant section of C
   (eg., ($2KN_B + {N_B}^2 \leq S_L$ AND ATLAS copies the entire inner matrix)
   OR ($3KN_B + {N_B}^2 \leq S_L$ AND ATLAS copies a panel of the inner matrix in the inner loop, thus doubling the inner panel's footprint in the cache)):
   - $NK + \frac{MNK}{N_B} + MN$ reads (B, A and C, respectively) from higher level(s) of cache

- $\frac{MNK}{N_B}$ writes to first level of non-write-through cache; higher levels of cache receive only the final MN writes

3. If the cache is too small for either of the previous cases to hold true, (eg., $2KN_B + {N_B}^2 > S_L$) and `CacheEdge` is not set, and thus no explicit level L blocking is done, the memory access becomes:

   - $\frac{2MNK}{N_B} + MN$ reads (A, B, and C) from higher level(s) of cache
   - $\frac{MNK}{N_B}$ writes to first level of non-write-through cache; higher levels of cache receive only the final MN writes

4. Finally, if the first two cases do not apply (eg., $2KN_B + {N_B}^2 > S_L$), but `CacheEdge` is set to S_L, ATLAS can perform cache blocking to change the memory access from that given in 3 to:

   - $NK + \frac{MNK}{N_B} + \frac{MNK}{K_m}$ (B, A, C) reads from higher level(s) of cache
   - $\frac{MNK}{N_B}$ writes to first level of non-write-through cache; higher levels of cache receive at most $\frac{MNK}{K_m}$ writes

As mentioned above, case 4 is only used if `CacheEdge` has been set, and cases 1 and 2 do not apply (i.e, it is used as an alternative to case 3). At first glance changing case 3 to 4 may appear to be a poor bargain indeed, particularly since writes are generally more expensive than reads. There are, however, several mitigating factors that make this blocking nonetheless worthwhile. If the cache is write-through, case 4 does not increase writes over case 3, so it is a clear win. Second, ATLAS also does not allow $K_m < N_B$, and in many cases $K_m \gg N_B$, so the savings are well worth having. With respect to the expense of writes, the writes are not flushed immediately; This fact has two important consequences:

1. The cache can schedule the write-back during times when the algorithm is not using the bus.

2. Writes may be written in large bursts, which significantly reduces bus traffic; this can tremendously optimize writing on some systems.

In practice, case 4 has been shown to be at least roughly as good as case 3 on all platforms. The amount of actual speedup varies widely depending on problem size and architecture. On some systems the speedup is negligible; on others it can be significant: for instance, it can make up to 20% difference on DEC 21164 based systems (which have three layers of cache). Note that this 20% improvement is merely the difference between cases 3 and 4, not between ATLAS and some naive implementation, for instance.

The analysis given above may be applied to any cache level greater than 1; it is not for level 2 caches only. However, this analysis is accurate only for the algorithm used by ATLAS in a particular section of code, so it is not possible to recur in order to perform explicit cache blocking for arbitrary levels of cache. To put this another way, ATLAS explicitly blocks for L1, and only one other higher level cache. If an architecture has 3 levels of cache, ATLAS can explicitly block for L1 and L2, or L1 and L3, but not all three.

If ATLAS performs explicit cache blocking for level L, that does not mean that level $L+1$ would be useless; depending on cache size and replacement policy, level $L+1$ may still save extra read and writes to main memory through implicit cache blocking.

3.2.2 L1 Cache-contained Matmul

The only source generator required to support the Level 3 BLAS produces a L1 cache-contained matmul. The operation supported by the kernel is still: $C \leftarrow \alpha op(A)op(B)+\beta C$, where $op(X) = X$ or X^T. C is an $M \times N$ matrix, and $op(A)$ and $op(B)$ are matrices of size $M \times K$ and $K \times N$, respectively. However, by L1 cache-contained we mean that the dimensions of its operands have been chosen such

that Level 1 cache reuse is maximized (see below for more details). Therefore, the generated code blocks for the L1 cache using the dimensions of its operand matrices (M, N, and K), which, when not in the cleanup section of the algorithm, are all known to be N_B.

In a multiply designed for L1 cache reuse, one of the input matrices is brought completely into the L1 cache, and is then reused in looping over the rows or columns of the other input matrix. The present code brings in the matrix A, and loops over the columns of B; this was an arbitrary choice, and there is no theoretical reason it would be superior to bringing in B and looping over the rows of A.

There is a common misconception that cache reuse is optimized when both input matrices, or all three matrices, fit into L1 cache. In fact, the only win in fitting all three matrices into L1 cache is that it is possible, assuming the cache is write-back, to save the cost of pushing previously used sections of C back to higher levels of memory. Often, however, the L1 cache *is* write-through, while higher levels are not. If this is the case, there is no way to minimize the write cost, so keeping all three matrices in L1 does not result in greater cache reuse.

Therefore, ignoring the write cost, maximal cache reuse for our case is achieved when all of A fits into cache, with room for at least two columns of B and 1 cache line of C. Only one column of B is actually accessed at a time in this scenario; having enough storage for two columns assures that the old column will be the least recently used data when the cache overflows, thus making certain that all of A is kept in place (this obviously assumes the cache replacement policy is least recently used).

While cache reuse can account for a great amount of the overall performance win, it is obviously not the only factor. For the L1 matmul, other relevant factors are:

- instruction cache overflow
- floating point instruction ordering
- loop overhead

- exposure of possible parallelism
- the number of outstanding cache misses the hardware can handle before execution is blocked

3.2.2.1 Instruction Cache Overflow

Instructions are cached, and it is therefore important to fit the L1 matmul's instructions into the L1 instruction cache. This means optimizations that generate massive amounts of instruction bloat (completely unrolling all three loops, for instance) cannot be employed.

3.2.2.2 Floating Point Instruction Ordering

When this paper discusses floating point instruction ordering, it will usually be in reference to *software pipelining*. Most modern architectures possess pipelined floating point units. This means that the results of an operation will not be available for use until X cycles later, where X is the number of stages in the floating point pipe (typically somewhere around 3-8). Remember that our L1 matmul is of the form $C \leftarrow A^T B + C$; individual statements would then naturally be some variant of `C[X] += A[Y] * B[Z]`. If the architecture does not possess a fused multiply/add unit, this can cause an unnecessary execution stall. The operation `register = A[Y] * B[Z]` is issued to the floating point unit, and the add cannot be started until the result of this computation is available, X cycles later. Since the add operation is not started until the multiply finishes, the floating point pipe is not utilized.

The solution is to remove this dependence by separating the multiply and add, and issuing unrelated instructions between them (requiring the loop to be skewed, since the multiply must now be issued X cycles before the add, which comes X cycles before the store). This reordering of operations can be done in hardware (out-of-order execution) or by the compiler, but this will oftentimes generate code

that is not as efficient as doing it explicitly. More importantly, not all platforms have this capability (for example, gcc on a Pentium), and in this case the performance win can be large.

3.2.2.3 Reducing Loop Overhead

The primary method of reducing loop overhead is through loop unrolling. If it is desirable to reduce loop overhead without changing the order of computations, one must unroll the loop over the dimension common to A and B (i.e., unroll the K loop). Outer loop unrolling, with its associated duplication of the inner loop results in very little overhead reduction unless it is combined with fusing the replicated innermost loops. This technique is known as unroll-and-jam [23], and it changes the memory reference pattern (and provides much greater opportunity for register blocking).

3.2.2.4 Exposing Parallelism

Many modern architectures have multiple floating point units. There are two barriers to achieving perfect parallel speedup with floating point computations in such a case. The first is a hardware limitation, and therefore out of our hands: All of the floating point units will need to access memory, and thus, for perfect parallel speedup, the memory fetch will usually also need to operate in parallel.

The second prerequisite is that the compiler recognize opportunities for parallelization, and this is amenable to software control. The fix for this is the classical one employed in such cases, namely through unrolling the M and/or N loops, and choosing the correct register allocation (using scalar replacement and scalar expansion [24]) so that parallel operations are not constrained by false dependencies.

3.2.2.5 Finding the Correct Number of Cache Misses

Any operand that is not already in a register must be fetched from memory. If that operand is not in the L1 cache, it must be fetched from further up in the memory hierarchy, possibly resulting in large delays in execution. The number of

cache misses which can be issued simultaneously without blocking execution varies between architectures. To minimize memory costs, the maximal number of cache misses should be issued each cycle, until all memory is in cache or used. In theory, one can permute the matrix multiply to ensure that this is true. In practice, this fine a level of control would be difficult to ensure (there would be problems with overflowing the instruction cache, and the generation of such a precise instruction sequence, for instance). So the method ATLAS uses to control the cache-hit ratio is the more classical one of M and N loop unrolling.

3.2.2.6 Source Generator Parameters

The source generator is heavily parameterized in order to allow for flexibility in all of the areas. In particular, the options are:

- Support for A and/or B being either standard form, or stored in transposed form
- Register blocking of "outer product" form (the most optimal form of matmul register blocking). Varying the register blocking parameters provides many different implementations of matmul. The register blocking parameters are:
  - a_r : registers used for elements of A,
  - b_r : registers used for elements of B

  Outer product register blocking then implies that $a_r \times b_r$ registers are then used to block the elements of C. Thus, if N_r is the maximal number of registers discovered during the floating point unit probe, the search needs to try all a_r and b_r that satisfy $a_r b_r + a_r + b_r \leq N_r$.
- Loop unrollings: There are three loops involved in matmul, one over each of the provided dimensions (M, N and K), each of which can have its associated unrolling factor (m_u, n_u, k_u). The M and N unrolling factors are restricted to

varying with the associated register blocking (a_r and b_r, respectively), but the K-loop may be unrolled to any depth (i.e., once a_r is selected, m_u is set as well, but k_u is an independent variable).

- Choice of floating point instruction:
  - combined multiply/add with associated scalar expansion
  - separate multiply and add instructions, with associated software pipelining and scalar expansion
- User choice of utilizing generation-time constant or run-time variables for all loop dimensions (M, N, and K; for non-cleanup copy L1 matmul, $M = N = K = N_B$). For each dimension that is known at generation, the following optimizations are made:
  - If unrolling meets or exceeds the dimension, no actual loop is generated (no need for loop if fully unrolled).
  - If unrolling is greater than one, correct cleanup can be generated without using an if (thus avoiding branching within the loop).

  Even if a given dimension is a run-time variable, generator can be told to assume particular, no, or general-case cleanup for arbitrary unrolling.
- For each operand array, the leading dimension can be a run-time variable or a generation-time constant (for example, it is known to be N_B for copied L1 matmul), with associated savings in indexing computations
- For each operand array, the leading dimension can have a stride (stride of 1 is most common, but stride of 2 can be used to support complex arithmetic).
- The generator can eliminate unnecessary arithmetic by generating code with special alpha (1, -1, and variable) and beta (0, 1, -1, and variable) cases. In

addition, there is a special case for when alpha and beta are both variables, but it is safe to divide beta by alpha (this can save multiple applications of alpha).

- Various fetch patterns for loading A and B registers

3.2.2.7 Putting It All Together – Outline of the Search Heuristic

It is obvious that with this many interacting effects, it would be difficult, if not impossible to predict a priori the best blocking factor, loop unrolling etc. ATLAS's matmul kernel search is outlined in Figure 3.4.

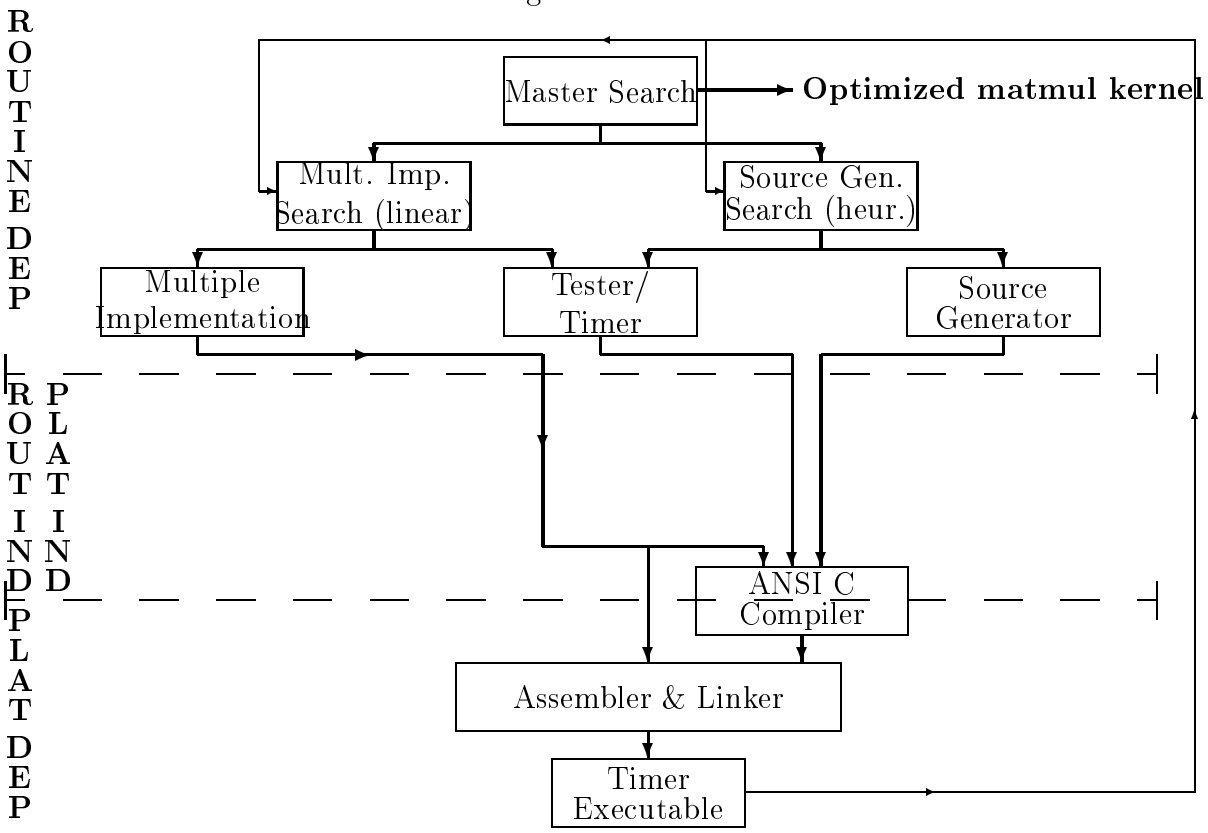


Figure 3.4: ATLAS's empirical search for the Level 3 BLAS

Our master search first calls the generator search, which uses a heuristic to probe the essentially infinite optimization space allowed by the source generator, and returns the parameters (eg., blocking factor, unrollings, etc) of the best case found. The master search then calls the multiple implementation search, which

simply times each hand-written matmul kernel in turn, returning the best. The best performing (generated, hand-tuned) kernel is then taken as our system-specific L1 cache-contained kernel.

Both multiple implementation and generator searches pass the requisite kernel through a timing step, where the kernel is linked with a AEOS-quality timer, and executed on the actual hardware. Once the search completes, the chosen kernel is then tested to ensure it is giving correct results, as a simple sanity test to catch errors in compilation or kernels.

For both searches, our approach takes in some initial information such as L1 cache size, types of instructions available, types of assembly supported, etc., to allow for an up-front winnowing of the search space. The timers are structured so that operations have a large granularity, leading to fairly repeatable results even on non-dedicated machines. All results are stored in files, so that subsequent searches will not repeat the same experiments, allowing searches to build on previously obtained data. This also means that if a search is interrupted (for instance due to a machine failure), previously run cases will not need to be re-timed. A typical install takes from 1 to 2 hours for each precision.

During installation, ATLAS runs some tests to determine what assembly dialect(s) an architecture supports. This information is then used during the multiple implementation search to avoid long error reports as contributed assembly kernels fail to compile on unsupported platforms.

The first step of the master search probes for the size of the L1 cache. This is done by performing a fixed number of memory references, while successively reducing the amount of memory addressed. The most significant gap between timings for successive memory sizes is declared to mark the L1 cache boundary. For speed, only powers of 2 are examined. This means that a 48K cache would probably be detected as a 32K cache, for instance. We have not found this problem severe enough to justify

the additional installation time it would take to remedy it. With this information, both searches have a good bound on the blocking factors to try.

Next, ATLAS probes to determine information regarding the floating point units of the platform. First ATLAS needs to understand whether the architecture possesses a combined muladd unit, or if independent multiply and add pipes are required. To do this, ATLAS generates simple register-to-register code which performs the required multiply-add using a combined muladd and separate multiply and add pipes. Both variants are tried using code which implies various pipeline lengths. ATLAS then replicates the best of these codes in such a way that increasing numbers of independent registers are required, until performance drops off sufficiently to demonstrate that the available floating point registers have been exceeded. With this data in hand, ATLAS is ready to begin actual L1 matmul timings.

Further details on the multiple implementation and generator searches are provided in the following sections. When both searches are completed, the master search designates the fastest of these two kernels (generated and hand-written) as the architecture-specific kernel for the target machine.

3.2.2.8 Source Generator Search

The general timings done by the master search provide the generator search with the L1 cache size, the kind of instructions to issue (MAC or separate multiply and add), the pipeline depth (for software pipelining and associated scalar expansion) and a rough estimate of the number of available floating point registers. This information may then be used as constraints on the search space.

The size of the L1 cache provides the search with an upper bound on the blocking factors to examine. Knowing the type of floating point instruction the underlying hardware needs cuts the cases to be searched in half, while the maximum number of registers implies what register blockings are feasible, which in turn dictates the M and/or N loop unrollings to perform. The pipeline length provides an upper bound on

the amount of software pipelining and associated scalar expansion to perform. Thus, the matmul search (and indeed many other searches) is shortened considerably by doing these general architecture probes.

In practice, K loop unrollings of 1 or K have tended to produce the best results. Thus ATLAS times only these two K loop unrolling during our initial search. This is done to reduce the length of install time. At the end of the install process, ATLAS attempts to ensure optimal K unrollings have not been missed by trying a wide range of K loop unrolling factors with the best case code generated for the unrollings factors of 1 or K.

The theoretically optimal register blocking in terms of maximizing flops/load are the near-square cases that satisfy the aforementioned equation $a_r b_r + a_r + b_r \leq N_r$ (see Section 3.2.2.6 for details). Since the ATLAS generator requires that $a_r = m_u$ and $b_r = n_u$, these M and N loop unrollings are then used to find an initial blocking factor. The initial blocking factor is found by simply using the above discussed loop unrollings, and seeing which of the blocking factors appropriate to the detected L1 cache size produce the best result.

With this initial blocking factor, which instructions set to use (muladd or separate multiply and add), and a guess as to pipeline length, the search routine loops over all M and N loop unrollings possible with the given number of registers. Once an optimal unrolling has been found, ATLAS again tries all blocking factors, and various latency and K-loop unrolling factors, and chooses the best.

3.2.2.9 Multiple Implementation Search

After the generated search is found, we perform a linear search on the available hand-tuned matmul routines. Many of these routines allow the blocking factor to be compile- or run-time constants, and so to reduce the search time, blocking factors as near as possible to the one chosen by the generator search will be used (hand-written matmul routines which take variable blocking factors are allowed to restrict the range

and multiples of the blocking factor, so in these cases we choose the blocking factor closest to that found in the generator search). When the best case is discovered, if it allows for multiple blocking factors, the entire N_B search space is checked with the specific kernel, to ensure that the hand-written code is using its best blocking factor.

3.2.3 ATLAS performance

Figure 3.5 shows the performance of double precision matrix multiply of order 500 across multiple architectures. These timings are now a couple of years old, but spot timings on various architectures has shown that the overall trend is unchanged. The matrix size of 500 is simply a midrange problem size with no particular special properties; it is not the best problem size in terms of ATLAS performance. As ATLAS is not the main focus of this dissertation, we omit more complete timing results (see [7, 8, 10] for more in-depth timings).

This graph compares performance of ATLAS, vendor, and the Fortran 77 reference BLAS. The reference BLAS are naive implementations of the standard, written in the most straightforward way possible and therefore are not optimized for any particular platform. The vendor BLAS are libraries supplied by individual hardware vendors, and can be taken to represent the apex of hand-optimization for a given platform. Not all platforms possess vendor-supplied BLAS (eg., AMD Athlon), and on these platforms ATLAS can only be compared to the reference BLAS.

The first thing to notice here is the large performance gap between the reference implementations and the tuned codes. For instance, on the Athlon platform, we see that there is currently no vendor-supplied BLAS, and that the reference BLAS run more than *fifteen* times slower than the ATLAS code. This gap may help supply an intuitive idea of the importance of optimized libraries to scientific computing.

The next point of interest is the consistency of ATLAS's performance across all of these architectures. On some platforms ATLAS is somewhat faster than the vendor, and on others ATLAS loses somewhat, but it is competitive everywhere, and in all

cases, we see order-of-magnitude speedups over code that relies completely on the compiler for optimization. This is all the more impressive when one considers that a vendor library may have a history almost as long as that of the company, while ATLAS tunes itself in only a couple of hours.

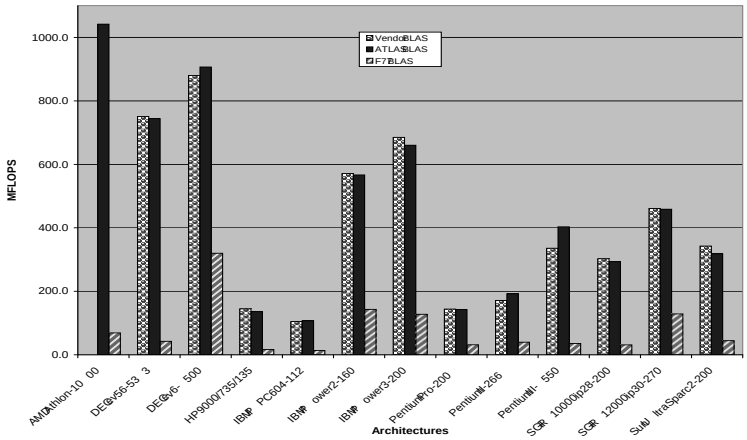


Figure 3.5: Performance of double precision matrix multiply across various architectures

3.3 AEOS Framework for the Level 1 and 2 BLAS in ATLAS

ATLAS presently uses multiple implementation (augmented by parameterization) to tune both the Level 1 and Level 2 BLAS. Therefore Figure 3.6 shows the search framework for both the Level 1 and Level 2 BLAS levels.

We give a brief overview of the details of tuning each level in turn below.

3.4 Optimizing the Level 2 BLAS

The Level 2 BLAS perform matrix-vector operations of various sorts. All routines have at most one matrix operand, and one or two vector operands. In order to concentrate on the iFKO work, space considerations rule out covering ATLAS's Level 2 BLAS implementation in any real detail. Therefore, this section will explain the theoretical underpinnings of all Level 2 optimizations: the basic memory access

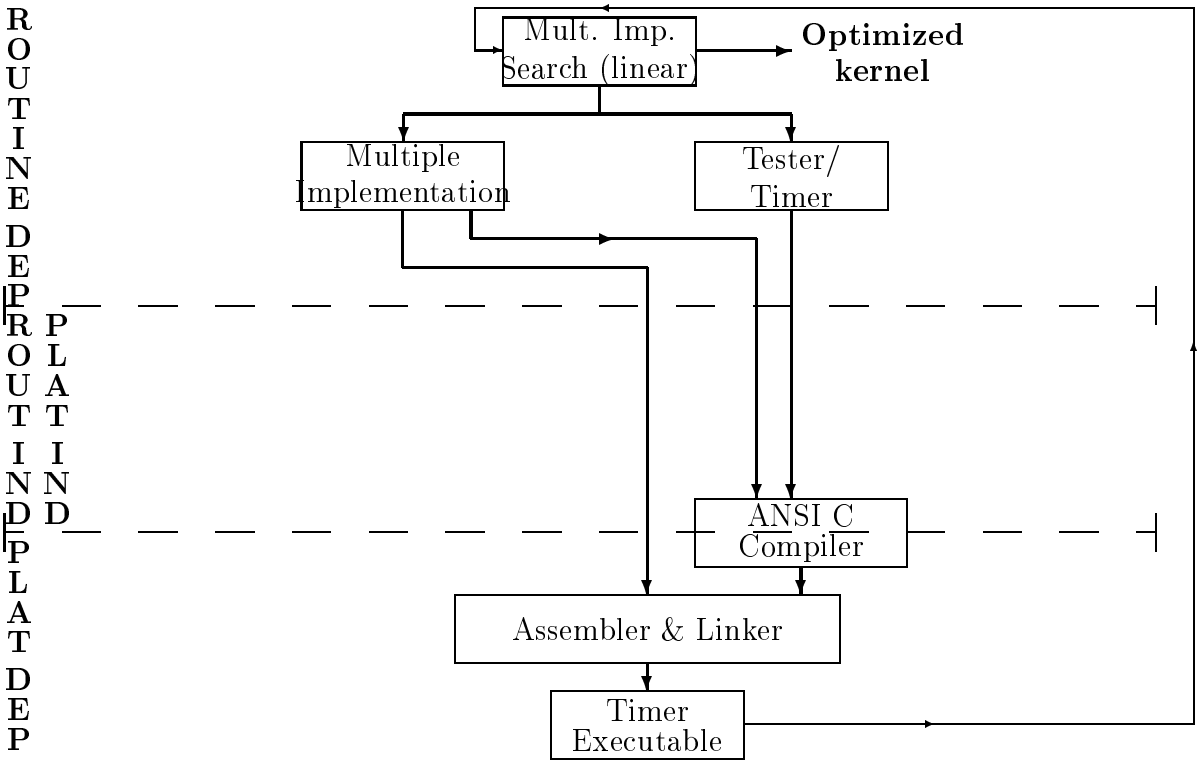


Figure 3.6: Present ATLAS empirical search for the Level 1 & 2 BLAS

techniques that allowing the vector operand(s) main memory access to be reduced from $O(N^2)$ to $O(N)$. We then describe, in the broadest possible strokes, how these and other optimizations are used by ATLAS.

3.4.1 Register and Cache Blocking for the Level 2 BLAS

If no register or cache blocking were done, the Level 2 operations would require $O(N^2)$ data access on each operand. With the appropriate register and cache blocking, the vector operands' access can be reduced to $O(N)$. Obviously enough, the $O(N^2)$ matrix access cannot be reduced, since the matrix is actually of size $O(N^2)$.

To understand this in detail, we look at the matrix vector multiply operation. In the BLAS, the matrix-vector multiply routine performs $y \leftarrow \alpha op(A)x + \beta y$, where $op(A) = A$, A^H or A^T and A has M rows and N columns. For our discussion, it is enough to examine the case $y \leftarrow Ax + y$, where A is a square matrix of size N.

This operation may be summarized as $\sum_{i=1}^{N}(y_i = \sum_{j=1}^{N} A_{ij}x_j + y_i)$. From this equation it is clear that calculating an element of y requires reading the entire N-length vector x, reading and writing the ith element of y N times, and reading the entire N length row i of the matrix A. Since there are N elements of y, it follows that this algorithm requires N^2 reads of A, N^2 reads of x, N^2 reads and N^2 writes of y. Just as with the Level 3 operations, the number of references cannot be changed without changing the definition of the operation, but by using appropriate cache and register blockings, the number of the references that must be satisfied out of main memory or higher levels of cache can be drastically reduced.

The minimum number of main memory references required to do this operation results in accessing each element from main memory only once, which reduces the accesses from ($3N^2$ reads + N^2 writes) to ($N^2 + N$ reads + N writes).

As an interesting aside, even this trivial analysis is sufficient to understand the large performance advantage enjoyed by the Level 3 over the Level 2 BLAS routines. All Level 2 BLAS require $O(N^2)$ FLOPs (Floating Point Operations); a completely optimal implementation can at best reduce the number of main memory accesses to the same order, $O(N^2)$. The Level 3 BLAS, in contrast, require $O(N^3)$ FLOPs, but can reduce the number of main memory accesses to a lower order term, $O(N^2)$. Since most modern machines have relatively slow memory when compared to their peak FLOP rate, this analysis dictates that Level 3 BLAS will achieve a much higher percentage of the peak FLOP rate than the Level 2 BLAS.

Getting back to Level 2 BLAS, we now examine the register and cache blocking, which are used in order to reduce the vector accesses.

3.4.1.1 Register Blocking

Registers are scalars which are directly accessed by the floating point unit. In a way, registers thus correspond to a "Level 0" cache. Given an infinite number of registers, only one main memory access per element would be required for all

operations. Unfortunately, the number of user-addressable floating point registers available in most ISAs typically varies between 8 and 32, and thus all but the most trivial operations will overflow the registers.

For this reason, register blocking alone can reduce either the y or x access term from $O(N^2)$, to $O(N)$, but not both. This is easily seen using the simplified GEMV operation introduced in the previous section. The basic algorithm required to reduce the accesses of y to $O(N)$ is most easily shown in the following pseudo-code:

```
do I = 1, N
   r = y(I)
   do J = 1, N
      r += A(I,J) * x(J)
   end do
end do
```

This is an "inner product" or dot product-based matrix vector multiply. If we unroll the I loop and use R_y registers to hold the elements of y, we can reduce the N^2 accesses of x to $\frac{N^2}{R_y}$, by using a register to reuse the element `x(J)` R_y times for each load.

Unrolling the loop like this essentially creates a hybrid algorithm, in the sense that the R_y y access constitute a small outer product. However, since registers cannot hold both y and x throughout the algorithm, one or the other must be flushed as the loop progresses (thus necessitating multiple loads to registers), and since we drop the value of x and maintain y in the registers, this "hybrid" algorithm is still essentially inner product.

Reducing the x component to $O(N)$ accesses requires the "outer product" or AXPY-based (AXPY being a Level 1 BLAS routine performing the operation $y \leftarrow \alpha x + y$) version of GEMV:

```
do J = 1, N
   r = x(J)
   do I = 1, N
      y(I) += A(I,J) * r
   end do
end do
```

This gives us N read accesses on x, and, just as with the inner product, unrolling the J loop and using R_x registers to hold the elements of x, we can reduce the accesses of y to $\frac{N^2}{R_x}$ reads **and** writes, by using an additional register to reuse `y(I)` R_x times.

Therefore, strictly for register blocking purposes, the inner product formulation is superior to the outer product: the total number of reads of both formulations is $O(N^2) + O(N)$, but the number of writes is $O(N)$ for inner product, but $O(N^2)$ for outer product. In practice, when array columns are stored contiguously, a heavily unrolled AXPY-based algorithm may in fact be used, since it better utilizes hardware prefetch, cache line fetch, TLB access, etc. As mentioned before, however, such details are beyond the scope of this paper, so we will assume the register blocking used will be inner product formulation.

As another practical note, the number of registers available for doing multiple AXPYs or dot products is severely limited, even beyond the 8 or 32 ISA (instruction set architecture) limit. In the inner product formulation, where R_y registers are used to form the R_y simultaneous dot products, at least two registers must be available for loading elements of x and A. Further registers will be used in order to support software pipelining and fetch scheduling. Large unrollings also mean accessing many more memory locations simultaneously, which can swamp the memory fetch capabilities of the architecture. This means that R_y is usually kept to a relatively small number (typically in the range of $2 - 8$).

In summation, register blocking reduces one vector access to $O(N)$ cost; the vector usually chosen for this reduction is the output vector (i.e., an inner product type register block), due to its higher cost. In order to reduce the remaining vector to $O(N)$, we must apply cache blocking.

While it is tempting to regard register blocking as a special case of cache blocking, their implementations are fundamentally different. As we will see, cache blocking can be easily accomplished simply by parameterizing the relevant code, so that properly blocked sections of the operands are accessed. Register blocking, as this section has demonstrated, relies on source adaptation, since varying it requires changing the loop order, number of registers, loop unrollings, etc., all of which change the code in ways that cannot be supported via simple parameterization.

3.4.1.2 Cache Blocking

As previously discussed, register blocking has reduced the access of y to $O(N)$, leaving the x access at $O(N^2)$. Therefore, loading x to registers $O(N^2)$ times cannot be avoided. However, the optimal algorithm will guarantee that main memory satisfies only $O(N)$ of these requests, leaving lower levels of cache to satisfy the rest.

Again, GEMV can be used to better understand this idea. The register block is doing R_y simultaneous dot products, so that the y access is N reads and N writes, while the x fetch to registers is $\frac{N^2}{R_y}$. Since x is reused in forming each successive dot product, x is a candidate for cache reuse. It is easily seen that forming R_y dot products accesses R_y elements of y, all N elements of x, and $R_y \times N$ elements of A. Thus the footprint in cache of one step of this algorithm is roughly $R_y + N + R_yN$.

Therefore, we can effectively guarantee L1 cache reuse by partitioning the original problem so that the footprint in cache is small enough that the relevant portion of x is not flushed between successive sets of dot products. Therefore, the correct blocking for x may be determined by solving an equation, whose simplified expression would

be: $R_y + N_p + R_y N_p = S_1 \Rightarrow N_p = \frac{S_1 - R_y}{R_y + 1}$, where S_1 is the size, in elements, of the Level 1 cache, and N_p is the partitioning of x for which we are solving.

In practice, this equation is more complicated: some memory unrelated to the algorithm will always be in cache, there will be problems associated with cache line conflicts, etc. In addition, the equation needs to be adapted to the underlying register blocking so that the initial load of the next step does not unnecessarily flush x. However, these details, while important in extracting the maximal performance, are not required for conceptual understanding, and so are omitted here.

With the correct partitioning (N_p) known, the original $N \times N$ GEMV is then blocked into $\lceil N/N_p \rceil$ separate problems of size $N \times N_p$ (the last such problem will obviously be smaller if N_p does not divide N evenly). The data access to main memory is then $\lceil N/N_p \rceil N$ reads and writes of y, N reads of X, and N^2 reads of A.

N_p is typically very close to N in size, and so this algorithm is very near optimal in its memory access. N_p will typically be in the range 350 - 1500, so even very large problems still have extremely small coefficients on the y access term. Note that any problem with $N \leq N_p$ will achieve the optimal result (N^2 access of A, N access of x and y) without any need for any cache blocking (register blocking is still required).

There is little point in explicitly blocking for higher levels of cache in the Level 2 BLAS. However, if the machine possesses a level of cache large enough to hold the footprint of the entire L1-blocked algorithm (with the previously stated simplifications, this is roughly $N_p N + N_p + R_y$), y will be reused without need for explicit blocking, and the main memory access will be reduced to its theoretical minimum.

3.4.2 ATLAS's Level 2 Compute Kernels

As we have seen, ATLAS employs one low-level compute kernel (the L1 matmul), from which the BLAS's more general GEMM routine is built. The L1 matmul and GEMM are then used in turn to generate the rest of the Level 3 BLAS. With this

method, only this one relatively simple kernel needs to be supported using source adaptation, and its performance dictates that of the entire Level 3 BLAS.

The same strategy is employed for the Level 2 BLAS, but two types of compute kernels are needed rather than one. Just as with the L1 matmul, these kernels perform register blocking and various floating point optimizations, but do no cache blocking, as it is assumed that the dimensions of the arguments have been blocked by higher level codes in order to ensure L1 cache reuse. The compute kernels for the Level 2 BLAS are:

- *L1 matvec*: An L1-contained matrix vector multiply, with four variants:
  1. No Transpose – matrix A's rows are stored in rows of input array
  2. Conjugate (complex only) – matrix A's rows are stored in conjugated form in rows of input array
  3. Transpose – matrix A's rows are stored in columns of input array
  4. Conjugate Transpose (complex only) – matrix A's rows are stored in conjugated form in columns of input array
- *L1 update1*: An L1-contained rank-1 update

Both of these kernels further supply three specialized β cases (0, 1, and variable).

3.4.3 Building ATLAS's Level 2 BLAS

This section presents a very rough outline of how ATLAS supports the Level 2 BLAS. The install of the Level 3 BLAS precedes that of the Level 2, and from this process ATLAS knows the size of the L1 cache. Thus, using a slightly more complicated version of the equations given in Section 3.4.1.2, ATLAS can obtain a good estimate of the correct Level 1 cache partitioning to use. With this in hand, ATLAS is ready to find the best compute kernels for the Level 2 BLAS.

Presently, ATLAS relies solely on multiple implementation to support these kernels (e.g. source generation is not employed). Therefore, the search simply tries each implementation in turn, and chooses the best. The conjugate forms of the L1 matvec have the same performance characteristics as their non-conjugate equivalents, so ATLAS need search only 3 differing kernels: notranspose matvec, transpose matvec, and L1 update1.

Using these best algorithms, ATLAS empirically discovers the optimum percentage of the L1 cache to use. These empirically-discovered blockings and kernel implementations are then used to build the Level 2 BLAS routines GEMV and GER (much as GEMM was built using the L1 matmul), and all of this information and these building blocks are then used to produce the rest of the Level 2 BLAS.

3.5 Optimizing the Level 1 BLAS

Unlike the Level 2 and 3 BLAS, the Level 1 BLAS, due to their simple nature, are not generally reducible to one or two simpler kernels. Therefore, each Level 1 routine must be essentially optimized individually. For some kernels, the complex case can utilize the real case, and occasionally one Level 1 routine will simplify to another due to a setting of a particular parameter, but this is the exception rather than the rule. For further details on Level 1 optimization, see [25].

3.6 Historical Context / Related Work

ATLAS was not the first project to harness empirical techniques in the interest of high performance kernels. The first such project that we are aware of was PHiPAC [1], released in December of 1995. Like early ATLAS, this project focused on using a source generator to produce varying ANSI C programs for performance tuning of matrix multiply. Due to an overly-complicated kernel, an inadequate winnowing of the search space, and insufficiently accurate timing techniques, PHiPAC

never achieved the performance and portability inherent in the AEOS concept, but nonetheless served as an inspiration for following work.

The second project, released in March 1997, to utilize this basic idea was FFTW [3, 4], which applied similar techniques to FFTs. The first version of ATLAS [6, 7, 8, 9], tuning matrix multiply only, was released in December of 1997. Subsequent versions added support for tuning all the BLAS, and later, a subset of the LAPACK [17] API as well. In 2000, the SPIRAL [26] project began utilizing empirical techniques to tune signal processing libraries.

CHAPTER 4

MOTIVATION AND DESIGN OF OUR EMPIRICAL COMPILATION FRAMEWORK – IFKO

This chapter outlines our design and approach for empirical compilation. Section 4.1 motivates and describes our approach, and Section 4.2 expounds on the design philosophy that is used to drive the research. With these guiding principles established, Section 4.3 provides an overview of the compilation framework, and Section 4.4 describes how the framework can be interfaced with ATLAS. The following chapter describes our current implementation of this design.

4.1 Motivation

This section outlines and motivates the approach we have employed in our empirical compilation research. Key features of our compilation framework, and their broad motivation, include:

1. *Our compiler is both iterative and empirical*, for all the reasons explained in the introduction.

2. *Our transformations are done in the backend, at a very low level*, allowing for the exploitation of extremely low-level architectural features such as SIMD vectorization, CISC instruction formats, special register features, as well as enabling the compiler to avoid architecture-specific resource limitations, etc., all of which can be critical in achieving extremely high performance.

3. *The search is part of the compilation framework*, rather than being managed by an external program such as a library generator. In this way, each new supported kernel necessarily increases the generality of the search, leading in the long run to a compilation framework capable of dealing with a much wider range of operations than the union of studied kernels, as happens when each set of optimization targets employs its own search.

4. *We provide for extensive user markup*, that allows a kernel writer to provide the compilation framework with information that is difficult or impossible to discover using front-end analysis (eg. aliasing information between pointers passed in as formal parameters to a library routine). This approach allows us to concentrate on the backend rather than on front-end analysis, as well as providing an opportunity to perform transformations that would be illegal without such user markup.

This approach is a direct consequence of our experience with ATLAS, and we have been careful to ensure that iFKO's design is synergistic with that of ATLAS. iFKO was designed to remove the two major limitations inherent in our ATLAS work: (1) ATLAS is operation/kernel specific, and (2) low-level architectural features (eg. SIMD vectorization, CISC instruction sets, etc.) are often not automatically exploited due to reliance on the native compiler. Generalizing the empirical optimization into a floating-point specialized compiler is a direct consequence of ATLAS's kernel specificity, and our concentration on low-level optimization arises naturally from our frustration in having to employ hand-tuning in order to fully exploit architectural features such as SSE (Intel's SIMD vectorization).

iFKO presently does most of its optimization on the innermost loop. Given the extensive list of loop transforms available in the literature, many readers may be surprised that inner-loop optimization is not fully realized by modern optimizing compilers, but our direct ATLAS experience (supplemented by hand tuning to get

around such problems) demonstrated that the majority of the lost performance opportunities when using source generation instead of hand tuning came from the inner loop, and thus iFKO's initial goal is to handle innermost loop optimization as efficiently as possible. As another example of such synergy, we can afford to put off some higher level (and outer-loop) transforms such as blocking, because the ATLAS framework does them at higher levels (eg., the blocking is not done by the kernel routine, as explained in Section 3.2). This does not mean that iFKO should never support transformations such as blocking, since iFKO is designed to be more general than ATLAS. Rather, this synergy allows FKO to (initially) target those transformations that cannot be easily handled by a framework such as ATLAS. This plan for focusing our implementation efforts is described more fully in the following design philosophy section.

One drawback of doing the transformations at a low level is that while it provides even greater *persistent* optimization, it is a barrier to *portable* optimization, as the compiler is not helpful until it has been ported to the target ISA. Again, however, ATLAS's source generator provides for *portable* optimization on truly unknown architectures, and so this drawback (due to operating at a low level, which is mandated by the required levels of performance) is ameliorated. This is discussed further in Section 4.4.

4.2 Design Philosophy

A compiler specialized for HPC kernel optimization must make the effect of each transformation, and the interaction between transformations, as optimal as possible. If the compiler cannot capture roughly the same amount of optimization from a given series of transformations as hand-tuned kernel production typically does, the HPC community is unlikely to use the compiler for its intended purpose at all. Therefore, unlike in general-purpose compilation, it is better to do a limited number

of transformations extremely well than to support many transformations that do not fully realize their potential. This is particularly true in our case, since we can rely on the ATLAS framework for many transformations that iFKO does not yet support. Therefore, our overriding focus must be "narrow and deep", rather than "broad and shallow".

This may seem counterintuitive in at least one way: one of the great strengths of empirical optimization is that it can employ an extensive array of transformations, even ones that cause significant slowdown in some instances (since only successful optimizations will be retained by the empirical search), and as the palette of supported optimizations is expanded, the generality and efficiency of the framework naturally increase. Therefore, while it is clear that a "deep" focus is mandated in order to achieve the required level of performance, in the end we must be broad as well. However, it is impossible to begin with "broad and deep", and so we must accept a narrow focus in order to demonstrate the effectiveness of this approach, and as the number of supported transformations increase, the audience for which the framework supplies a real solution grows as well.

Therefore, in each area of iFKO's design, we add features as the studied kernels demand them, allowing us to narrowly focus on each optimization study in turn. As each set of optimization targets brings in new requirements, iFKO is expanded to handle them, and thus the framework will indeed eventually be both broad and deep.

There are four general areas in iFKO that must be expanded in this way (this discussion employs terminology that is further explained in Section 4.3). These areas are: (1) the transformations supported by the optimizing compiler, FKO, (2) the analysis performed by FKO, which directs and limits the iterative search, (3) the number, type and interactions between sub-searches supported in the master search of the iterative compiler, iFKO, and (4) the type and number of user markups supported by our HIL (FKO's input language), which also serve to guide and limit the search of the optimization space.

As an example, our present implementation concentrates on inner loop transforms, and relies on ATLAS for outer-loop transformations such as blocking, but as we enlarge the target kernels to those that have not been explicitly blocked, iFKO must be expanded to support it. Further, because the x86 architecture is relatively insensitive to scheduling issues, we do not presently support software pipelining, which will clearly become critical as the framework is fully supported on architectures such as the SPARC.

Because our initial work concentrates on inner-loop transformations, we have chosen the Level 1 BLAS as our initial optimization targets (See 6.1 for further details). For these simple operations, the main markup required is identification of the loop which should be empirically optimized. On kernels with more complex dependencies, dependence markup will be added. Similarly, as operations expand to include more deeply nested loops, more complicated prefetch algorithms will need to be tried by the search, etc.

4.3 Overview of Framework

4.3.1 Anatomy of an Iterative and Empirical Compiler

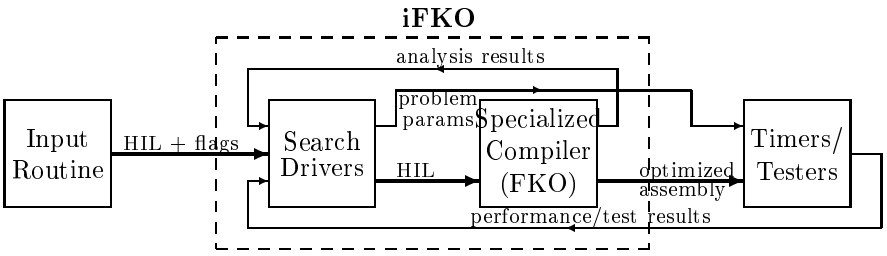


Figure 4.1: Overview of our Empirical and Iterative Compilation System

Figure 4.1 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 (though these will usually be search-controlling options, rather than the more common optimization phase options). 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 4.1 stands for high-level intermediate language, and is the language (specialized for floating point kernel optimization) which FKO accepts as input.

This graph also points out a significant overhead still associated with our iterative compiler. While the compiler makes the search and optimization kernel-independent, it depends on externally supplied timers, which are at least somewhat kernel-specific, and can be quite complex when they are written to allow for the capturing of context-sensitive usage patterns (eg., allowing a selection of cold and warm cache states, differing operand sizes and types, etc). In our case, we utilize ATLAS's preexisting AEOS-quality timers for this purpose, but an interesting area of future work would investigate the extent to which such timers could be described in a high-level way (or ultimately, even discovered through analysis of the submitted kernel), and automatically generated.

4.3.2 Optimizing compiler – FKO

The heart of this project is an optimizing compiler called FKO (Floating point Kernel Optimizer). This compiler is very similar to a traditional optimizing compiler in design, but it has been specialized in several ways. First, of course, it is designed specifically for maximizing performance of floating point kernels, which strongly affects our choice of optimizations, and their interactions, as previously discussed. This focus on kernel optimization has also led us to adopt a specialized input language, as described in Section 4.3.2.1. FKO has been further specialized for iterative and empirical use. The main way this is reflected in the design is that the compiler must be able to analyze the submitted kernel, and communicate this analysis to the master search, so the appropriate optimization techniques can be selected. The analysis presently provided by FKO is described in Section 5.3.

4.3.2.1 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 want to keep our HIL 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. Beyond this, the main interesting feature of our HIL is the ability to provide markup, which is presently quite limited. Therefore, for the sake of brevity, a full description of the input language is omitted here, but examining Appendix A, which shows the corresponding ANSI C and HIL implementations of the kernels optimized in Section 5, provides a reasonable understanding of our HIL.

We refer to our input language as a HIL (high-level intermediate language), both to stress that our focus is on the low-level backend, and because, given the success of this backend research, it seems likely that an interesting associated project would involve performing front-end analysis in order to automatically generate HIL inputs based on higher level language implementations, at least for those kernels that can be sufficiently analyzed in this way. This line of research could be extended to attempting to automatically find kernels and extract them from applications, as in the early work reported in [27].

4.3.3 Iterative Search – iFKO

In order to make our compiler iterative (adding the 'i' to FKO), we must add a search layer which attempts to find the best available optimization parameters for a given kernel. 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, because compiler writers 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). 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. At the same time, the line search has a very simple design, which in turn makes updating it to support additional transformations and explore new ideas much easier. Thus, we will utilize more advanced search techniques only once enough transformations are available to make their use compelling. Our current iterative search is outlined in Section 5.8.

4.4 Interfacing ATLAS and iFKO

As previously described, iFKO has been designed to work synergistically with (though not be limited to) ATLAS, and this can be more fully appreciated by understanding how iFKO and ATLAS can be interfaced. iFKO may be naturally added to ATLAS using ATLAS's preexisting multiple implementation support. As far as ATLAS is concerned, iFKO is simply another kernel compiler taking as input a particular language (our HIL, instead of the assembly and C kernels currently used by ATLAS). The fact that iFKO is itself iterative and empirical, affects ATLAS's own empirical search not at all (except in install time, obviously).

Figures 4.2 and 4.3 show how ATLAS's present iterative searches (as shown in Figures 3.6 and 3.4) can be augmented to interface with iFKO. Because iFKO cannot adapt to unknown ISAs, it should make sense to retain the high level (ANSI C) multiple implementation kernels for operations that ATLAS does not support through source generation. In the long run, however, iFKO should make retaining system-specific assembly kernels for ISAs where iFKO is supported unnecessary.

Because ATLAS's current Level 1 and 2 BLAS tuning uses only parameterization and multiple implementation, their support should be particularly improved by adding iFKO explicitly to the package. On the ISAs for which iFKO is ported, this automated tuning should provide much more adaptability than can be supplied through even the most extensive battery of hand-tuned implementations. On the other hand, ATLAS's Level 3 search employs both source generation and multiple implementation, and so iFKO should primarily help in reducing the need for hand-tuning in order to exploit architecture-specific features.

4.5 Related Work

Chapter 3 discusses the work closely related (both in time and topic) to our original ATLAS work. Given the demonstrated success of these packages, there has been increasing interest in the compiler community in applying similar techniques in a compiler-oriented setting. However, our approach is the first of which we are aware to perform all transformations at 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 a library generator). As discussed in Section 4.1, we believe these two factors are key in realizing the full benefits of these techniques.

The OCEANS (Optimizing Compilers for Embedded Applications) group has done some work in the area of iterative compilation. A brief declaration of effort was published in [28]. The idea is that like high-performance libraries, embedded applications are an area where very long compilation times can be successfully amortized, and so is a rich area for iterative and empirical optimization. Unlike with high performance libraries, code size is an extremely important consideration, and so differing optimization strategies should be expected. Subgroups of this extensive

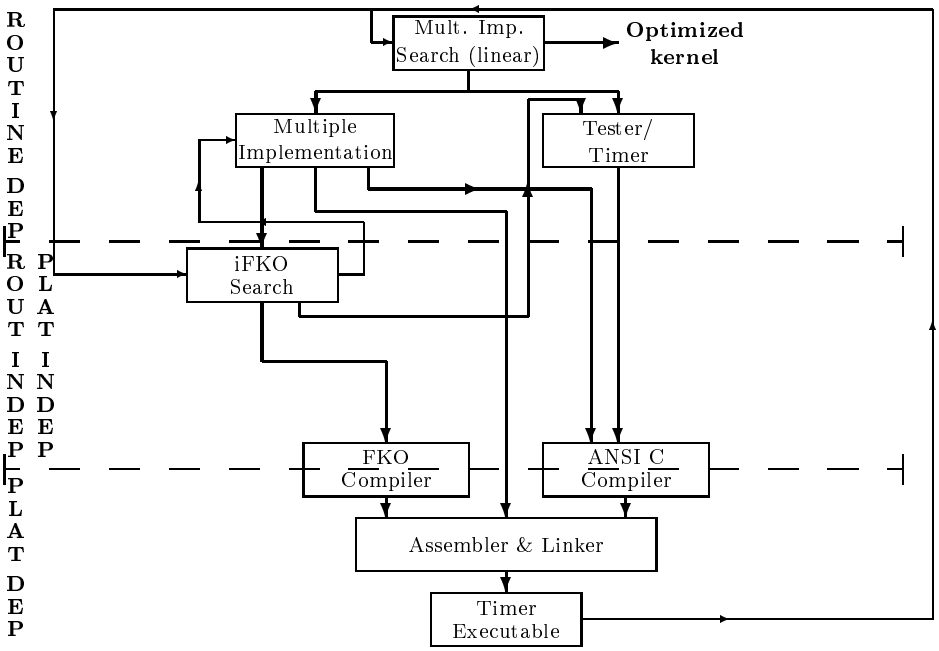


Figure 4.2: ATLAS+iFKO empirical search for the Level 1 & 2 BLAS

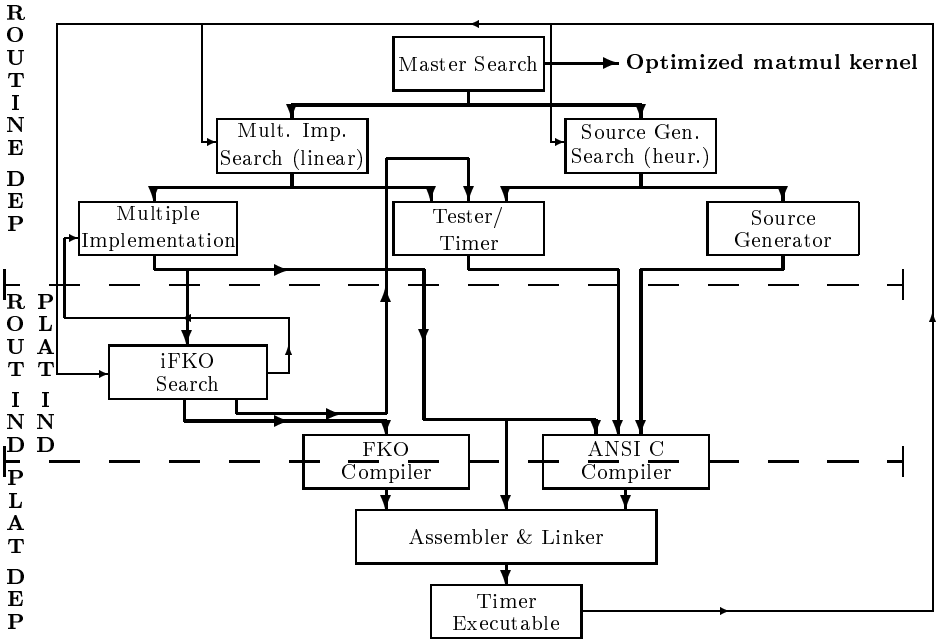


Figure 4.3: ATLAS+iFKO empirical search for the Level 3 BLAS

author list later published various iterative compilation-related papers, as outlined below.

In [29] the authors do an initial study of iterative compilation on three very restricted operations. The main compilation techniques appear to be blocking/tiling, and loop unrolling. Almost no detail is provided about the compilation system; indeed, it is never stated that the transformations were not performed by hand in some manner. The main thrust of the paper is on how to search the essentially infinite optimization space. This work is thus very far from our newer research, although it is very close to the work already done (ATLAS uses a relaxed line search to probe a more complicated space, and provides unrestricted versions of two of the three mentioned kernels). While the ultimate goal of the project is still cited as embedded systems, all results are given on general purpose hardware (again, a subset of the architectures we previously surveyed in [6, 7]).

A more fleshed out study was presented in [30]. In this work, they report a general purpose compiler, which produces FORTRAN 77 as the output language (thus allowing for architectural independence). This fact alone differentiates this work from ours, since we are concentrating on the backend and low-level architectural optimization. In contrast, the studied transformations in this paper are all high-level: blocking, unrolling, and array padding.

In [31], the authors again consider mainly loop unrolling and blocking factors; since these optimizations require little backend information, this is again done at a high level.

The work in [32, 33] is closer to our own research in that it involves both high and low level optimizations, and that they concentrate on simple computational loops. However, this work examines in detail only two optimizations (loop unrolling and software pipelining), and is concerned with embedded systems. As such, they optimize for a combination of code size and performance.

Probably the group that comes closest to our approach in gestalt is the SPIRAL project [34, 35]. They have a compiler that takes in a mathematical description of in a digital signal processing algorithm, and generates varying ANSI C or FORTRAN 77 implementations. Thus, they work mainly at the high level, and in a different field. However, authors affiliated with this group have done some low-level work (as in [36]), but this is done in a traditional compiler used to compile the generated source.

There has also been research on applying empirical techniques to general-purpose computing. In order to do so, the cost of the tuning must be greatly reduced, and thus a less empirical and much more targeted approach is called for, as in [37], where the main goal is to ameliorate the competing resource problem between optimization phases in a more traditional compilation framework. This work is targeted at general-purpose compilation, and is not intended to produce kernel-level performance.

CHAPTER 5

CURRENT IFKO IMPLEMENTATION

This chapter describes our current FKO implementation. Section 5.1 discusses our target architectures, Section 5.2 overviews the interface to FKO, and Section 5.3 outlines the analysis communicated to the iFKO's search. We next survey the transformations presently supported by FKO, which are split into two types. Section 5.4 describes FKO's *fundamental* transformations, which are applied only one time and in a known order (thus easing the extra analysis required for some of these optimizations), while Section 5.5 goes into some detail on the complexities of handling alignment issues for one of our critical fundamental optimization, SIMD vectorization. Section 5.6 outlines the *repeatable* transformations, which may be applied multiple times and in almost any order. Each transformation is outlined in its own section, but since it is the empirical application of these techniques, not the techniques themselves that are the main research element of this work, we do not go into excessive detail. In order to clarify the actual operation of these transformations, Section 5.7 shows examples of their effect on the kernels surveyed in this paper.

5.1 Supported Architectures

As described in Section 4.2, we must have a relatively narrow area of focus in order to achieve success. However, it is important that persistent framework shortcomings are not introduced due to such tunnel vision. Therefore, FKO was designed and written to support four different ISAs. Once the basic framework was implemented and working, we concentrated on the x86 in order to achieve

results, and we will probably not return to the other architectures until this work is considerably more advanced. However, since we are working at such a low level, doing the initial design with several very different ISAs helped avoid creating an inflexible or overly-specialized backend. Therefore, iFKO's basic framework is presently supported on four ISAs:

1. *IA-32* [38, 39]: Also known as x86 or x86-32, this is probably the most widely used ISA in general-purpose computing, including a diverse array of machines such as the Pentium line (PPRO, II, III, 4, 4E), the AMD Athlon, Athlon-64, Opteron, etc. Normally in this paper we use the generic term x86 to apply to both the IA-32 and x86-64 ISAs, and we will call this architecture x86-32 or IA-32 when we mean to exclude the x86-64.

2. *x86-64* [40, 41] or *IA-32e* [42]: 64 bit extension of the x86 ISA originally designed and implemented by AMD. Intel has recently begun supporting it on their line of chips, but they call it IA-32e (Intel Architecture 32 bit extended) in order to distinguish it from their x86-incompatible IA64 (Itanium line) ISA, while still avoiding using the AMD terminology. Machines implementing this ISA include AMD's Athlon-64 and Opteron, as well as the Intel's newest Pentium 4 variant. All machines implementing x86-64 run IA-32 code unchanged as well. When used in 64-bit mode, x86-64 also offers 16 integer and SSE registers, which is a vast improvement over IA-32's eight. Note that FKO explicitly supports x86-64 (i.e. it does not run on an x86-64 architecture merely through IA-32 compatibility mode), thus allowing us to exploit the additional registers, new (more efficient) calling sequence, and the fact that integer registers are 64 bits wide.

3. *PowerPC* [43, 44]: This ISA is used in embedded systems, Apple's G4 and G5 line, and IBM's workstations and supercomputers.

4. *UltraSPARC* [45, 46]: We also support FKO on the Sun UltraSPARC.

Presently, our framework can generate code for all of these architectures. However, SIMD vectorization, a key computational optimization, is presently supported only on the x86 architectures, and we have targeted our transformation selection to this ISA family. We will examine the PowerPC next, but will do so only once our goals are more fully met on the x86 ISAs.

Presently, FKO's floating point instructions always use SSE (i.e., we do not exploit the x87 FPU). When SIMD vectorization cannot be applied, we use SSE's scalar instructions. This decision was made because newer machines stress SSE at the expense of the x87 unit, and supporting the x87 register stack is a significant overhead. This does, however, mean that FKO cannot generate valid floating point code for machines prior to the Pentium III, as they do not possess vector units. We currently do not plan on adding x87 support, mostly because the unit tends to get worse performance on modern machines. For instance, on the Pentium line, even SSE scalar code has twice the theoretical peak of x87 code, and on x86-64 machines, the x87 did not receive additional registers. Therefore, x87 support will be added only if this trend of marginalizing the x87 unit is reversed in future architectures.

5.2 Interface Overview

As previously mentioned, our HIL provides a special markup that allows the user to identify the key (innermost) loop that should serve as a basis for the empirical tuning search. Let us call this special loop the *optloop*. Some FKO optimizations can only be applied to this loop (including all fundamental optimizations), while others can be applied to any section of code. FKO's present interface allows the specification of two scopes for transformation application: (1) apply to optloop only, and (2) apply to entire function (we refer to this as *global* application).

Like any compiler, FKO takes a host of flags which affect the application of optimizations. Fundamental transformations are either on or off, and if selected they are applied to the optloop, and in a known order, and so they are controlled with simple flags as in a traditional compiler. Repeatable transformations, on the other hand, may be applied repeatedly, in any order, and to any scope, and we may wish to control this from the empirical search. Therefore, while our interface presently allows specifying only global or optloop scope, the order and number of applications for repeatable phases can be more fully controlled. Repeatable transformations are specified in a grouping referred to as an optimization block, where individual phases are applied until they no longer change the code, or a maximum application count is reached (to prevent infinite loops in the case of transforms that interfere or reverse each other, or indeed to avoid any repetitive application when set to 1). Global optimization blocks are specified with the compiler flag `-G`, and optloop blocks are specified by `-L`. The arguments composing both types of optimization blocks are `blknum`, an integer label identifying the block, `maxN`, the maximum number of times to apply the indicated optimizations, `nopt`, the number of optimizations in the block, followed by the list of optimizations to apply (which may be either single optimizations or other optimization blocks). The starting `blknum` must be 1, but other block numbers may be chosen arbitrarily, as indicated by the optimization lists. A full description would probably not be useful here, but for example the flags:

```
-G 1 1 2 2 3 -L 2 10 2 ra cp -G 3 10 2 ra cp
```

result in first applying register assignment and copy propagation at most 10 times to the optloop, stopping sooner if an iteration is completed without any code changes, followed by doing the same thing to the function as a whole. Section 5.6.10 provides further examples of optimization block usage.

```
NCACHES=1
   LINESIZES : 128
OPTLOOP=1
   MaxUnroll=0
   LoopNormalForm=1
   Vectorizable=1
   Moving FP Pointers: 2
      'X': type=d prefetch=1 sets=0 uses=1
      'Y': type=d prefetch=1 sets=0 uses=1
   Scalars used in loop: 3
      'dot': type=d sets=1 uses=1 accum=1
      'y': type=d sets=1 uses=1 accum=0
      'x': type=d sets=1 uses=1 accum=0
```

(a) FKO Analysis for `ddot` (Figure A.5)

```
NCACHES=1
   LINESIZES : 128
OPTLOOP=1
   MaxUnroll=0
   LoopNormalForm=1
   Vectorizable=1
   Moving FP Pointers: 2
      'X': type=d prefetch=1 sets=0 uses=1
      'Y': type=d prefetch=1 sets=1 uses=1
   Scalars used in loop: 3
      'alpha': type=d sets=0 uses=1 accum=0
      'y': type=d sets=2 uses=2 accum=0
      'x': type=d sets=2 uses=2 accum=0
```

(b) FKO Analysis for `daxpy` (Figure A.4)

Figure 5.1: Example FKO analysis output for P4E

5.3 Current Analysis and Communication with the Search

Unlike a traditional 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 information such as the numbers of available cache levels and their line sizes. It also reports 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 floating point scalar and array accessed in the loop, the analysis further reports its type, sets and uses. Finally, the analysis returns a list of all such scalars that are valid targets for accumulator expansion (see Section 5.4), and all such 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 to prohibit prefetch on arrays known to be cache-resident, using mark-up). Figure 5.1(a) and (b) shows the results of this analysis when run on `ddot` (Figure A.5) and `daxpy` (Figure A.4), respectively.

5.4 Current Fundamental Transformations

As previously mentioned, fundamental optimizations are applied only on the opt-loop, and in a known order, and the following subsections present these fundamental optimizations 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. Fundamental optimizations are applied before other optimizations both because they require more high-level analysis than the repeatable optimizations, and because their transformations are higher level as well (SIMD vectorization, which is fairly architecture-specific, is the exception to this rule). Therefore, when examples are needed to clarify a fundamental optimization, we will normally do so in ANSI C, even though the compiler, of course, performs these transforms on our LIL (low level intermediate language).

After the optimizations are surveyed, Section 5.4.7 discusses the default values used for these parameters in the empirical search, as well as how they are controlled in FKO.

5.4.1 SIMD Vectorization (SV)

SIMD Vectorization (**SV**) transforms the loop nest (when legal) from scalar instructions to vector instructions. Vector instructions operate on multiple elements of a given type at the same time (thus the 'Single Instruction Multiple Data' of SIMD vectorization). Applying **SV** typically results in keeping the number of instructions in the loop constant, 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). This extremely modest vector length is the primary distinguishing factor between SIMD vectorization and vectorization for traditional vector machines, but this difference is of such magnitude that significantly different optimization strategies are required.

Both the x86 and PowerPC architectures have SIMD vector units which are supported through ISA extensions. On both machines, the vector length is 128 bits. The PowerPC vector unit (AltiVec unit), however, cannot operate on double precision floats, and so can operate on four single precision floats or a varying number of integral values at a time. FKO presently supports SIMD vectorization only on the x86, and so the following discussion focuses on it exclusively.

The x86 vector units are utilized through a series of ISA extensions including MMX (MultiMedia eXtensions), 3DNow!, SSE (Streaming SIMD Extensions), SSE2, and SSE3. MMX deals primarily with operating on integer vectors (unimportant to our present discussion), while 3DNow! is an AMD specification that has been largely superseded by SSE, and so we will not describe it here. SSE, or SSE1, mainly added support for operating on single precision vectors, SSE2 added double precision vector support, and SSE3 includes some extra vector permutation code, and instructions specialized for complex arithmetic.

As mentioned, the vector length for all current machines is 128 bits, which allows for parallel operation on four single precision values, or two double precision values. Thus we say the vector length (*veclen*) is two for double precision, and four for single. All current machines can employ these instructions to do *veclen* floating point operations (FLOPs)/cycle. Therefore, the theoretical peak will be four FLOPs/cycle for single precision, and two FLOPs/cycle for double.

Vector instructions are available not only for computation, but for loads and stores as well. Loads and stores must be 128-bit aligned to get maximal performance, and special workarounds must be employed when the data is not so aligned. Optimally handling alignment issues is a complicated issue in its own right, and is therefore discussed separately in Section 5.5.

Some substantial analysis is required to ensure that **SV** is legal, and so it is the optimization we apply first. Performing **SV** before any other optimization allows us to assume the input code is in the restricted format generated by our front-end,

which considerably eases the burden of analysis. The transformation is either applied to the loop as a whole, or it is not applied at all (i.e. we do not mix floating point vector and scalar code in the loop). Our current implementation therefore requires:

1. All arrays accessed in the loop are 128 bit aligned,
2. All arrays being vectorized are accessed contiguously in successive iterations of the loop,
3. The dependence distance between successive elements of such arrays is $> veclen$,
4. All floating point computation in the loop is of the same precision, and consists of solely of a mixture of absolute value, add and multiply,
5. Scalars applied to vectorized arrays must meet the criteria given in Section 5.4.1.1.

All of these requirements except alignment are determined by analysis (see Section 5.5 for details on alignment), and SIMD vectorization is not performed when the loop does not meet these criteria. Note that this "all-or-nothing" approach has been chosen as it is well-matched to the kinds of operations we are interested in (except for a few Level 1 BLAS, all of the BLAS may be fully vectorized in this way). As we expand the supported kernels to less regular and/or non-contiguous operations, we may want to employ a more general strategy, where more arbitrary vectorization opportunities are searched for after a series of optimizations (including loop unrolling), as in [36]. However, this style of vectorization, while more flexible in application, is also more fragile, in that other optimizations can make it more and more difficult to fully vectorize the loop. Therefore, if we are unable to capture the same functionality as a fundamental optimization, this style of vectorization would be added as an additional repeatable transformation when and if it is needed.

Since vectorizing the loop is computationally similar to unrolling by *veclen*, this transformation also produces a cleanup loop (dumped to the end of the function) which handles those cases where the number of iterations of the loop are not even multiples of *veclen*. Let N be the number of times the loop will iterate (N is almost always a run-time variable). SIMD vectorization adds a conditional branch on N both before (to handle $N < veclen$) and after the vectorized loop (to handle N mod $veclen \neq 0$). Section 5.4.2 shows an example of creating such cleanup code for an unrolling of four, and Section 5.7.1.1 shows vectorization at a lower level.

5.4.1.1 Handling Scalars in SV

SV is applied primarily to arrays whose access changes with the loop iteration, but the loop body will almost always include references to scalar (single value) temporaries that are involved in the computations on these arrays. These scalar values must be changed to vector values by the compiler. Essentially, any temporary or variable whose address does not move with the loop is treated as a scalar for this purpose. Scalars that are live on loop entry or exit must be transformed from scalar to vector on loop entry, and from vector to scalar on exit, and how this is legally done depends on usage, as outlined below.

Input scalars whose first use is assignment or multiplication should have all vector values initialized to the scalar value, whereas scalars used as a target of vector adds (accumulators) should have only one value set to the scalar value, and the rest of the vector should be set to zero.

Output scalars reverse this process. Because it is always the case in our kernels, our present implementation only vectorizes code where any output scalars are accumulators. If their last use was an accumulator, the individual vector values must be summed after the loop to produce the required result.

On loops with multiple basic blocks, it is possible to have mixed first (last) usage in differing blocks that represent different paths of flow through the loop. If this

mixed usage occurs (eg., a variable's first use in one path is assignment, but first use elsewhere is as an accumulator), then vectorization will not be applied. There are no cases in the present kernels, and we know of none in all of the BLAS, where this problem prevents vectorization. However, if mixed usage becomes a problem, transformations such as scalar expansion and loop unswitching [24] can be employed to enable **SV**.

5.4.2 Loop Unrolling (UR)

Loop Unrolling [24] (**UR**) duplicates the loop body of the loop N_u times. Since it is performed after SIMD vectorization the computational unrolling is actually $N_u * veclen$ when vectorization is also applied.

If it is possible to do so, our unrolling avoids repetitive index and pointer updates, as well as having only one test/branch for the unrolled loop. Just as with vectorization, a cleanup loop is automatically generated to handle when the iteration count is not a multiple of the unroll factor, and conditional branches are inserted so that the correct answer is always produced regardless of the iteration count. As a minor optimization, if a loop is both vectorized and then unrolled, only one cleanup loop is generated and used.

Figure 5.2 shows a simple dot product loop before and after unrolling to 4. Note that the loop control optimization discussed in the following section is always applied during unrolling as well, resulting in loop index reversal when the loop index is not referenced in the loop.

5.4.3 Optimize Loop Control (LC)

Optimize Loop Control (**LC**) is the only fundamental optimization that is always applied when legal, and it attempts to optimize the loop branching and index computation when possible. On the x86, this primarily consist of transforming the loop from the form `for(i=0; i<N; i++)` to the equivalent `for(i=N; i; i--)`

```
for (i=0; i < N; i++) {
   dot += X[0] * Y[0];
   X++;
   Y++;
}
return(dot);
```

(a) before loop unrolling

```
if (N < 4) {
   i = N;
   goto CU;
}
for (i=N-3; i > 0; i -= 4) {
   dot += X[0] * Y[0];
   dot += X[1] * Y[1];
   dot += X[2] * Y[2];
   dot += X[3] * Y[3];
   X += 4;  Y += 4;
}
i += 3;
if (i != 0) goto CU;
return(dot);
CU:
   for (; i; i--) {
      dot += X[0] * Y[0];
      X++;  Y++;
   }
   return(dot)
```

(b) after loop unrolling

Figure 5.2: Dot product before and after UR and LC

when `i`'s only use in the loop is for loop control. This second formulation avoids a comparison required by the first on the x86 (and indeed, most architectures), as well as enabling the compiler to avoid assigning `N` to a register throughout the body of the loop. This transformation also handles the index computations necessary to correctly handle loop unrolling and vectorization efficiently.

5.4.4 Accumulator Expansion (AE)

Accumulator Expansion (**AE**): In order to avoid unnecessary pipeline stalls, **AE** uses a specialized version of scalar expansion [24] to break dependencies in scalars that are exclusively the targets of floating point adds within the loop. Figure 5.3(a) shows a dot product loop that has been unrolled by a factor of 2. If the FPU is pipelined, and the pipe length is greater than one, this code will result in an unneeded pipeline

```
1   dot = start;
2   for (i=0; i < N; i += 2) {
3      dot += X[0] * Y[0];
4      dot += X[1] * Y[1];
5      X += 2;  Y += 2;
6   }
```

(a) Without AE

```
1   dot = start; dot1 = 0.0;
2   for (i=0; i < N; i += 2) {
3      dot  += X[0] * Y[0];
4      dot1 += X[1] * Y[1];
5      X += 2;  Y += 2;
6   }
7   dot += dot1;
```

(b) With AE=2

Figure 5.3: DDOT before and after Accumulator Expansion

stall. After line 3 is executed, the register holding `dot` will not be available to add into as required by line 4 for pipeline length cycles, and so line 4 will cause a delay each time through the loop. Accumulator expansion removes this dependency by using multiple scalars to store the accumulator, as shown in Figure 5.3(b). Note that for a machine with a FPU pipeline length of four, for example, we would probably want to unroll to at least that length, and use four registers, rather than the two shown in this simple example.

In this example, we have shown `dot` being initialized to a `start` value. Notice that the additional accumulators generated by **AE** must be set to zero (line 1 of Figure 5.3(b)), and summed into `dot` after the loop is complete (line 7 of Figure 5.3(b)).

5.4.5 Prefetch (PF)

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 memory 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.

5.4.6 Non-temporal Writes (WNT)

Our final fundamental transformation is *non-temporal writes* (**WNT**), which employs non-temporal write instructions on the specified output array. Non-temporal writes are designed to be useful when the value written will not be accessed again soon, but its implementation varies strongly by architecture (for instance, on the Opteron **WNT** is only useful for write-only arrays, but it is useful for any output array that does not need to be retained in the cache on the P4E).

5.4.7 Default Values

There are two types of "default" values for these optimizations. One is which fundamental transformations are automatically applied by FKO without special flags, and only **LC** is handled in this way. All other fundamental transformations are applied only when the requisite flag is passed to FKO. The other "default" of interest is what values iFKO employs during the empirical search, and these defaults are as follows: Let L be the line size of the first prefetchable cache, and L_e 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 iFKO's search are: **SV**='Yes' (if legal), **WNT**='No', **PF**(type,dist) = ('prefetchnta', $2 \times L$), **UR**=L_e, **AE**='None'.

5.5 SIMD Alignment Issues

As previously mentioned, vector loads and stores are by default assumed to be 128-bit aligned. The current FKO implementation assumes such alignment whenever

SV is applied. FKO already possesses all the transformations necessary to handle non-aligned access *safely*, but we currently assume alignment because we do not have the required infrastructure to handle these cases *efficiently*. Since we can leverage higher level routines to ensure that these alignment requirements are met, we will add non-aligned considerations explicitly to FKO only once we have expanded the framework so that they may be handled efficiently as well as safely.

As discussed later, producing a highly-tuned non-aligned code takes explicit tuning for the non-aligned case (i.e., the tuning decisions made for the aligned code will not, in general, be valid for the non-aligned), and so we first concentrate on tuning the aligned cases, letting ATLAS handle getting them so aligned. The following sections discuss these alignment issues in further detail. First, Section 5.5.1 discusses how we use ATLAS to guarantee alignment so that we can assume correct alignment in kernels generated by FKO. After this overview, we discuss the actual transformations that can be employed (both by ATLAS presently, and FKO ultimately) to correctly handle these cases. Section 5.5.2 discusses a safe method that works for all cases, but results in inefficient code. Section 5.5.3 provides an overview of how loop peeling can be used to force alignments in those cases where all relevant memory addresses have the same alignment. Section 5.5.4 then describes how this can be extended to handle mutual misalignment, and Section 5.5.5 discusses some possible refinements that can be applied when the pointers in question arise from a constantly strided multidimensional array (a very common case). Finally, Section 5.5.6 describes how we envision adding this complicated support to the framework.

5.5.1 Present Handling of Alignment

In our present use of FKO, we exploit ATLAS's framework to guarantee alignment. All of ATLAS's timers and testers optionally accept flags that tell them to force particular alignments, and so we encounter no problems during tuning. When it is time to use the FKO-generated kernels to build an ATLAS-tuned library, we

use multiple implementation to add the routine to ATLAS. At this stage, we write wrapper code by hand that guarantees particular alignment(s), using the techniques discussed in the following sections. Also, the ATLAS framework *already* guarantees alignment for the Level 3 BLAS kernels, and so we know we don't need to handle these cases for some of our more important targets.

5.5.2 Handling Alignment Safely, but Inefficiently

Since vectorization has an implicit computational unrolling, a scalar cleanup loop (which has no vector alignment requirements) is always generated. Trivially, we could add a branch to this cleanup anytime our operands are not appropriately aligned. While this would mean that the code would handle all cases correctly, the non-aligned case would not only be scalar code, but largely untuned scalar code at that. It is for this reason that we do not use this mechanism.

5.5.3 Fixing Some Alignment Problems through Loop Peeling

A subset of alignment problems may be addressed through simple loop peeling [24]. Again, we have most of the infrastructure needed for this, in that peeling can reuse with very little modification the unrolling and cleanup infrastructures.

In peeling for alignment, iterations of the loop are peeled and conditionally executed based on alignment, but the iterations that are peeled are *scalar* iterations, so that we do the appropriate number of scalar iterations until the relevant pointers are 128-bit aligned, and then we enter the vectorized loop.

This method fully solves the problem if and only if all vectorized pointers have the same alignment. If two or more vectors are mutually misaligned (eg, $P_0 \mod 128 = 32$ and $P_1 \mod 128 = 64$), then at least one pointer is still misaligned, and more complicated techniques are required, as described in the following sections.

5.5.4 Handling Mutual Misalignment

If two or more pointers are mutually misaligned, the general solution is to force the alignment of one of the pointers via peeling as before, and then generate code that assumes the given array/pointer is aligned, and any others are not. Obviously, if one array is accessed more than the others (for instance, if one array both used and set while other arrays are use only), then that array is the obvious target for peeling for alignment.

To make this more concrete, assume we are accessing two single precision arrays, X (read only) and Y (both read and written), and that $X \mod 128 = 32$ and $Y \mod 128 = 64$. In this case we use loop peeling to force Y to be aligned to 128 (which in this example would result in doing two scalar iterations of the loop before entering the vectorized loop). With Y forced to the correct alignment, we would now empirically tune a sub-kernel specialized for aligned Y and non-aligned X.

There are at least two general techniques to try in these cases. On the x86, there are non-aligned vector loads, which are slower than the aligned loads. Therefore, the first technique would involve using non-aligned loads on X, but then it makes sense to do some scheduling to reflect the fact that X access is slower than Y access. For instance, we might want to software pipeline all X accesses so that this iteration fetches the next iteration's X data.

Another, even more aggressive, technique is to instantiate the various possibilities of X misalignment, and then use aligned access on X, but permute the data before applying it to Y. This in fact is required for the PowerPC, where the AltiVec unit does not support non-aligned load. Again, if we are required to permute one array's data before use, we will probably want to software pipeline it so that waiting on the permute does not create a bottleneck.

As can be seen, handling the non-aligned case represents a more significant tuning problem than **SV** itself, and this is the main reason we have not yet handled it efficiently in FKO.

5.5.5 Special Alignment Considerations for Constantly Strided Multidimensional Arrays

Constantly strided multidimensional arrays are common in floating point kernels. For instance, the Level 2 and 3 BLAS operate on constantly strided two-dimensional arrays. The non-unit stride of these arrays, in elements of the native type, is called the lda (Leading Dimension of Array). When outer loop unrolling is performed on such arrays, the inner loop then gets multiple pointers, which may be mutually misaligned, depending on lda. To make this more concrete, consider a double precision column-major array with $lda = 7$. Since there are two double precision elements in a SIMD vector, all contiguous columns will be mutually misaligned by 64 bits. However, every second column will have the same alignment.

More generally, if there are N_e native elements in a SIMD vector, a particular column j is guaranteed to have the same alignment as column $j+N_e$, regardless of lda. Further, if $(lda \times sizeof()) \mod 128 = 0$, then all columns have the same alignment. NOTE: swap the word 'row' for 'column' in the above analysis for row-major arrays.

We can use these insights to reduce alignment complexities when unrolling outer loops, or more generally, anytime we can discover that inner-loop pointers come from constantly strided arrays.

5.5.6 Adding Misalignment Support to the Framework

As this discussion has made clear, well-tuned misalignment support is a complicated issue. Support will undoubtedly be phased in as we develop iFKO. For instance, the peeling and safety cases discussed in Sections 5.5.3 and 5.5.2 will probably

be added soon, since we already have the needed infrastructure, and peeling, in particular, does completely solve the problem for kernels that access only one array.

The next step will require significant overhead, and will probably only be undertaken when we have some actual usage that require it. In this step, we would add explicit tuning for non-aligned cases, which requires extensions throughout the framework, including the search, the transformations supported, and instructions available in code generation.

Once we are able to explicitly tune for misalignment, we will still initially use multiple implementation to choose when to call particular versions (each tuned in separate iFKO invocations). Once sufficient performance and generality is available in this tuning, we can automate the search of all misalignment subcases and the generation of the wrapper code as well.

5.6 Current 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). FKO's present interface allows the user to specify that these optimizations be applied to the optloop, or globally to the entire function. As we will see in Section 5.6.10, most of these transformations are by default first exhaustively applied to the optloop, and then applied globally as well.

We presently support repeatable transformations for improving register usage and control flow. In register usage optimization, we support register allocation (Section 5.6.1) and various types of copy propagation (Sections 5.6.2 and 5.6.3). Several of our register usage improving transformations (Sections 5.6.7, 5.6.8 and 5.6.9) are in fact 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 (Section 5.6.6), useless jump elimination (Section 5.6.4), and useless label elimination (Section 5.6.5), which, when applied together, merge basic blocks (critical after extensive loop unrolling).

5.6.1 Register Allocation (ra)

This is our most complex repeatable transformation. It performs interference graph based register allocation on an relatively arbitrary scope (a scope being a list of basic blocks). This optimization is applied both within and across basic blocks. We assume that the scope has a single preheader (a single predecessor block that must be passed through in order to reach all blocks in the scope), but allow for multiple successors to the scope, which we call post-tails. We use loop terms such as header and tail because our most important scopes are indeed loop bodies.

If a variable is live on scope entry, then the register load of that variable is hoisted to the preheader of the scope, and if a variable is live on scope exit, and a store is required, then the register-to-memory store is pushed into the relevant post-tail(s). The most common scope is a loop body, and when applied in this way to a variable live throughout the loop, all associated memory accesses will be hoisted/pushed out of the loop.

Presently, FKO by default applies **ra** to the loop identified for empirical tuning, and then to the entire function. As we consider more deeply nested loops, we will apply **ra** to each loop level in turn, starting from the innermost. Performing **ra** on the innermost loop first ensures that the inner loop's register needs are completely met before outer loops are considered, which is critical in floating point kernels, where long-running loops are the common case. Until registers are exhausted, **ra** and copy propagation applied to outer loops will continue to expand the live range to the maximum extent, and eventually moving the load (store) to beginning (end)

of the function, if possible. Thus, this rather complicated version of **ra** provides an efficient algorithm for register spilling in the case of register exhaustion, as well as allowing us to postpone implementation of a more generalized loop-invariant code motion.

5.6.2 Copy Propagation (cp)

Copy propagation [24] is a technique for removing unnecessary register-to-register moves, often generated by preceding optimization phases, such as register allocation. Our implementation operates both inter- and intra-block, and performs several related transformations. Trivially, it deletes any such move where the source and destination are the same register.

In it's main use, our copy propagation phase proceeds through the scope in forward order, looking for register-to-register moves. If the source register is dead at this point, we delete the register-to-register move, and replace all succeeding references of the destination register with the source register, until either the source becomes live again, or the destination is dead. If the source is still live, we do the same, but stop the propagation if the destination register is set. When we are forced to stop the propagation before the destination register's live range is complete, we put the register-to-register move back into the code, but as far down in the scope as possible (hopefully out of a critical path, for instance). If copy propagation must be halted on the next instruction after the move, no change is made.

5.6.3 Reverse Copy Propagation (rc)

Our current reverse copy propagation (**rc**) operates only within a basic block. We look for register-to-register moves where the source register is dead, but starting at the bottom of the block and proceeding towards the beginning. When we find such a move, we find the initialization of the source's live range, and if it is in this

block, we delete the move and change all references to the source register between that initialization and the move to the destination register.

Thus, this transformation is designed to complement **cp**, in that **cp** attempts to maximize live ranges (and thus minimize moves) by finding moves and extending the source's live range forward in the code, while this optimization instead extends the destination's live range backwards. When applied together, this can remove obstacles to copy propagation caused by register reuse, which would otherwise require register renaming.

5.6.4 Useless Jump Elimination (uj)

This transformations removes any unconditional jumps to blocks that are positioned directly after the jump.

5.6.5 Useless Label Elimination (ul)

Removes local labels that are either not referenced in the routine, or have no executable statements between them and another label. In this latter case, all references to the removed label are replaced by the retained label. This has the effect of removing empty basic blocks when possible.

5.6.6 Branch Chaining (bc)

Replaces branches to unconditional jumps (or a chain of such jumps) with a branch to the final target.

5.6.7 Enforce Load Store (ls)

The x86 is not a true load/store ISA, and thus many of its non-load instructions allow memory addresses as sources. This transformation removes any such in-memory usage, replacing them with the more standard load to register, followed by register

use. This is useful in exposing the possibility of register reuse (with correlated hoisting/pushing) to the other optimization phases. An example of how this can be useful is given in Section 5.7.2.

5.6.8 Remove One Use Loads (u1)

This is an x86-specific peephole optimization which searches for loads to a register whose live range is complete on the first use (it cannot be applied to sets, as the x86 ISA does not provide non-store instructions that accept destination operands that address memory). When an in-memory version of the instruction exists, **u1** then deletes the explicit load, and changes the use to an in-memory version of the instruction. This reduces register pressure, and is therefore almost always worth applying on these systems.

5.6.9 Last Use Load Removal (lu)

Like **u1**, this is an x86-specific peephole optimization employed to reduce register pressure. As discussed, **lu** replaces single-use instructions with their in-memory equivalents. For registers that are accessed multiple times, we can sometimes avoid an unnecessary load by changing the last use of the register to an in-memory instruction. This is done by changing the order of the instruction, so that a multiple-use register is overwritten on its last use (i.e., we change the use of that register to a set, and the register now contains a live range that was originally in another register). Because we swap the source and destination, the instruction that we are changing to in-memory must be commutative, or we cannot apply **lu**. This is probably the hardest of all the repeatable optimizations to explain, and so a review of the actual example of its application given in Section 5.7.3 may be needed for more complete understanding.

```
-L 1 0 4 ls 2 3 4
-G 2 10 3 bc uj ul
-L 3 10 5 ra cp rc u1 lu
-G 4 10 5 ra cp rc u1 lu
```

(a) as command-line args

Local **ls**	while (CHANGES)
while (CHANGES)	{
{	Global **ra**
Global **bc**	Global **cp**
Global **uj**	Global **rc**
Global **ul**	Global **u1**
}	Global **lu**
while (CHANGES)	}
{	
Loop **ra**	
Loop **cp**	
Loop **rc**	
Loop **u1**	
Loop **lu**	
}	

(b) as pseudocode

Figure 5.4: Repeatable optimization defaults

5.6.10 Default Values

Presently, the iterative search does not vary the repeatable optimizations, and always uses the defaults (applied *after* any fundamental optimizations), which are summarized in Figure 5.4.

5.7 FKO in Action

In this section we use actual code generated by FKO to illustrate how some of these transformations work in practice. Section 5.7.1 uses `ddot` (Figure A.5) to demonstrate register allocation, copy propagation, one use load elimination, and, most importantly, SIMD vectorization. Section 5.7.2 then uses `dasum` (Figure A.3) to illustrate prefetch, enforce load/store, and the linked transformations of loop unrolling and useless label elimination. Finally, Section 5.7.3 employs `daxpy` (Figure A.4) to explore the use of non-temporal writes and last use load removal.

Because this is actual code generated by FKO, it is in IA-32 assembly, which is not as accessible as the ANSI C examples shown previously. However, many of the lower-level optimizations (eg., SIMD vectorization, register allocation, copy

propagation, etc.) are difficult to illustrate in high level languages. Comments are provided that explain the general effects of lines of interest, and we provide a few notes here as an aid for those interested in examining these examples in greater detail.

Assembly for the x86 is two operand (in most cases; it is, after all, CISC), and in the gnu dialect generated by FKO, the last operand is always the destination. As previously mentioned, FKO uses SSE for both scalar and vector instructions. All examples use double precision data, and double precision SSE instructions end in the suffix `d`. Scalar SSE instructions' second to last character is `s`, and vector instructions (which operate in *p*arallel on multiple scalars) have this character set to `p`. Thus, [`movsd`, `mulsd`, `addsd`] are SSE opcodes to perform scalar double precision move, multiply and add, respectively, and their vector equivalents are [`movpd`, `mulpd`, `addpd`]. Integer instructions are typically suffixed by `l` (for long, as this ISA has its roots in 8-bit operation). Constants are prefixed by $, registers by %, and the SSE registers are `xmm0` through `xmm7`, while the stack pointer is `esp`.

5.7.1 DDOT Example Illustrating ra, cp, rc, u1, and SV

In this section, we use the Level 1 BLAS kernel `ddot` to illustrate register allocation/assignment (**ra**), copy propagation (**cp**), reverse copy propagation (**rc**), remove one use loads (**u1**), and SIMD vectorization (**SV**). The `ddot` kernel performs a vector product operation, and its HIL loop is shown in Figure 5.5(a).

Figure 5.5(b) shows the same region of code in assembly, when generated by FKO without any optimization other than optimize loop control, which is always applied when legal. For this architecture, reversing the loop allows us to avoid a comparison instruction, and so we see in lines 26-30 that the loop given in our HIL, which was of the form `for(i=0; i < N; i++)`, has been transformed to one of the form `for(i=N; i; i--)`.

Note that our front-end generates simplistic load/store versions of each operation: for any computation, all operands are loaded from memory, then the computation is

```
1   LOOP i = 0, N
2   LOOP_BODY
3       x = X[0];
4       y = Y[0];
5       dot += x * y;
6       X += 1;
7       Y += 1;
8   LOOP_END
9   RETURN dot;
```

(a) Relevant HIL Loop

```
1 .local  _LOOP_0
2 _LOOP_0:
3#    x = X[0];
4     movl    36(%esp),%edx
5     movsd   (%edx),%xmm0
6     movlpd  %xmm0,16(%esp)
7#    y = Y[0]
8     movl    32(%esp),%edx
9     movsd   (%edx),%xmm0
10    movlpd  %xmm0,8(%esp)
11#   dot += x * y;
12    movsd   16(%esp),%xmm0
13    movsd   8(%esp),%xmm1
14    mulsd   %xmm1,%xmm0
15    movsd   (%esp),%xmm2
16    addsd   %xmm0,%xmm2
17    movlpd  %xmm2,(%esp)
18#   X += 1
19    movl    36(%esp),%edx
20    addl    $8, %edx
21    movl    %edx,36(%esp)
22#   Y += 1
23    movl    32(%esp),%edx
24    addl    $8, %edx
25    movl    %edx,32(%esp)
26#   while(--i);
27    movl    44(%esp),%edx
28    subl    $1, %edx
29    movl    %edx,44(%esp)
30    jne     _LOOP_0
31 .local  _LOOP_END_0
32 _LOOP_END_0:
```

(b) Assembly, no optimization other than LC

```
1#    Hoisted loads from ra
2     movl    48(%esp),%ebp
3     movsd   4(%esp),%xmm2
4     movl    36(%esp),%eax
5     movl    40(%esp),%ecx
6 .local  _LOOP_0
7 _LOOP_0:
8#    x = X[0];
9     movl    %ecx,%edx
10    movsd   (%edx),%xmm0
11    movsd   %xmm0,%xmm1
12#   y = Y[0]
13    movl    %eax,%edx
14    movsd   (%edx),%xmm0
15    movsd   %xmm0,%xmm3
16#   dot += x * y;
17    movsd   %xmm1,%xmm0
18    movsd   %xmm3,%xmm1
19    mulsd   %xmm1,%xmm0
20    movsd   %xmm2,%xmm2
21    addsd   %xmm0,%xmm2
22    movsd   %xmm2,%xmm2
23#   X += 1
24    movl    %ecx,%edx
25    addl    $8, %edx
26    movl    %edx,%ecx
27#   Y += 1
28    movl    %eax,%edx
29    addl    $8, %edx
30    movl    %edx,%eax
31#   while(--i);
32    movl    %ebp,%edx
33    subl    $1, %edx
34    movl    %edx,%ebp
35    jne     _LOOP_0
36 .local  _LOOP_END_0
37 _LOOP_END_0:
38#   Pushed store from ra
39    movlpd  %xmm2,4(%esp)
```

(c) Assembly, after ra

Figure 5.5: DDOT Loop in HIL and Assembly with no optimization, and ra

performed, and then result is stored back to memory. Lines 3-17 are all involved in performing a simple `dot += X[0]*Y[0]`, for instance. At this point, registers are live only across a single computation, and so there is no reuse. FKO relies on repeatable optimizations to improve register usage, as the shown in the following examples.

Figure 5.5(c) shows the same assembly, but this time we have applied register assignment/allocation. In lines 2-5 we see that the loads of the variables `i` (the loop index), `dot`, `X` (pointer to first vector), and `Y` (pointer to second vector) have been hoisted out of the loop. Further, since `dot` is live on output and written in the loop, the store of `dot` back to memory has been pushed out of the loop, to line 39. At this point, the size of the code supporting the loop has been expanded, as we have added the hoisted/pushed code, while changing affected loads to register-register moves. Not only do we have repetitive moves, but notice that line 22 actually moves a register to itself! Cleaning up all these unnecessary register moves is the job of the various copy propagation forms.

Figure 5.6(a) shows this same code, but we have applied forward copy propagation (**cp**) as well as **ra**, and we have done them in this order, and continued to apply them together until they no longer transform the code. While this has reduced the number of unnecessary moves, there are still some obvious ones, as in lines 9 and 10, where the fact that we reused the `xmm0` register has forced us to retain the move to `xmm1`. Adding reverse copy propagation (**rc**) to the optimization block handles these kinds of renaming problems, as shown in Figure 5.6(b), where line 8 loads the value directly into `xmm1`. Reverse copy propagation started with the register-to-register move at line 10 of Figure 5.6(a), and determined that the source's live range began on line 9. The move on line 10 was then deleted, and and the destination register (`xmm1`) was substituted for the source register (`xmm0`) on line 9, leading to line 8 of Figure 5.6(b).

The code is starting to look much better, but we can reduce register pressure by making one of the loads implicit. Exploiting the CISC ISA in this way is the

```
1#   Hoisted loads from ra
2    movl    48(%esp),%ebp
3    movsd   4(%esp),%xmm2
4    movl    36(%esp),%eax
5    movl    40(%esp),%ecx
6 .local  _LOOP_0
7 _LOOP_0:
8#   x = X[0];
9    movsd   (%ecx),%xmm0
10   movsd   %xmm0,%xmm1
11#  y = Y[0]
12   movsd   (%eax),%xmm0
13#  dot += x * y;
14   mulsd   %xmm0,%xmm1
15   addsd   %xmm1,%xmm2
16#  X += 1
17   addl    $8, %ecx
18#  Y += 1
19   addl    $8, %eax
20#  while(--i);
21   subl    $1, %ebp
22   jne     _LOOP_0
23 .local  _LOOP_END_0
24 _LOOP_END_0:
25#  Pushed store from ra
26   movlpd  %xmm2,4(%esp)
```

(a) **Assembly, after ra and cp**

```
1    movl    48(%esp),%ebp
2    movsd   4(%esp),%xmm2
3    movl    36(%esp),%eax
4    movl    40(%esp),%ecx
5 .local  _LOOP_0
6 _LOOP_0:
7#   x = X[0];
8    movsd   (%ecx),%xmm1
9#   y = Y[0]
10   movsd   (%eax),%xmm3
11#  dot += x * y;
12   mulsd   %xmm3,%xmm1
13   addsd   %xmm1,%xmm2
14#  X += 1
15   addl    $8, %ecx
16#  Y += 1
17   addl    $8, %eax
18#  while(--i);
19   subl    $1, %ebp
20   jne     _LOOP_0
21 .local  _LOOP_END_0
22 _LOOP_END_0:
23#  Pushed store from ra
24   movlpd  %xmm2,4(%esp)
```

(b) **Assembly, after ra, cp and rc**

Figure 5.6: DDOT Loop Assembly with ra, cp, and rc

job of remove one use loads (**u1**), which merges a computation and load into one instruction. Note that reducing register pressure is not an obvious benefit in the current loop, but after additional transformations such as unrolling and accumulator expansion, it can become critical. Even absent such changes, additional benefit may be provided both by the increased code density and having the additional register available for later use in global application of **ra**.

Figure 5.7 demonstrates **u1**. In order to make the examples more compact, we hereafter omit the hoisted/pushed load/stores and the computation identifying comments. Therefore, Figure 5.7(a) recapitulates Figure 5.6(b) without such instructions, and Figure 5.7(b) shows the same code after **u1** is added to the optimization block. We see that we have one less instruction in the loop, and the register `xmm3` is no

```
1 .local  _LOOP_0
2 _LOOP_0:
3     movsd   (%ecx),%xmm1
4     movsd   (%eax),%xmm3
5     mulsd   %xmm3,%xmm1
6     addsd   %xmm1,%xmm2
7     addl    $8, %ecx
8     addl    $8, %eax
9     subl    $1, %ebp
10    jne     _LOOP_0
```

(a) DDOT Loop Assembly with ra, cp, and rc

```
1 .local  _LOOP_0
2 _LOOP_0:
3     movsd   (%ecx),%xmm1
4     mulsd   (%eax),%xmm1
5     addsd   %xmm1,%xmm2
6     addl    $8, %ecx
7     addl    $8, %eax
8     subl    $1, %ebp
9     jne     _LOOP_0
```

(b) DDOT Loop Assembly with ra, cp, rc, and u1

Figure 5.7: DDOT Loop Assembly with ra, cp, rc and u1

longer used. This is because the load of `xmm3` from line 4 of Figure 5.7(a) has been merged into the computational instruction on line 4 of Figure 5.7(b).

5.7.1.1 SIMD Vectorization

The next step is to show how SIMD Vectorization is applied. Since **SV** makes global changes to the function, it is not easily understood by viewing isolated code fragments, and therefore Figure 5.8 shows the complete assembly generated after vectorization. For the sake of both clarity and brevity, we have applied all repeatable optimizations using the defaults given in Section 5.6.10.

Vectorization creates many complexities, one of which is the need to keep vectors aligned to 128 bits. This is problematic in that the IA-32 ABI (Application Binary Interface) only guarantees stack pointer alignment to 32 bits. Thus, in lines 7 and 8, we shift off the last 4 bits of the stack pointer so that it is known to be aligned to 128 (note that the stack grows downward in this ISA, and so this just results in expanding our local frame by a bit). On this architecture FKO normally employs the frame pointer as a general purpose register (otherwise, the compiler would only have six available integer registers, as two would be tied up pointing to the stack). However, since how much the shifts change the stack pointer is not known at compile time, we must save the old stack pointer, or we will unable to recover it (in order to restore the stack pointer on function exit, or to access parameters passed to the

```
1 .text
2 .globl  ATL_UDOT
3 ATL_UDOT:
4 #   Ensure 128 bit alignment of stack ptr
5     movl    %esp,%edx
6     subl    $128, %esp
7     shrl    $4, %esp
8     shll    $4, %esp
9     movl    %edx,8(%esp)
10#   Save registers & ld old stack ptr new reg
11    movl    %ebp,(%esp)
12    movl    %ebx,4(%esp)
13    movl    8(%esp),%ebp
14#   para 0, name=N
15    movl    4(%ebp),%edx
16    movl    %edx,%eax
17#   para 1, name=X
18    movl    8(%ebp),%edx
19    movl    %edx,%ecx
20#   para 2, name=incX: UNUSED
21#   para 3, name=Y
22    movl    16(%ebp),%edx
23#   para 4, name=incY: UNUSED
24#   Initialize locals to constants
25    xorpd   %xmm0,%xmm0
26#   END OF FUNCTION PROLOGUE
27    subl    $1, %eax
28    jle _CUNE_LOOP_0
29#   Init accumulator vector for dot
30    movsd   %xmm0,%xmm2
31    xorpd   %xmm0,%xmm0
32    movsd   %xmm2, %xmm0
33    movl    %eax,%ebp
34    movapd  %xmm0, %xmm2
35    movl    %ecx,%eax
36    movl    %edx,%ecx
37 .local  _LOOP_0
38 _LOOP_0:
39    movapd  (%eax), %xmm0
40    mulpd   (%ecx), %xmm0
41    addpd   %xmm0, %xmm2
42    addl    $16, %ecx
43    addl    $16, %eax
44    subl    $2, %ebp
45    jg  _LOOP_0
46#   Reduce accumulator vector for dot
47    movl    %ecx,%ebx
48    movl    %eax,%ecx
49    pshufd  $0xee,%xmm2,%xmm1
50    addpd   %xmm1, %xmm2
51    movlpd  %xmm2, 80(%esp)
52    movsd   80(%esp),%xmm0
53    movsd   %xmm0,%xmm2
54    subl    $-1, %ebp
55    movl    %ebp,%eax
56    jne _IFKOCD0__LOOP_0
57 .local  _CUDONE_LOOP_0
58 _CUDONE_LOOP_0:
59#   set return val, restore regs, and return
60 .local  _IFKO_EPILOGUE
61 _IFKO_EPILOGUE:
62    movsd   %xmm2,104(%esp)
63    fldl    104(%esp)
64    movl    (%esp),%ebp
65    movl    4(%esp),%ebx
66    movl    8(%esp),%esp
67    ret
68 .local  _CUNE_LOOP_0
69 _CUNE_LOOP_0:
70    addl    $1, %eax
71    movsd   %xmm0,%xmm2
72    movl    %edx,%ebx
73 .local  _IFKOCD0__LOOP_0
74 _IFKOCD0__LOOP_0:
75    movsd   (%ecx),%xmm0
76    mulsd   (%ebx),%xmm0
77    addsd   %xmm0,%xmm2
78    addl    $8, %ecx
79    addl    $8, %ebx
80    subl    $1, %eax
81    jne _IFKOCD0__LOOP_0
82    jmp _CUDONE_LOOP_0
```

Figure 5.8: SIMD Vectorized DDOT Assembly

routine). Therefore, we see that line 5 copies the original stack pointer to a temporary register before the stack pointer is modified. This temporary register is then used to store the old stack pointer to the newly allocated stack frame (line 9), allowing the original stack pointer to be a target for register allocation like any other local (it is loaded to a new register on line 13).

Line 25 initializes `dot` to zero for use in the loop. However, the `dot` that has just been set is a scalar. The loop, however, has been vectorized, and so it needs `dot` in a vector register. This is done on lines 31, 32 and 34. A careful examination of these lines reveals that they are, in fact, not needed. This is easily discoverable in the assembly, but not apparent in our LIL. These lines are not needed because for FKO on the x86, vector and scalar registers are aliased, and so we can use normal moves to transfer them, and in this case, both our scalar and vector `dot` registers wind up assigned to the physical register `xmm2`. However, our LIL assumes that scalar and vector registers are separate (as indeed they often are, for instance on the PowerPC, or indeed on the x86 if we used the x87 FPU for scalar floating point computation), and thus by default goes through memory when transferring between scalar and vector registers. In this case, FKO has avoided going through memory, but still retains some useless register-to-register moves. We will need to introduce some more architecture-specific information into the copy propagation phases (indicating exactly how scalar and vector registers are aliased) to avoid these moves. Because these scalar/vector conversions are always introduced outside the main loop, we have not bothered to introduce this system-dependent optimization yet, but we will probably do so eventually, particularly as we examine more complex kernels (where outer-loop transforms become more critical, as the inner loop is deeply nested).

Lines 31 and 32 are themselves the result of a series of optimizations. They started out as a store to memory of the scalar value, followed by a read into a vector register, but copy propagation knows how to move a value from a scalar register to the lower 64 bits of the vector register, as shown in line 32. However, this leaves the upper 64

bits untouched, and so the target vector register must be zeroed before the move, which is what instruction 31 is doing. Line 34 is a vector-to-vector register move, moving the constructed vector register `xmm0` to its eventual target, `xmm2`. Note that we cannot construct the vector value in `xmm2` directly, due to live range conflicts. While our LIL treats vector and scalar registers as separate sets, it of course has accurate dependency information, and so it knows that when `xmm2` is used as to hold a scalar value (in this case the scalar value of `dot`), whichever vector register that corresponds to it cannot be assigned until that scalar live range is complete. Since the scalar version of `xmm2` is not dead until vector construction is complete, and construction takes multiple instructions, we have to use the temporary vector register `xmm0`.

Lines 49-53 (after the vectorized loop) do the opposite: they take the vector values of `dot`, and reduce them to a scalar value. After the loop, however, this requires summing the two partial results as well as moving the data between scalar/vector types. Therefore, line 49 moves the upper 64 bits of the vector register `xmm2` into the lower 64 bits of `xmm1`, and we then add them together on line 50, so that the complete `dot` is in the lower 64 bits of `xmm2`. Unfortunately, FKO was unable to remove the store and load to memory this time, and so line 51 stores `dot` to memory from a vector register, line 52 reads it from that location into a temporary scalar register, and line 53 moves it to its final scalar destination register. Again, there are many optimizations we can apply to make this more efficient, but we have not yet done so since it outside the loop.

Lines 38-45 contain the vectorized loop. We see that this loop is essentially the same as the scalar loop shown in Figure 5.7(b), with two notable exceptions. First, all scalar instructions have been replaced by their vector equivalents. Second, the update of the pointers is by 16 bytes (128 bits) rather than 8, and the index is updated by subtracting two rather than one. This is because vectorization is equivalent to a scalar unrolling of two for double precision.

Because it is equivalent to a computational unrolling of 2, we must have a cleanup loop for odd values of N, and we must introduce the appropriate branches to this cleanup loop, just as we do in unrolling. Lines 74-82 comprise the scalar cleanup loop. Lines 27 and 28 supply the pre-loop branch to cleanup (to handle the case of $N < 2$), and lines 54 and 56 do the same for post-vector-loop cleanup (for the case $N > 2$, but $N \mod 2 \neq 0$).

Lines 60-66 comprise the function epilogue, which restores the callee-saved registers (including the stack pointer), and returns. Lines 62 and 63 reveal another complexity of this architecture. The IA-32 ABI requires functions returning floating point values to store them as the top register of the x87's register stack. Moving between SSE registers and x87 registers requires going through memory, so line 62 stores the return value (`dot`) from an SSE register, and line 63 loads it to the x87 register stack top.

In our pre-loop test, we subtracted `UR`-1 from the start value of the loop index in order to get the numbers correct for efficient unrolled loop indexing. If we never enter this loop, however, this value must be added back in, which is why the pre-loop tests jumps to the block on lines 69-72, rather than directly to the cleanup loop, as the post-loop test does.

Note that since the cleanup loop and corresponding conditionals are generated, FKO has considerable freedom to choose where to add these blocks. Therefore, we have have added them in such a way that the fall-through cases assume that the vectorized loop is entered, and loop cleanup is not required (thus not adding overhead to the most efficient case).

5.7.2 DASUM Example Illustrating UR, AE, PF, ul, and ls

In this section we use `dasum` (Figure A.3) to illustrate various optimizations. First, we use a simple example to show how the enforce load/store **ls** transformation can be useful, and then a more detailed listing is given in order to demonstrate how

```
1  LOOP i = 0, N
2  LOOP_BODY
3      x = X[0];
4      x = ABS x;
5      sum += x;
6      X += 1;
7  LOOP_END
```

(a) As HIL

```
1 .local  _LOOP_0
2 _LOOP_0:
3     movsd (%ecx),%xmm0
4     andpd 16(%esp),%xmm0
5     addsd %xmm0,%xmm2
6     addl  $8, %ecx
7     subl  $1, %eax
8     jne   _LOOP_0
```

(b) Without **ls**

```
1     movapd 16(%esp),%xmm3
2 .local  _LOOP_0
3 _LOOP_0:
4     movsd (%ecx),%xmm0
5     andpd %xmm3,%xmm0
6     addsd %xmm0,%xmm2
7     addl  $8, %ecx
8     subl  $1, %eax
9     jne   _LOOP_0
```

(c) With **ls**

Figure 5.9: ASUM Loop

loop unrolling (**UR**), accumulator expansion (**AE**), prefetch (**PF**), and useless label elimination (**ul**) work together.

Figure 5.9(a) shows the `dasum` written in our HIL. Line 3 loads the array value, line 4 takes its absolute value, and line 5 adds this into the running sum (`dasum` is the absolute value sum of an array). Figure 5.9(b) shows this same inner loop in assembly. Here, we are using all of the default optimizations, except we have turned off **ls**. Line 3 of this listing loads the array value, line 4 takes its absolute value (more on this below), and line 5 adds the absolute value into the running sum.

In order to understand this code, we need to understand how absolute value is performed using SSE, which does not have an explicit absolute value instruction. Fortunately, a bitwise `and` can be used to produce such an operation. First, we construct a 128 bit integral value that has all bits set to 1, except the sign bit of each *veclen* floating point elements, which are instead set to 0. Absolute value of a scalar or vector may then be produced by performing a bitwise `and` of this integral value and the register holding the number to be absolute valued. Because we want to be able to issue absolute value with minimal register use, the front-end generates the in-memory version of the instruction, as shown on line 4 of Figure 5.9(b) (where the integral value has been written to the stack location `16(%esp)`). Inside a loop, however, this can lead to repetitive memory reads. By running enforce load store as shown in the architectural defaults, the code in Figure 5.9(c) is created, where this

```
1      xorpd   %xmm1,%xmm1
2      movsd   %xmm1,%xmm3
3  .local  _LOOP_0
4  _LOOP_0:
5      movsd   (%ecx),%xmm0
6      andpd   %xmm2,%xmm0
7      addsd   %xmm0,%xmm3
8  .local  _IFKOCD1__LOOP_0
9  _IFKOCD1__LOOP_0:
10     movsd   8(%ecx),%xmm0
11     andpd   %xmm2,%xmm0
12     addsd   %xmm0,%xmm3
13 .local  _IFKOCD2__LOOP_0
14 _IFKOCD2__LOOP_0:
15     movsd   16(%ecx),%xmm0
16     andpd   %xmm2,%xmm0
17     addsd   %xmm0,%xmm3
18 .local  _IFKOCD3__LOOP_0
19 _IFKOCD3__LOOP_0:
20     movsd   24(%ecx),%xmm0
21     andpd   %xmm2,%xmm0
22     addsd   %xmm0,%xmm3
23     addl    $32, %ecx
24     subl    $4, %eax
25     jg  _LOOP_0
```

(a) Unrolled to 4

```
1      xorpd   %xmm1,%xmm1
2      movsd   %xmm1,%xmm3
3  #   Shadow accum init
4      xorpd   %xmm0,%xmm0
5      movsd   %xmm0,%xmm4
6  .local  _LOOP_0
7  _LOOP_0:
8      prefetchnta 256(%eax)
9      movsd   (%eax),%xmm0
10     andpd   %xmm2,%xmm0
11     addsd   %xmm0,%xmm4
12     movsd   8(%eax),%xmm0
13     andpd   %xmm2,%xmm0
14     addsd   %xmm0,%xmm3
15     movsd   16(%eax),%xmm0
16     andpd   %xmm2,%xmm0
17     addsd   %xmm0,%xmm4
18     movsd   24(%eax),%xmm0
19     andpd   %xmm2,%xmm0
20     addsd   %xmm0,%xmm3
21     addl    $32, %eax
22     subl    $4, %ebp
23     jg  _LOOP_0
24 #   Accumulator reduce
25     addsd   %xmm4,%xmm3
```

(b) With **UR**=4, **ul**, **PF**, and **AE**=2

Figure 5.10: DASUM loop unrolled to 4

implicit load has first been changed back to an explicit load by **ls**, and then register assignment has hoisted the load of the integral value out of the loop.

Figure 5.10(a) shows the `dasum` loop that has been unrolled to 4, but without applying useless label elimination. We see that the loop simply repeats Figure 5.9(c) four times, with a few minor changes. First, **UR** does not just blindly repeat the loop control and pointer updates, but instead changes the address references using constants in the loop, and does these updates only one time. Notice that the loads from the array X have an offset, as shown on lines 5, 10, 15, and 20 of Figure 5.10(a). We can then add $\mathbf{UR} * sizeof() = 4 * 8 = 32$ to the X pointer at the bottom of the loop (line 23). If we updated the pointer between each access, there would be a potential slowdown, since an integer add would need to be performed before each of

the last three loads. In the present formulation, the architecture is free (assuming register renaming is done by the hardware) to issue all four loads in parallel. Line 24 shows that the update of the loop index is now by 4, rather than 1.

When duplicating the loop body, any labels must be made distinct, and so we see that the labels of the duplicated blocks (lines 9, 14 and 19) have a standard prefix added to them. These extra labels are in fact useless in this operation. If labels aren't being used, we want to remove them, since most optimizations are more robust within a basic block than when applied across blocks.

Figure 5.10(b) shows the same loop, but we have applied **ul**, **PF** and **AE**=2. The useless label elimination verifies that the duplicated labels are not referenced anywhere in the code, and removes them, resulting in a loop consisting of a single basic block again. On this architecture, the cache line size is 128 bytes, or 16 double precision elements, and so one prefetch instruction (line 8) is sufficient.

Our final transformation of interest is accumulator expansion. Notice that the summation updates (lines 7, 12, 17 and 22) of Figure 5.10(a) all update the same register, `xmm3`, but in Figure 5.10(b), we alternate between uses of `xmm4` and `xmm3` (lines 11, 14, 17 and 20). This means that the extra register must be initialized before the loop (lines 4 and 5), and added back into the total after the loop (line 25).

5.7.3 DAXPY Example Illustrating WNT and lu

Figure 5.11(a) shows the main `daxpy` loop in our HIL, while Figure 5.11(b) shows the assembly generated by FKO when vectorization is applied, and all repeatable defaults other than last use removal (**lu**) are applied.

The correspondence of these two loops is easy to see: in both loops, the first two lines of the loop body load the input values from their respectively arrays, the third multiplies by the X value by *alpha*, and the fourth adds the result to the value obtained from Y. We then store the value back to Y, and increment the array and

```
1 LOOP i = 0, N
2 LOOP_BODY
3     x = X[0];
4     y = Y[0];
5     x = x *
          alpha;
6     y += x;
7     Y[0] = y;
8     X += 1;
9     Y += 1;
10 LOOP_END
```

(a) As HIL

```
1  .local  _LOOP_0
2  _LOOP_0:
3      movapd    (%eax), %xmm1
4      movapd    (%ecx), %xmm3
5      mulpd     %xmm2, %xmm1
6      addpd     %xmm1, %xmm3
7      movl      %ecx,%edx
8      movapd    %xmm3, (%edx)
9      addl      $16, %ecx
10     addl      $16, %eax
11     subl      $2, %ebp
12     jg        _LOOP_0
```

(b) As vectorized assembly

```
1  .local  _LOOP_0
2  _LOOP_0:
3      movapd    (%eax), %xmm1
4      mulpd     %xmm2, %xmm1
5      addpd     (%ecx), %xmm1
6      movl      %ecx,%edx
7      movntpd   %xmm1, (%edx)
8      addl      $16, %ecx
9      addl      $16, %eax
10     subl      $2, %ebp
11     jg        _LOOP_0
```

(c) As assembly, with **WNT** & **lu**

Figure 5.11: DAXPY Loop

index counters (since Figure 5.11(b) has been vectorized, it increments by twice as much as Figure 5.11(a)).

On the P4E, if this loop traverses enough memory to overflow the cache, it is a performance win to use non-temporal writes (**WNT**) for the stores of Y, and we have done so in Figure 5.11(c). Notice that the store of Y (line 8) of Figure 5.11(b) uses the instruction `movapd`, while the store of Y (line 7) of Figure 5.11(c) uses its non-temporal equivalent, `movntpd`.

The final optimization shown here is last use removal (**lu**). In our our previous examples, we have usually been able to make one array load implicit through the use of **u1**, but we cannot do so here, because of the usage pattern. It's not a good idea to overwrite the register holding *alpha* (`xmm2`) since it is loop invariant, and overwriting it would require us to reload it in the loop. Therefore, since x86 assembly can have only source operands coming from memory, we are unable to apply **u1** to line 5 of Figure 5.11(b).

Instead, we notice that the last use of the register holding the value of $X(i)*alpha$ (`xmm1`) is in the commutative instruction `addpd` on line 6 of Figure 5.11(b). In this case, we can make the load of Y implicit, by reordering the instruction so that the Y value comes from memory, and `xmm1` is overwritten (since this was its last use in the register), as in line 5 of Figure 5.11(c).

5.8 Current Iterative Search

The present iterative search varies only the fundamental optimizations. The repeatable optimizations are therefore always those given in Section 5.6.10, while our fundamental defaults are outlined in Section 5.4.7. The master search performs the following sub-searches in this order: *WNT Search*: FKO is queried for the arrays that are set in the loop, and non-temporal writes are tried on each in turn, and are used for that array if they provide a speedup. *PF Type Search*: Each supported type of prefetch instruction is tried for "prefetch for read" and "prefetch for write". Best values are kept. *PF Distance Search*: For each prefetch target (returned by FKO analysis) a linear search is performed using line size increments. We also try not prefetching the array, and prefetching shorter distances less than the line size. Best discovered values for each array are retained. *Loop Unroll Search*: Try all powers of two between [1:128]. Powers of two are used because they allow for a quick search and keep data access in a given loop iteration a multiple of the cache line size; a more complete search would probably yield some improvement, but this value will be refined further by later stages of the search anyway. *Accumulator Expansion Search*: Try performing **AE** on all valid targets (returned by FKO) in turn. This optimization depends on unrolling, so we try a few different unrollings for each expansion. Let the number of accumulators currently being tried for a given variable be N_a, and the present unrolling factor be N_u. Our present search tries all N_a in the range $2 \leq N_a \leq 6$ (six is a safe maximum for the x86, where the ISA has only 8 registers). For each such N_a, we try using the current loop unrolling, N_u. When there is a mismatch between N_a and N_u, we try additional loop unrollings in order to avoid cross-iteration pipeline stalls. More precisely, if $N_u < N_a$, additionally try the loop unrolling of N_a. If $N_u > N_a$ and $N_u \bmod N_a \neq 0$, we try two additional unrollings of $\lceil \frac{N_u}{N_a} \rceil \times N_a$ and $\lfloor \frac{N_u}{N_a} \rfloor \times N_a$.

CHAPTER 6

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter presents and analyzes results on two of today's premier x86 implementations, and is organized in the following way: Section 6.1 outlines the floating point kernels that are being optimized, Section 6.2 discusses version and timing methodology information, and Section 6.3 presents the raw results. Section 6.4 then provides the main analysis of these results, while Section 6.5 points out some interesting (but non-essential) details. Finally, in those few cases where iFKO fails to provide the fastest implementation, Section 6.6 describes the transformations that the most successful tuning technique utilized to get the fastest kernel, so it is clear whether or not the required optimization(s) can be eventually be generalized into our compilation framework.

6.1 Problem Domain and 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 6.1 (Appendix A provides a complete listing of the actual kernels input to the compilers, both in ANSI C and in our HIL). The performance of the BLAS are usually reported in MFLOPS (millions of floating point operations per second), but 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.

Table 6.1: Level 1 BLAS summary

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]); }`
	`}`
		$2N$

6.2 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.

Table 6.2: Compiler flag and version information by platform

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

We report numbers for two very different high-end x86 architectures, the Intel Pentium 4E and AMD Opteron. Further platform, compiler and flag information is summarized in Table 6.2 (for the profile build and use phases, the appropriate flags were suffixed to those shown Table 6.2.) 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, and the minimum was taken. All timings were done sequentially, and run on an unloaded machine. Because these are actual timings (as opposed to simulations), there is still some fluctuation in performance numbers despite these precautions, so small gaps of around a percentage point may not represent true differences.

Therefore, because the search is empirical, it is not strictly repeatable. In general, truly bad choices are rarely made, as they tend to be above clock resolution. Nonetheless, techniques can be employed to improve the results of any empirical search. The simplest is to run the search several times, and take the best available transformation list found for each routine (i.e., utilize the `ddot` flags from run A, and the `samax` flags from run B). A more sophisticated approach takes the result of previous searches as the starting point of a new search, or reruns certain sub-searches with updated information from subsequent sub-searches, or tweaks various bound information in hopes of finding undiscovered outlying transformations. However, we wanted to compare fully-automatic use of the present systems, and so each list of results was obtained by simply running two scripts sequentially. The first script times all fixed methods (gcc,icc,ATLAS,FKO), and the second is the the default empirical search of iFKO.

6.2.1 Input Routines

Appendix A shows the input routines to all compilers. With the exception of `iamax`, 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). Our HIL does not yet support scoped ifs, however, and so `iamax` was originally coded for all compilers (in the appropriate language) as shown in Figure A.7(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 implementation shown in Figure A.7(a) for these compilers.

6.3 Overview of Results

This section presents our experimental results, and explains the formats in which we present them. These results show adaptation to the kernel, architecture, and the context (in this case, out-of-cache, or L2-cache resident). Analysis of these results are provided in the following sections.

Figures (6.1, 6.2, 6.3, 6.4) report the percentage of the best observed performance provided by the following 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.

Since all results discussed so far are relative to the best tuning method, it is easy to lose track of the actual performance of the individual operations. Therefore, Figure 6.5 shows the speed of these operations in MFLOPS, computed as discussed in Section 6.1. 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).

Figure 6.6 shows the speedup of the in-L2 cache 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. The `iamax` operation, whose performance is limited mainly by branches, is a good example of this, in that the in-cache numbers show no improvement at all. On the other extreme, bus-bound operations such as `swap` or `axpy` show more than five-fold speedups for in-L2 timings. One oddity in these numbers is that `iamax`'s Pentium 4

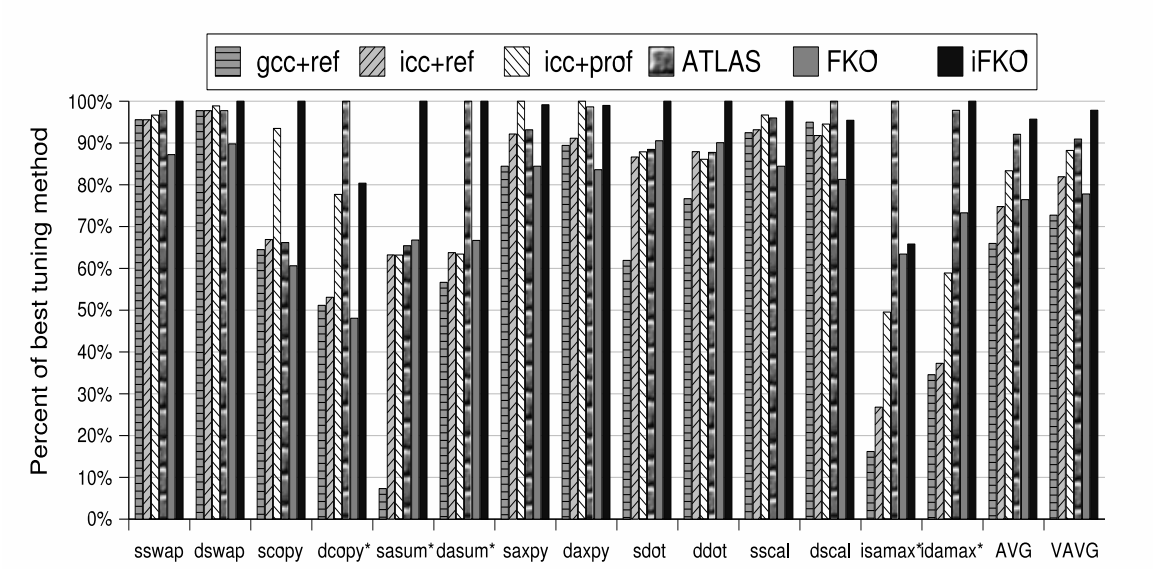


Figure 6.1: Relative speedups of various tuning methods on 2.8Ghz P4E, N=80000, out-of-cache

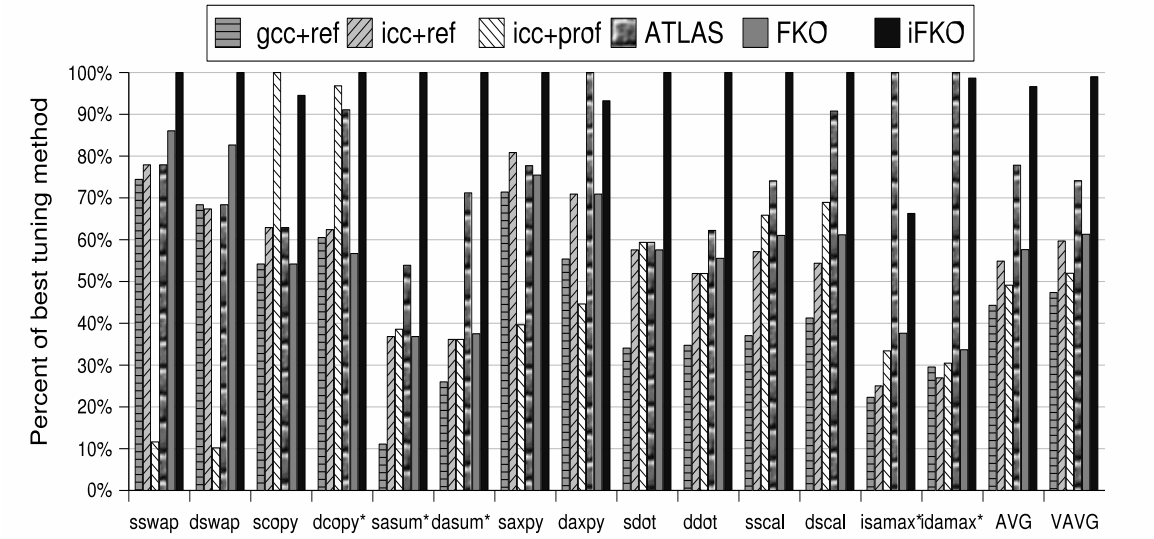


Figure 6.2: Relative speedups of various tuning methods on 1.6Ghz Opteron, N=80000, out-of-cache

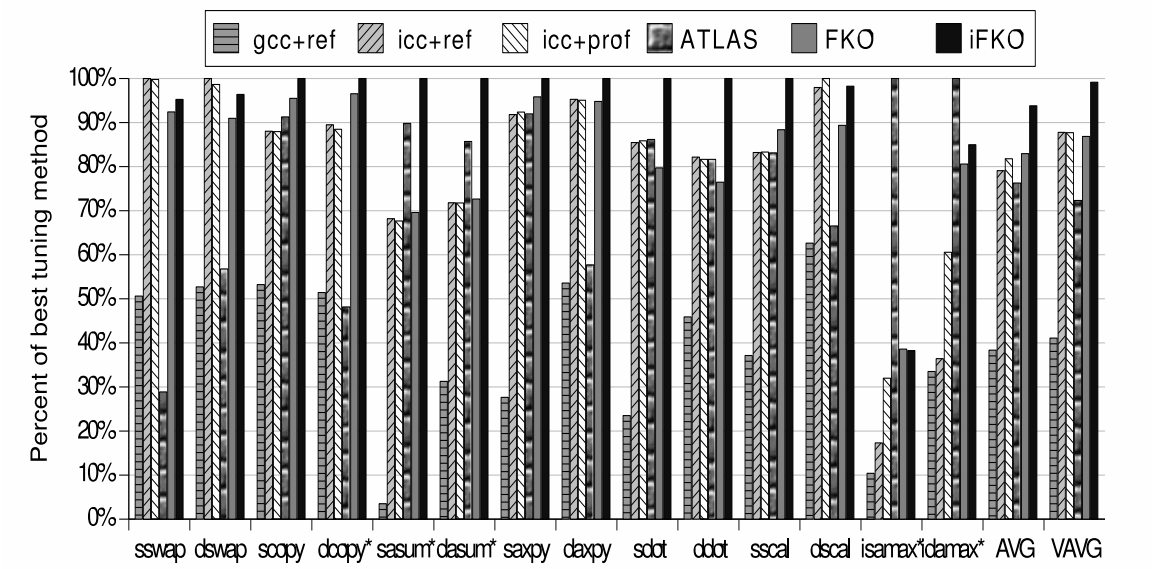


Figure 6.3: Relative speedups of various tuning methods on 2.8Ghz P4E, N=1024, in-L2-cache

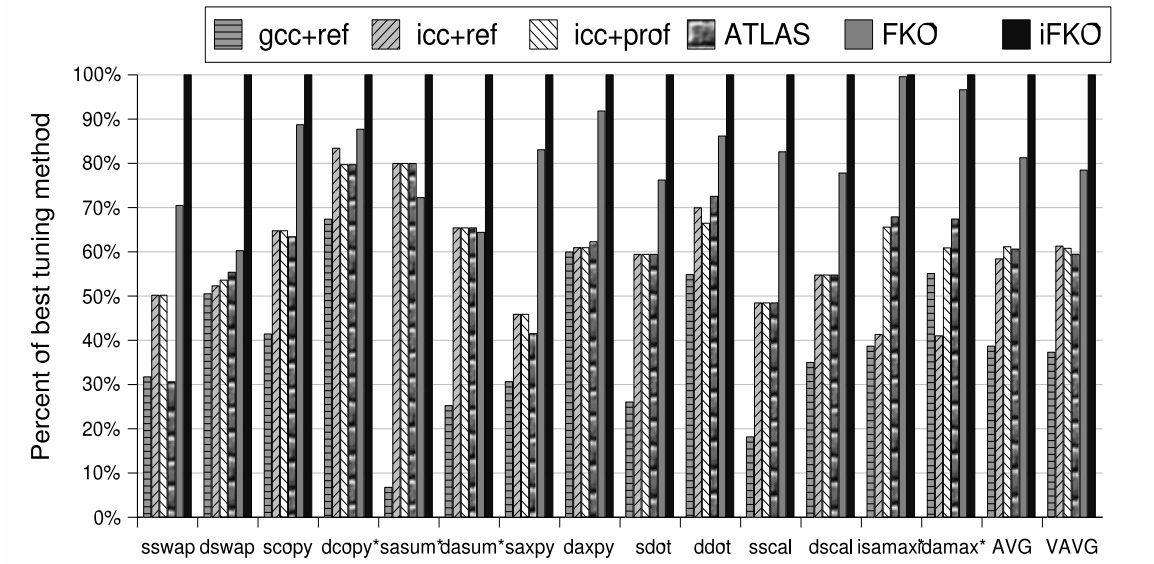
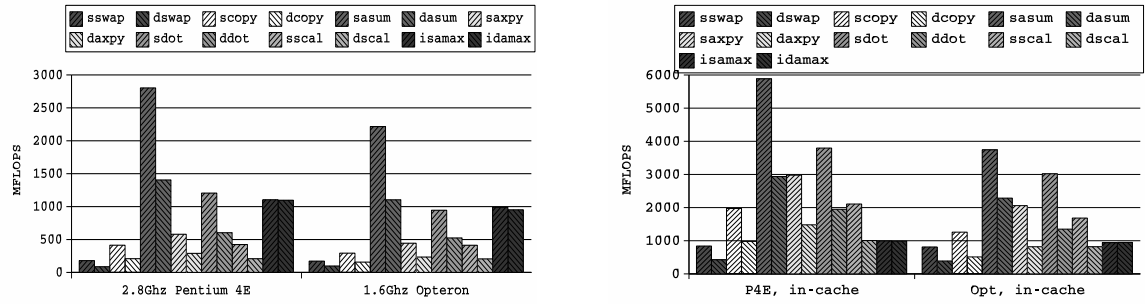


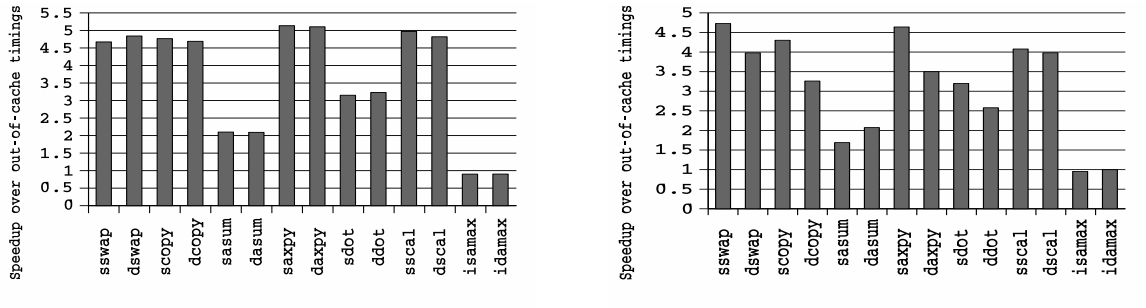
Figure 6.4: Relative speedups of various tuning methods on 1.6Ghz Opteron, N=1024, in-L2-cache

in-L2 performance is actually slightly slower than out-of-cache. This is not a timing error, and is discussed in Section 6.5.



(a) Out of cache (b) P4E, in Level 2 cache

Figure 6.5: BLAS performance in MFLOPS



(a) P4E (b) Opteron

Figure 6.6: Speedup of In-cache over Out-of-cache

Tables (6.3, 6.4, 6.5, 6.6) show the transformational parameter values found by the empirical search for each program/context. Section 5.4 defines the abbreviations used in the headings, and Section 5.8 provides the default values used by FKO. The prefetch parameters varied include 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:

- none : better performance was obtained without prefetching that operand,
- `tX`: SSE temporal prefetch to cache of level $X+1$ (eg., `prefetcht0`, `prefetcht1`, etc.),

- `nta`: SSE non-temporal prefetch to lowest level of supported cache (`prefetchnta`),
- `w`: 3DNow! prefetch for write (`prefetchw`).

Figure 6.7 shows the speedup obtained by empirically tuning the various optimization parameters, and Figure 6.8 shows the same values but zoomed so that only the first 150% of speedup is displayed (this is necessary because the large Opteron speedups make it difficult to see full details for other architectures). Therefore, these figures show the speedup of code tuned by iFKO over that produced by FKO, *not over code in which a given transformation has not been applied.* For instance, FKO defaults to unrolling so that one iteration of the loop accesses one cache line of data. In Figure 6.7 we see that the empirical tuning of **UR** often provides modest or no benefit. However, this does not imply that unrolling is unimportant in these cases; instead it says that FKO's default value is good.

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, for the out-of-cache P4E tuning of `sasum` shown in 6.8, empirically tuning the [non-temporal writes, prefetch instruction, prefetch distance, unrolling, accumulator expansion], provided speedups of [0, 1, 46, 0, 3]%, respectively, which together resulted in an iFKO-tuned kernel that ran 1.5 times faster than the same kernel when compiled by FKO. As shown by these graphs, all empirically tuned parameters contributed to speeding up at least some operations/contexts.

6.4 General Analysis

In comparing the tuning mechanisms (Figures 6.1, 6.2, 6.3 and 6.4), iFKO provides the best performance on average for all studied architectures and contexts, better even than the hand-tuned kernels found by ATLAS's own empirical search.

Table 6.3: Transformation parameters for 2.8Ghz Pentium 4E, N=80000, all caches flushed

BLAS	SV: WNT	PF X INS:DST	PF Y INS:DST	UR: AC
sswap	Y:Y	t0:56	t0:40	4:0
dswap	Y:Y	t0:128	t0:64	2:0
scopy	Y:Y	none:0	none:0	2:0
dcopy	Y:Y	none:0	none:0	1:0
sasum	Y:N	nta:1024	n/a:0	5:5
dasum	Y:N	t0:1024	n/a:0	5:5
saxpy	Y:Y	nta:1408	nta:32	2:0
daxpy	Y:Y	t0:768	t0:40	2:0
sdot	Y:N	nta:1024	nta:384	3:3
ddot	Y:N	nta:768	nta:384	5:5
sscal	Y:Y	nta:1792	n/a:0	1:0
dscal	Y:Y	none:0	n/a:0	2:0
isamax	N:N	nta:640	n/a:0	8:0
idamax	N:N	t0:1664	n/a:0	8:0

Table 6.4: Transformation parameters for 1.6Ghz Opteron, N=80000, all caches flushed

BLAS	SV: WNT	PF X INS:DST	PF Y INS:DST	UR: AC
sswap	Y:N	w:1792	w:448	2:0
dswap	Y:N	nta:960	nta:704	1:0
scopy	Y:Y	none:0	none:0	1:0
dcopy	Y:Y	none:0	none:0	1:0
sasum	Y:N	t0:1664	n/a:0	4:4
dasum	Y:N	nta:1920	n/a:0	4:4
saxpy	Y:N	t0:1536	t0:448	4:0
daxpy	Y:N	nta:1472	t0:832	4:0
sdot	Y:N	nta:1600	nta:1664	3:3
ddot	Y:N	t0:1728	t0:704	4:4
sscal	Y:N	nta:640	n/a:0	1:0
dscal	Y:N	nta:1344	n/a:0	1:0
isamax	N:N	nta:768	n/a:0	16:0
idamax	N:N	nta:1920	n/a:0	32:0

Table 6.5: Transformation parameters for 2.8Ghz P4E, N=1024, only L1 cache flushed

BLAS	SV: WNT	PF X INS:DST	PF Y INS:DST	UR: AC
sswap	Y:N	nta:512	nta:32	16:0
dswap	Y:N	t0:384	t0:40	32:0
scopy	Y:N	nta:512	nta:1408	2:0
dcopy	Y:N	nta:1152	t0:1152	2:0
sasum	Y:N	t0:1408	n/a:0	16:2
dasum	Y:N	nta:1792	n/a:0	16:2
saxpy	Y:N	t0:768	t0:1152	8:0
daxpy	Y:N	t0:768	t0:384	8:0
sdot	Y:N	nta:896	nta:1664	64:4
ddot	Y:N	nta:1280	nta:1792	32:4
sscal	Y:N	nta:256	n/a:0	2:0
dscal	Y:N	nta:1536	n/a:0	2:0
isamax	N:N	t0:1152	n/a:0	32:0
idamax	N:N	nta:256	n/a:0	32:0

Table 6.6: Transformation parameters for 1.6Ghz Opteron, N=1024, only L1 cache flushed

BLAS	SV: WNT	PF X INS:DST	PF Y INS:DST	UR: AC
sswap	Y:N	w:256	w:128	32:0
dswap	Y:N	w:128	w:128	32:0
scopy	Y:N	t0:64	none:0	4:0
dcopy	Y:N	nta:192	none:0	64:0
sasum	Y:N	nta:64	n/a:0	64:3
dasum	Y:N	t0:256	n/a:0	4:4
saxpy	Y:N	nta:128	w:128	4:0
daxpy	Y:N	nta:32	w:128	4:0
sdot	Y:N	nta:192	nta:320	16:4
ddot	Y:N	nta:256	nta:448	6:3
sscal	Y:N	w:256	n/a:0	32:0
dscal	Y:N	w:128	n/a:0	4:0
isamax	N:N	t0:32	n/a:0	16:0
idamax	N:N	t0:768	n/a:0	32:0

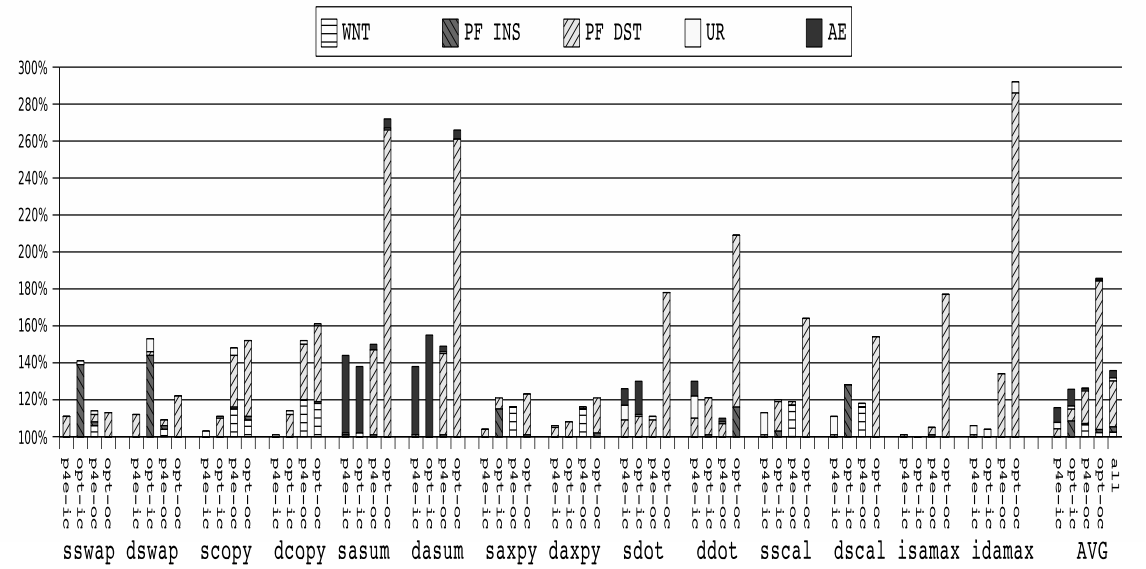


Figure 6.7: Percent speedup by transform due to empirical search

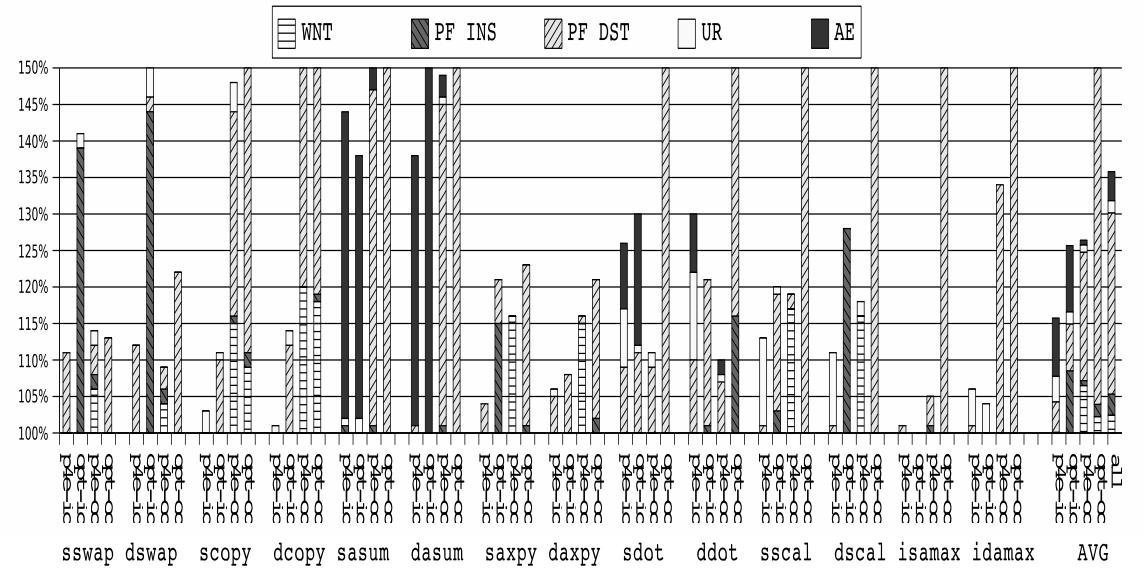


Figure 6.8: Percent speedup by transform due to empirical search (zoomed)

However, there are several individual cases where iFKO fails to provide the best performance. We analyze each such case in Section 6.6.

In examining the empirically tuned transformation parameter values (Tables 6.3, 6.4, 6.5 and 6.6) the most important observation is how variable these parameters are: they vary strongly depending on operation, architecture, *and* context. They vary weakly (mainly in prefetch distance) on precision as well. Without vectorization, other parameters would vary more strongly with precision (in vectorized code double and single precision operands are of the same size in bytes, and performing a vector computation on that data takes the same number of cycles, all of which is not true for scalar code of differing precisions). This suggests that any model capable of capturing 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.

Empirical methods can be invaluable in adapting to unexpected architectural changes, particularly when the compiler has not yet been (or will never be) fully tuned to the new platform (eg. Intel compiler on AMD platform). Examining the results for the Opteron demonstrates 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 6.2. To understand this behavior, we first observe that 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. 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. Figures 6.3 and 6.4 and Tables 6.5 and 6.6 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 Pentium 4E accounts for an impressive 43% of `sasum` speedup in-cache, while only improving performance by 3% for out-of-cache.

The effect of context on which optimizations are most critical can be most easily seen by examining the AVG results of Figures 6.7 and 6.8. For out-of-cache, the most important adaptation is clearly prefetch distance, which is only modestly important for in-cache timings, where accumulator expansion (and for the Opteron, prefetch instruction type as well) becomes the more critical optimization.

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 them in this case. This is why operations such as `swap` or `axpy` get relatively modest benefit in out-of-cache timings. Since prefetch optimization is one of our key strengths for this context, it is easy to see why iFKO does much better on the Opteron than on the P4E (when compared against all tuning mechanisms, including icc) for this context: 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. For the Opteron's in-cache performance, iFKO gets a similar boost in performance by varying the key parameters for this context, prefetch instruction type and accumulator expansion.

Figures 6.7 and 6.8 illustrate the importance of empirical tuning. While FKO has not undergone the intensive hand-tuning of the driving models for each particular architecture that occurs in traditional compiler porting, we tried to pick default values that make sense. Nonetheless, empirical tuning provided an almost factor of three speedup in the best case. Note that, again, operation, architecture, and context all strongly influence which transformation is most important. As we add more transformations that compete for the same resources, the value of empirical tuning should continue to climb. For instance, software pipelining, accumulator expansion, and register assignment all compete for registers, and thus striking the optimal balance will require information about their relative importance to the given operation, architecture and context. Empirical techniques appear to us to be the only tractable way to address these concerns.

6.5 Interesting Asides

From examining the generated assembly, it appears that neither gcc nor icc chose to issue software-directed prefetch. This makes some sense, in that prefetch (particularly distance) is difficult to model, while these vector access patterns are easily detected by the hardware, which allows for the hardware prefetch unit to examine the fetch pattern during runtime in order to optimize. However, iFKO would have chosen 'no prefetch' if the hardware could do a better job, and as the timings demonstrate, the hardware prefetch is clearly no match for empirically-tuned software prefetch.

Gcc's relative performance drop for in-cache tuning may at first seem surprising, but is easily understood given that gcc's main weakness compared to the other compilers is that it cannot auto-vectorize the loops. In-cache, computational

optimizations are all-important, and vectorization is the computational optimization which generally provides the greatest speedup. Obviously, gcc's relatively good performance on out-of-cache P4E is due to the flip-side of this: the P4E is more bus-bound than the Opteron, and therefore vectorization is less important for out-of-cache operation.

Figure 6.6 presents a puzzle for `iamax` on the P4E: the iFKO-tuned kernels are actually slightly slower (measured in MFLOPS) when ran and tuned for in-L2 operands than for when the kernels are run with cold caches! This appears to be a function of the vector length (remember that in-cache numbers use $N = 1024$, and out-of-cache numbers use $N = 80000$): as the length is increased, performance goes up, until the asymptotic performance is reached. Note that after tuned prefetch is added, this routine is not bus bound. Therefore, there is little to no benefit from having the operands in cache.

Having a long vector length, however, conveys several advantages. One obvious one is amortizing loop startup and shutdown. This should be as true for the Opteron as for the P4E, however, and the Opteron did not run slower for the in-cache timings (it may appear so for single precision, but these numbers are within clock resolution of each other). The P4E also has a trace instruction cache, which means that the x86-decoding cost is also amortized over the loop length; since the in-cache loop is heavily unrolled, this may be a small factor, but it seems unlikely to make such a noticeable difference. Note that if unrolling was a large burden, it would not have resulted in speedup, and the empirical search would not have retained it. Therefore, the bulk of the difference is probably due to the way `iamax` operates on normalized vectors. As the vector length grows, the number of branches that are actually taken (in order to change the maximum) shrinks in proportion to the total number of branches that are considered. Since each of these taken max branches results in a mispredict, and the P4 has 20 stage pipeline, getting the percentage of mispredicts

Table 6.7: Loss Case Summary

(a) For Out-of-cache Timings

ARCH	BLAS	BEST METH	MFLOP iFKO	MFLOP BEST	WHY
p4e	dcopy	atl+asm	209	260	fko-ns
p4e	saxpy	icc+prof	580	585	srch-clk
p4e	daxpy	icc+prof	290	293	srch-clk
p4e	dscal	atl+gcc	209	219	srch-asu
p4e	isamax	atl+asm	1105	1679	fko-an
opt	scopy	icc+prof	293	310	clk
opt	daxpy	atl+gcc	234	251	srch-asu
opt	isamax	atl+asm	990	1494	fko-an
opt	idamax	atl+asm	952	965	fko-an

(b) For In-L2 Timings

ARCH	BLAS	BEST METH	MFLOP iFKO	MFLOP BEST	WHY
p4e	sswap	icc+ref	841	883	srch-clk
p4e	dswap	icc+ref	426	442	srch-clk
p4e	dscal	icc+pref	1007	1025	srch-clk
p4e	isamax	atl+asm	995	2601	fko-an
p4e	idamax	atl+asm	989	1164	fko-an

down is critical on this architecture, and thus the longer the loop, the more efficient the algorithm.

6.6 Learning from Defeat

In this section we examine the cases where iFKO failed to provide the most optimal kernel implementation. Tables 6.7(a) and 6.7(b) summarize the cases (for out-of-cache and in-L2 cache, respectively) where the previously reported timings indicate that FKO did not provide the most well-tuned kernel. In these tables we first supply the architecture where the loss occurred (p4e/opt) and the tuning methodology that provided the best observed performance. We then report the performance of the two kernels in question (best tuning method and iFKO) in MFLOPS, followed by the appropriate abbreviation describing the reason for the loss (defined below). Many of the cases turn out to be the result of search resolution errors, and in those cases Table 6.8 shows the new parameter values and performance results (this table is described in greater detail below).

There are several possible causes for iFKO to lose, and we need to distinguish between them in order to draw proper conclusions. The categories of importance (and their abbreviations used in Table 6.7) are:

Table 6.8: Better Transformation Parameters Found by Repeated Searches

ARCH	CACHE	BLAS	SV: WNT	PF X INS:DST	PF Y INS:DST	UR: AC	BEST: MFLOP
P4E	OC	dcopy	Y:Y	none:0	none:0	2:0	N:205
P4E	OC	saxpy	Y:Y	nta:384	nta:512	2:0	Y:586
P4E	OC	daxpy	Y:Y	nta:384	nta:512	2:0	Y:293
P4E	OC	dscal	Y:Y	nta:896	N/A:0	1:0	N:210
P4E	IC	sswap	Y:N	t0:128	t0:1408	4:0	T:899
P4E	IC	dswap	Y:N	nta:96	nta:1280	1:0	T:443
P4E	IC	dscal	Y:N	t0:1536	N/A:0	1:0	Y:1060

- **clk**: Two timings are actually within clock resolution. Determined by timing each kernel five times, if one kernel wins at least four of the head-to-head timings, it is declared the winner, otherwise they are declared to be within clock resolution.

- **srch-clk**: Clock resolution has caused the search reported in this paper to choose a less-optimal value for one or more of the empirical tuned parameters (for example, assume that **UR**=16 is optimal, but during the timing of this case unrelated load caused the timing to be inflated, and so **UR**=8 was selected instead). In order to find this kind of error, we run the search for each disputed kernel three additional times, and see if we get better results. If we do, we employ the clock resolution test to determine if it is better than the other two kernels in question (first, the previous iFKO kernel, and second, the so-far best kernel). When a rerunning of the search provides an iFKO-tuned kernel that is genuinely better than that previously reported, we summarize the new results in Table 6.8. This table of search results also includes several new columns from the previously reported tables (Tables 6.3, 6.4, 6.5 and 6.6). First, we specify the architecture (p4e/opt) and cache state (OC: all caches flushed, IC: only L1 cache flushed). The last column of the table is also new, and reports whether the new kernel was actually better than any other tuned kernel (Y), or if it ran within clock resolution of the best of the other tuned kernels (T),

or if, even after the search improvement, it is still more than clock resolution slower than the best tuned mechanism (N). To make this determination, we perform the clock resolution test comparing the new iFKO kernel against the previous best case.

- **srch-as**: Our empirical search has an error in it's assumptions. For instance, perhaps additional parameters need to be empirically tuned, or a greater range of values need to be searched for a parameter that is already empirically tuned. It is difficult to automate the detection of this loss category, but monitoring full tuning output for general trends and examining the generated assemblies by eye can help.

- Lastly, FKO can be inadequate in some way:

    1. **fko-ns**: FKO does not support a needed transformations. Can be diagnosed by examining the generated assembly, and seeing what transformation(s) the best-tuned kernel performed that FKO does not. In this case, we need to identify if the transformation can be added to FKO, and whether it will be worthwhile to do so.

    2. **fko-an**: The most well-tuned kernel used a transformation that FKO can apply, but that it failed to apply to the kernel in question because FKO's analysis was unable to determine either how to do so, or if such a transformation was legal. In this case, we would like to understand if the analysis can be expanded, and/or if the problem may be addressed using markup.

    3. **fko-ma**: FKO has misapplied known optimization(s). I.e., another method has used the same transformations as FKO, but has applied them more optimally or synergistically. In this case, we will want to examine if

this differing application is better in general, and if so, if FKO can use it as well.

One of the main purposes in categorizing these cases is determining which losses represent opportunities for learning, and thus deserve greater examination. Therefore, we do not further analyze the cases where Table 6.8 shows iFKO actually producing winning or tying results. All other loss cases are examined in detail in their own subsection. There are no in-L2 cache cases where iFKO cannot produce the most well-tuned implementation, other than `iamax`, so each of the following sections deals mainly with out-of-cache results. Section 6.6.1 describes the problems leading to the `iamax` results for all contexts and architectures, Section 6.6.2 investigates `dcopy` on the P4E and Section 6.6.3 details the issues for `dscal` on the same architecture. Finally, Section 6.6.4 investigate the `daxpy` loss on the Opteron.

6.6.1 `iamax` for All Architectures

The routine where iFKO is least effective in general is `iamax`, and the main reason for this is easily understood. This operation has a dependence distance of one, and we are currently unable to automatically vectorize it. It can, however, be legally vectorized, and the hand-tuned assembly code does so, which allows it to handily outperform iFKO's kernel in many instances. Whether or not we can discover how to auto-vectorize this loop through additional analysis, and how much if any user markup we would need to do so, is an area of research that we have not yet undertaken. Since it is not immediately obvious how to do the required analysis, we are unlikely to spend time on this problem in the immediate future, as this is the only kernel in all of the BLAS that would benefit from this fix.

The reason for the magnitude of the gap between iFKO and the hand-tuned performance is inherent in the way vectorization affects this operation. Normally, vectorization is primarily a computational optimization, which is usually a fairly

low-order term in these Level 1 operations as they are more typically constrained by the bus speed. However, `iamax` has only a single input vector, and no output vectors, and so it is less bus-bound than most. Further, unlike the other surveyed operations, `iamax` involves a branch. When this branch must be taken due to finding a new maximum, it will usually be mispredicted (as the most common case is when the new value is not larger than the current maximum), which will cause a pipeline flush. As the P4E has a 20 stage pipeline, this is a significant cost.

When `iamax` is vectorized, not only does it reduce the computation by something close to the vector length (as in most operations), *but it also decreases the number of branches executed by a similar amount.* As would be expected (due to its increased vector length), single precision shows a much larger gap between vectorized and unvectorized than double precision.

6.6.2 Pentium 4E `dcopy`

On the P4E, the iFKO-tuned kernel is significantly slower than the hand-tuned assembly. The iFKO-tuned kernel gets 205 MFLOPS, whereas the hand-tuned achieves 260. In order to understand why, we examine the implementations in question. Figure 6.9 shows the listing of the hand-tuned kernel from ATLAS, while Figure 6.10(a) shows the inner loop of iFKO's tuned kernel.

The inner loop of the hand-tuned kernel is comprised of lines 44-64 of Figure 6.9. Contrasting this with Figure 6.10(a) might lead to the idea that it is either the greater unrolling, or the differing scheduling that is providing the speedup. However, Figure 6.10(b) actually runs runs slightly slower (198 MFLOPS) than Figure 6.10(a). Note that the loops are still different, in that Figure 6.10(b) uses SSE rather than Figure 6.9's MMX, but their effective unrolling and scheduling are now the same.

The real reason for the hand-tuned kernel's substantial win is that it employs an optimization that FKO does not presently support, called *block fetch* [47]. The basic idea is to perform a given computation (in this case, a copy) in two phases. In the

```
#define nblk    %ebx
#define N       %eax
#define X       %esi
#define Y       %ecx
#define stX     %edx
#define stXF    %edi
#define NB 512
#define SH   9
.global ATL_UCOPY
ATL_UCOPY:
    subl    $16, %esp
    movl    %ebx, (%esp)
    movl    %esi, 4(%esp)
    movl    %edi, 8(%esp)
    movl    %ebp, 12(%esp)
    movl    20(%esp), N
    movl    24(%esp), X
    movl    32(%esp), Y
    movl    N, stXF
    shl $3, stXF
    addl    X, stXF
#
#   Find how many NB-size chunks
    we have got, bail if 0
#
    movl    N, nblk
    shr $SH, nblk
    jz  LOOP1

LOOPB:
#
#   Burst load X
#
    movl    X, stX
    addl    $NB*8, stX
    .align  16
BURST:
    movl    -64(stX), %ebp
    movl    -128(stX), %ebp
    subl    $128, stX
    cmp X, stX
    jne BURST
    addl    $NB*8, stX
    .align  16
LOOP8:
    movl    (X), %mm0
    movl    8(X), %mm1
    movl    16(X), %mm2
    movl    24(X), %mm3
    movl    32(X), %mm4
    movl    40(X), %mm5
    movl    48(X), %mm6
    movl    56(X), %mm7
    movntq  %mm0, (Y)
    movntq  %mm1, 8(Y)
    movntq  %mm2, 16(Y)
    movntq  %mm3, 24(Y)
    movntq  %mm4, 32(Y)
    movntq  %mm5, 40(Y)
    movntq  %mm6, 48(Y)
    movntq  %mm7, 56(Y)
    addl    $64, Y
    addl    $64, X
    cmp X, stX
    jne LOOP8
#
#   Keep going until out of
    blocks
#
    subl    $1, nblk
    jnz LOOPB

    cmp X, stXF
    je  DONE
LOOP1:
    movl    (X), %mm0
    movntq  %mm0, (Y)
    addl    $8, Y
    addl    $8, X
    cmp X, stXF
    jne LOOP1
DONE:
    sfence
    emms
    movl    (%esp), %ebx
    movl    4(%esp), %esi
    movl    8(%esp), %edi
    movl    12(%esp), %ebp
    addl    $16, %esp
    ret
```

Figure 6.9: Hand-tuned `dcopy` Assembly Routine for P4E

```
1 _LOOP_0:
2     movapd  (%ecx), %xmm0
3     movntpd %xmm0, (%eax)
4     movapd  16(%ecx), %xmm0
5     movntpd %xmm0, 16(%eax)
6     addl    $32, %ecx
7     addl    $32, %eax
8     subl    $4, %ebp
9     jg  _LOOP_0
```

(a) As generated

```
9 _LOOP_0:
10    movapd  (%ecx), %xmm0
11    movapd  16(%ecx), %xmm1
12    movapd  32(%ecx), %xmm2
13    movapd  48(%ecx), %xmm3
14    movntpd %xmm0, (%eax)
15    movntpd %xmm1, 16(%eax)
16    movntpd %xmm2, 32(%eax)
17    movntpd %xmm3, 48(%eax)
18    addl    $64, %ecx
19    addl    $64, %eax
20    subl    $8, %ebp
21    jg  _LOOP_0
```

(b) Hand-scheduled

Figure 6.10: Inner loop of iFKO-tuned P4E `dcopy`

first phase, the operands are burst loaded into cache via a series of cache line length separated loads (lines 33-41 of Figure 6.9). In the second phase (lines 44-64) the computation is performed on the data that was loaded in phase 1.

In order to bring the data into cache, the problem must be blocked or partitioned, and in this case the block factor was 512 double precision elements. The dual phases result in the two inner loops, and the blocking around these phases results in the outer loop around them (this outer loop starts on line 29 and ends on line 69).

Block fetch can be particularly effective for bus-bound operations, where prefetch cannot help (or indeed, for architectures not possessing prefetch). In normal code, loads are intermixed with computation, and there are multiple loads per cache line. This can result in poor bus utilization even when the operation is bus bound. Block fetch drives the bus at its maximal rate by issuing only one fetch per cacheline, with no delays between requests. As an additional optimization, this burst loop (lines 36-41) runs backwards (i.e., starts at the end of array and iterates to the beginning), but the fetches unrolled inside it run forward. This non-linear fetch pattern is designed to confuse the hardware prefetch unit, so that it will not issue any hardware prefetch instructions (which would represent useless overheads on most architectures).

This optimization can be added safely to a general compilation framework, and since it is one of the few techniques that can help truly bus-bound operations, and because it can be applied to any architecture (no special hardware/ISA support required), we plan to add it to to FKO.

6.6.3 Pentium 4E `dscal`

The best kernel for this routine is actually a hand-written ANSI C implementation (using inline assembly for prefetch) compiled by gcc. The performance of this kernel was reported at 219 MFLOPS, but subsequent timings showed performance more in the range of 212. However, it *does* always beat the iFKO-tuned kernel, which gets performance of around 209. Our search always applies vectorization when legal. In this case, slightly better results are obtained when vectorization is *not* applied. In fact, the iFKO-tuned kernel moves slightly ahead if we use an unvectorized kernel, with unrolling of 8, and prefetch distance of 384 bytes. This boosted FKO performance to an average of 213 MFLOPS, enough to win four out of five head-to-head comparisons with the hand-tuned code.

Therefore, we are left with the question of why the scalar code would be faster than the vectorized loop. It is difficult to provide a definitive answer, but we can certainly hazard an educated guess. Cache line elements are filled in-order, and the scalar code needs to fill less of the cache line in order to start the computation, which could result in slightly fewer stalls on the first load from a given cache line. Since this computation is completely bus-bound, vectorization's greater computational peak is of no benefit, and thus the scalar version is very slightly faster. This would explain why the effect is so small, as well as why we don't see it with less bus-bound operations.

This minor improvement does not seem to mandate additional empirical tuning, particularly as additional optimizations may render it moot (eg., perhaps with block fetch or software pipelining of the loads and stores the vector code's computational

advantage will provide a speedup over scalar), unless it is shown to be true for more architectures and operations.

6.6.4 Opteron `daxpy`

The search chooses the prefetch instruction type to use before the distance is tuned. There is obviously a dependence here, and our assumption was that the type of instruction used was more fundamental, and thus it made sense to make this decision first. However, in this case, while using `prefetchw` for the prefetch of Y at the default distance results in slower execution than using `prefetch0`, just the opposite is true once the distance has been tuned. Changing the Y prefetch instruction to `prefetchw` in the parameters given in Table 6.4 boosts performance from 234 MFLOP to 257, which would make iFKO the best tuning methodology. This points out an error in our assumptions for the search, and we will need to perform additional studies to determine the correct adjustments to make.

In the worst case, we must do a true 2-D search on these parameters: perform the distance subsearch for each supported instruction. In order to see how best to address this dependence, this should probably be implemented and tested across a range of architectures so that general trends can be determined. Only if no unifying trends can be spotted will we leave this as iFKO's default methodology, however, since the distance search is already our longest-running sub-search.

There are various trends that could lend themselves to quicker searches. For instance, if the distance is fairly independent of instruction type across architectures, it makes sense to simply reverse the decision order (i.e., search for distance first, and then instruction type). We performed this reversal, and it resulted in no change for in-cache or P4E performance, but boosted Opteron out-of-cache performance on several operations, without losing performance anywhere. So, with our present sample set, this appears be a superior ordering. Doing the search in this order sped

up the following routines out-of-cache on the Opteron, by the specified percentage: `sswap` : 9%, `saxpy` : 15%, `daxpy` 13%, and `sdot` : 4%.

There are several other options that may work better in general. For instance, we could retain the present ordering, but after the prefetch distance is tuned, the prefetch instruction tuning search is performed again, and if this results in a change, the new instruction type is substituted. If and only if a substitution is required, we can then rerun the distance search, if the trends show this is necessary. Perhaps the best approach would be to perform a crude distance search first (for instance, searching only powers of two), then tune the instruction type, followed by a more complete distance search. It may be that indeed the prefetch instruction type is a more fundamental choice, but before distance tuning, the difference between instruction types is below clock resolution, leading to essentially a random selection. In this case, a crude distance tuning should be sufficient to allow for a more accurate selection.

CHAPTER 7

FUTURE WORK, SUMMARY AND CONCLUSIONS

This final chapter provides some concluding remarks, and is organized in the following way: Section 7.1 describes some key areas for future investigation, Section 7.2 briefly recapitulates the highlights of this work, and Section 7.3 draws some conclusions from the presented studies.

7.1 Future Work

The amount of future work, in particular the number of optimizations of interest, are so extensive that discussing them in full is probably not possible. Therefore, in this section we concentrate on some broad extensions that are clearly needed, as well as the specific optimizations that have been identified as particularly beneficial based on our current results. Section 7.1.1 discusses the extensions to our optimizing compiler (FKO), while Section 7.1.2 discusses some key search issues.

7.1.1 Future Work on FKO

There are two optimizations that we believe would improve even our current results, both of which would be implemented as fundamental transformations. The first is block fetch, as discussed in Section 6.6.2, which would probably be applied before any other optimization (since this transformation results in additional loops, it would be necessary for later fundamental optimizations to tune the computation loop).

The second (short-term) transformation of interest is software pipelining, which would be applied after vectorization. While the x86's out-of-order execution and register renaming makes software pipelining the linked multiples and adds less important than on in-order architectures, software pipelining of load/use and use/store should provide more efficient implementations, even for the studied Level 1 kernels. At the same time, software pipelining dependent multiplies and adds will be critical on architectures (such as the SPARC) that are both in-order and possess separate multiply and add FPU units.

The next targets for optimization would be the Level 3 BLAS. As we have seen, all of these operations are tuned using the in-L1 gemm kernel, which is implemented using three nested loops. For efficient optimization of these operations, we will therefore want to introduce additional optimization phases, new pointer support, and extended markup opportunities. In pointer support, we need a way to indicate (or derive) when inner-loop pointers actually point to separate locations (usually rows or columns) within a single multidimensional array. Knowing that given inner loop pointers come from a single array can allow us to alleviate integer register pressure through use of CISC indexing on the x86, and this is critical on the IA-32 ISA, where insufficient integer registers would otherwise restrict outer loop unrolling to values well below the optimal.

We will almost certainly want to add an outer-loop markup that allows the user to suggest and limit outer-loop unrolling. The type of outer loop unrolling we are interested in is called "unroll and jam" [48], as unrolling of the outer loop(s) results in issuing more instructions in the single inner loop, not, for instance, duplicating the inner loop, resulting in multiple inner loops. Along with unroll and jam, we will need a repeatable transformation similar to scalar replacement [24] in order to enable register blocking.

Our present prefetch strategies always assume that the data being prefetched will eventually be used during the loop iterations. In GEMM, it is often the case that it

is more efficient to fetch the next cache block while operating on this one. Therefore, we will want to introduce prefetching of unrelated memory during computation.

Finally, in GEMM, once we support unroll and jam, the inner loop (which on the x86 will always be vectorized) acquires multiple accumulators. We presently reduce any vector accumulators to scalars individually in the inner loop epilogue. We can optimize this process when multiple accumulators are being used, and since this epilogue code is now nested inside outer loops (and it is usually not the case that it can be pushed out of them), it becomes critical to do so. We will similarly need to make copy propagation more efficient in handling scalar-to-vector conversions at the beginning and end of the loops, as highlighted in the **SV** example given in Section 5.7.1.1.

While it will not be critical for our most important Level 3 kernel, as we deal with more deeply nested loops, it will probably become advantageous to add generalized loop invariant code motion, in order to hoist/push all operations (rather than just loads and stores) as far out of the loops as possible.

Once iFKO can fully tune the Level 3 BLAS, it will be time to consolidate some of our prexisting support. This includes handling misalignment for SIMD vectorization, complex type support, and additional architectures, all discussed in turn below.

We have previously discussed misalignment in detail, and we will proceed with this work as outlined in Section 5.5. Once we have support for exploiting alignment guarantees based on 2-D array usage, iFKO will be ready to tune the Level 2 BLAS.

We will certainly not examine other architectures in detail until we can convincingly tune both the Level 1 and 3 BLAS, as previously described. Only at this stage will it make sense to extend our architecture support, and we will examine the PowerPC architecture next. This may involve additional optimization support, and will certainly require tuning various presently-supported phases. For instance, SIMD vectorization needs to be ported to support the PowerPC's vector unit, AltiVec. We

will also want to examine using the PowerPC's specialized index register for **LC** (optimize loop control).

Just as with ANSI C, our HIL does not presently support complex numbers. Of course, complex kernels may be written in terms of real computations, but this is inconvenient for the implementer. Therefore, it makes sense to add a complex type. Note that this is not needed for Level 3 BLAS support, as ATLAS uses the real kernel to tune the complex case, as discussed in Chapter 3. Complex support will, however, help with the Level 1 and 2 BLAS support. Complex arithmetic is composed of a series of dependent real arithmetic operations, and since these real operations have a dependence distance of one, they can be a barrier to SIMD vectorization if they are not handled appropriately. The SSE3 ISA extension added instructions specifically designed to handle complex arithmetic without unneeded permutation or redundant computation, so FKO will need to exploit SSE3 to avoid these overheads. In order to enable this SIMD optimization, it seems likely that the front end will generate synthetic LIL instructions which are placeholders for complex arithmetic. In **SV** these synthetic instructions will then be substituted with the appropriate SSE3 instructions, or if **SV** is not applied, a new fundamental transformation phase would replace them with the appropriate real computations.

7.1.2 Future work on iFKO's Search

Section 6.6.4 pointed out the need for a better approach to prefetch instruction selection, and this will be the first area of work for the search. We have also seen that clock resolution problems have caused substandard results to be issued, and thus it makes sense to examine if the timings can be made more precise. More fundamentally, the addition of unroll and jam will provide several dimensions of dependent optimizations (eg., for matrix multiply, unrolling the two outer loops strongly changes the inner loop). In these cases, we must determine if we will be forced to employ a full multidimensional search in order to get robust results, or if

we can instead make simplifying assumptions which allow us to severely restrict the interactions. If we cannot find such simplifying assumptions, it becomes very likely that we will have to abandon the line search for a more advanced technique that can optimize the search of such a high dimensional space, and both simulated annealing and genetic algorithm are promising candidates.

7.2 Summary

In the introduction, we discussed the importance of performance tuning for high performance computing, and highlighted the key weaknesses inherent in traditional methodologies. We then described how empirical techniques, embodied in the AEOS concept, have proven to be a successful response to these challenges. Chapter 3 then described our first AEOS effort, the empirically tuned library generator ATLAS. This pioneering research has proven to be extremely successful, in both research aims and practical use. ATLAS-tuned libraries are used by a worldwide audience of scientists, engineers, and educators every day. The success of this project has inspired a great deal of related research, and as a result the ATLAS papers are highly cited in the literature (in both high performance computing, and more recently, compilation research).

The following chapters described the more generalized research we have undertaken recently, embodied in our empirical compilation framework, iFKO. Chapter 4 described the basic ideas behind this work, and the design philosophy we utilize to guide and prioritize our efforts, with Chapter 5 filling in the details of our current implementation of this framework. Finally, Chapter 6 discussed the results we have achieved in applying the current framework to the Level 1 BLAS.

7.3 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 demonstrated 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 (a large component of our future work) 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 [49], 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.

APPENDIX

ANSI C AND HIL KERNEL IMPLEMENTATIONS

This appendix provides the ANSI C and HIL implementations for each studied routine. We show the double precision version (the single precision is the same with the appropriate variable declarations changed). Figures [A.1, A.2, A.3, A.4, A.5, A.6, A.7] show [`dswap`, `dcopy`, `dasum`, `daxpy`, `ddot`, `dscal`, `idamax`], respectively.

```
void ATL_USWAP(const int N,
               double *X, const int incX,
               double *Y, const int incY)
{
   int i;
   double tmp;
   for (i=0; i < N; i++)
   {
      tmp = Y[i];
      Y[i] = X[i];
      X[i] = tmp;
   }
}
```

(a) ANSI C

```
ROUTINE ATL_USWAP;
   PARAMS :: N, X, incX, Y, incY;
   INT    :: N, incX, incY;
   DOUBLE_PTR :: X, Y;
ROUT_LOCALS
   INT    :: i;
   DOUBLE :: x, y;
ROUT_BEGIN
   LOOP i = 0, N
   LOOP_BODY
      x = X[0];
      y = Y[0];
      X[0] = y;
      Y[0] = x;
      X += 1;
      Y += 1;
   LOOP_END
ROUT_END
```

(b) HIL

Figure A.1: `dswap` implementations

```
void ATL_UCOPY(const int N,
               const double *X,
               const int incX,
               double *Y, const int incY)
{
   int i;
   for (i=0; i < N; i++)
      Y[i] = X[i];
}
```

(a) ANSI C

```
ROUTINE ATL_UCOPY;
   PARAMS :: N, X, incX, Y, incY;
   INT    :: N, incX, incY;
   DOUBLE_PTR :: X, Y;
ROUT_LOCALS
   INT    :: i;
   DOUBLE :: x;
ROUT_BEGIN
   LOOP i = 0, N
   LOOP_BODY
      x = X[0];
      Y[0] = x;
      X += 1;
      Y += 1;
   LOOP_END
ROUT_END
```

(b) HIL

Figure A.2: `dcopy` **implementations**

```
double ATL_UASUM(const int N,
                 const double *X,
                 const int incX)
{
   int i;
   register double t0=0.0;
   for (i=0; i < N; i++)
      t0 += fabs(X[i]);
   return(t0);
}
```

(a) ANSI C

```
ROUTINE ATL_UASUM;
   PARAMS :: N, X, incX;
   DOUBLE_PTR :: X;
   INT :: N, incX;
ROUT_LOCALS
   INT :: i;
   DOUBLE :: x, sum;
   CONST_INIT :: sum = 0.0;
ROUT_BEGIN
   LOOP i = 0, N
   LOOP_BODY
      x = X[0];
      x = ABS x;
      sum += x;
      X += 1;
   LOOP_END
   RETURN sum;
ROUT_END
```

(b) HIL

Figure A.3: `dasum` **implementations**

```
void ATL_UAXPY(const int N,
               const double alpha,
               const double *X,
               const int incX,
               double *Y, const int incY)
{
   int i;
   for (i=0; i < N; i++)
      Y[i] += alpha * X[i];
}
```

(a) ANSI C

```
ROUTINE ATL_UAXPY;
   PARAMS :: N, alpha, X, incX, Y, incY;
   INT    :: N, incX, incY;
   DOUBLE :: alpha;
   DOUBLE_PTR :: X, Y;
ROUT_LOCALS
   INT    :: i;
   DOUBLE :: x, y;
ROUT_BEGIN
   LOOP i = 0, N
   LOOP_BODY
      x = X[0];
      y = Y[0];
      x = x * alpha;
      y += x;
      Y[0] = y;
      X += 1;
      Y += 1;
   LOOP_END
ROUT_END
```

(b) HIL

Figure A.4: `daxpy` implementations

```
double ATL_UDOT(const int N,
                const double *X,
                const int incX,
                const double *Y,
                const int incY)
{
   register double dot=ATL_rzero;
   int i;
   for (i=0; i < N; i++)
      dot += X[i] * Y[i];
   return(dot);
}
```

(a) ANSI C

```
ROUTINE ATL_UDOT;
   PARAMS :: N, X, incX, Y, incY;
   INT    :: N, incX, incY;
   DOUBLE_PTR :: X, Y;
ROUT_LOCALS
   INT    :: i;
   DOUBLE :: x, y, dot;
   CONST_INIT :: dot = 0.0;
ROUT_BEGIN
   LOOP i = 0, N
   LOOP_BODY
      x = X[0];
      y = Y[0];
      dot += x * y;
      X += 1;
      Y += 1;
   LOOP_END
   RETURN dot;
ROUT_END
```

(b) HIL

Figure A.5: `ddot` implementations

```
void ATL_USCAL(const int N,
               const double alpha,
               double *X,
               const int incX)
{
   int i;
   for (i=0; i < N; i++)
      X[i] *= alpha;
}
```

(a) ANSI C

```
ROUTINE ATL_USCAL
   PARAMS :: N, alpha, X, incX;
   INT :: N, incX;
   DOUBLE :: alpha;
   DOUBLE_PTR :: X;
ROUT_LOCALS
   INT :: i;
   DOUBLE :: x, y;
ROUT_BEGIN
   LOOP i = 0, N
   LOOP_BODY
      x = X[0];
      x = x * alpha;
      X[0] = x;
      X += 1;
   LOOP_END
ROUT_END
```

(b) HIL

Figure A.6: `dscal` **implementations**

```
int ATL_UIAMAX(const int N,
               const double *X,
               const int incX)
{
   register double xmax, x0;
   int i, iret=0;
   if (N > 0)
   {
      xmax = *X;
      xmax = fabs(xmax);
      for (i=1; i < N; i++)
      {
         x0 = X[i];
         x0 = fabs(x0);
         if (x0 <= xmax) continue;
         else
         {
            xmax = x0;
            iret = i;
         }
      }
   }
   return(iret);
}
```

(a) ANSI C

```
ROUTINE ATL_UIAMAX;
   PARAMS :: N, X, incX;
   INT    :: N, incX;
   DOUBLE_PTR :: X;
ROUT_LOCALS
   INT    :: i, imax;
   DOUBLE :: x, amax;
   CONST_INIT :: amax = 0.0, imax=0;
ROUT_BEGIN
   LOOP i = N, 0, -1
   LOOP_BODY
      x = X[0];
      x = ABS x;
//    Branch if we have a new maximum
      IF (x > amax) GOTO NEWMAX;
ENDOFLOOP:
      X += 1;
   LOOP_END
   RETURN imax;

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

(b) HIL

Figure A.7: `idamax` **implementations**

REFERENCES

[1] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the ACM SIGARC International Conference on SuperComputing*, Vienna, Austria, July 1997.

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

[3] M. Frigo and S. G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, 1997.

[4] 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.

[5] R. Clint Whaley and Antoine Petitet. Atlas homepage. `http://math-atlas.sourceforge.net/`.

[6] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. `http://www.netlib.org/lapack/lawns/lawn131.ps`.

[7] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.** `http://www.cs.utk.edu/~rwhaley/papers/atlas_sc98.ps`.

[8] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[9] R. Clint Whaley, Antoine Petitet, and Jack 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 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[10] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. Accepted for publication in *Software: Practice and Experience*, 2004. `http://www.cs.utk.edu/~rwhaley/papers/spercw04.ps`.

[11] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vudue, R. Clint Whaley, and Katherine Yellick. Self adapting linear algebra algorithms and software. Accepted for putblication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

[12] R. Hanson, F. Krogh, and C. Lawson. A Proposal for Standard Linear Algebra Subprograms. *ACM SIGNUM Newsl.*, 8(16), 1973.

[13] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[14] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[15] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[16] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[17] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.

[18] B. Kågström, P. Ling, and C. van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. Technical Report UMINF 95-18, Department of Computing Science, Umeå University, 1995. Submitted to ACM TOMS.

[19] M. Dayde, I. Duff, and A. Petitet. A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors. *ACM Transactions on Mathematical Software*, 20(2):178–193, 1994.

[20] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and blas's for dense linear algebra algorithms. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 195–206, 1998.

[21] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Superscalar gemm-based level 3 blas – the on-going evolution of a portable and high-performance library. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 207–215, 1998.

[22] J. Dongarra, P. Mayes, and G. Radicati di Brozolo. The IBM RISC System 6000 and linear algebra operations. *Supercomputer*, 8(4):15–30, 1991.

[23] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 53–65, 1990.

[24] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[25] R. Clint Whaley. User contribution to atlas. `http://math-atlas.sourceforge.net/devel/atlas_contrib/`.

[26] J. Moura, J. Johnson, R. Johnson, D. Padua, M. Puschel, and M. Veloso. Spiral: Automatic implementation of signal processing algorithms. In *Proceedings of the Conference on High-Performance Embedded Computing*, MIT Lincoln Laboratories, Boston, MA, 2000.

[27] Pedro Diniz, Yoon-Ju Lee, Mary Hall, and Robert Lucas. A case study using empirical optimization for a large, engineering application. In *International Parallel and Distributed Processing Symposium*, 2004. CD-ROM Proceedings.

[28] Bas Aarts, Michel Barreteau, Francois Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John R. Gurd, Jan Hoggerbrugge, Ping Hu, William Jalby, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, Erven Rohou, Rizos Sakellariou, Henk Schepers, Andre Seznec, Elena Stohr, Marco Verhoeven, and Harry A. G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In *European Conference on Parallel Processing*, pages 1351–1356, 1997.

[29] Toru Kisuki, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, Francois Bodin, and Harry A. G. Wijshoff. A feasibility study in iterative compilation. In *ISHPC*, pages 121–132, 1999.

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

[31] M. O'Boyle, N. Motogelwa, and P. Knijnenburg. Feedback assisted iterative compilation. In *LCR*, 2000.

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

[33] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline – unrolling tradoffs. In *SCOPES99*, 1999.

[34] J.M.F. Moura, J. Johnson, R.W. Johnson, D. Padua, V. Prasanna, M. Pschel, and M.M. Veloso. Spiral: Automatic library generation and platform-adaptation for dsp algorithms, 1998. http://www.ece.cmu.edu/~spiral.

[35] Markus Pushel, Jose Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Frenchetti, Aca Cacic, Yevgen Voronenko, Kang Chen, Robert Johnson, and Nick Rizzolo. Spiral: Code generation for dsp transforms. Accepted for putblication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

[36] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph Ueberhuber. Efficient utilization of simd extensions. Accepted for putblication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

[37] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization*, pages 204–215, 2003.

[38] System V Application Binary Interface, Intel386 Architecture Processor Supplement. URL: http://www.caldera.com/developers/devspecs/abi386-4.pdf.

[39] Richard Detmer. *Introduction to The 80x86 Assembly Language and Computer Architecture*. Jones and Bartlett Publishers, Sudbury, MA, 2001.

[40] Jan Hubicka, Andreas Jaeger, and Mark Mitchel. System V Application Binary Interface, AMD64 Architecture Processor Supplement. http://www.x86-64.org/documentation/abi-0.92.pdf.

[41] Jan Hubicka, Andreas Jaeger, and Mark Mitchel. Software optimization guide for amd athlon 64 and opteron processors. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF, March 2004.

[42] Intel extended memory 64 technology. URL: http://www.intel.com/technology/64bitextensions/index.htm.

[43] Ed Sznyter and Babel Press. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2nd edition, 1994.

[44] Gary Kacmarcik. *Optimizing PowerPC Code*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[45] David Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994. URL: http://www.sparc.com/standards/SPARCV9.pdf.

[46] System V Application Binary Interface, SPARC Processor Supplement. URL: http://www.sparc.com/standards/psABI3rd.pdf.

[47] Mike Wall. Using Block Prefetch for Optimized Memory Performance. Technical report, Advanced Micro Devices, 2002. http://cdrom.amd.com/devconn/events/AMD_block_prefetch_paper.pdf.

[48] Steve Carr, Chen Ding, and Philip H. Sweany. Improving software pipelining with unroll-and-jam. In *HICSS (1)*, pages 183–192, 1996.

[49] Kamen Yotov, Xiaorning Li, Gang Ren, Maria Garzaran, Dvaid Padua, Keshav Pingali, and Paul Stodghill. A comparison of empirical and model-driven optimization. Accepted for publication in *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.

BIOGRAPHICAL SKETCH

R. Clint Whaley

The author was born on November 9, 1969, and received his B.S. in Mathematics (Summa Cum Laude) from Oklahoma Panhandle State University in May of 1991. He received his Master of Science in Computer Science in May of 1994 from the University of Tennessee, Knoxville (UTK), where his thesis dealt with communication on distributed memory systems. His professional career began with work at Oak Ridge National Laboratories, where he worked as a research student (1990-1991) on parallelizing (for distributed memory machines) nuclear collision models in the physics division. From May 1994 through June 1999, he was employed as a full-time researcher (Research Associate) at UTK. From June 1999 through December 2001, he was a Senior Research Associate at UTK. During his years at UTK, he worked on the well-known parallel package ScaLAPACK. Later, as a full time researcher, he founded the ongoing ATLAS research project, and ATLAS-tuned libraries are used by scientists and engineers around the world. His research interests include code optimization, compilation research, high performance computing, and parallel computing.