ADAPTIVE DYNAMIC SCHEDULING OF FFT ON HIERARCHICAL MEMORY AND MULTI-CORE ARCHITECTURES

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Ayaz Ali

May 2008

ADAPTIVE DYNAMIC SCHEDULING OF FFT ON HIERARCHICAL MEMORY AND MULTI-CORE ARCHITECTURES

Ayaz Ali

APPROVED:

Dr. S. Lennart Johnsson, Chairman
Dept. of Computer Science

Dr. Jaspal Subhlok
Dept. of Computer Science

Dr. Barbara Chapman
Dept. of Computer Science

Dr. Dragan Mirkovic
MD Anderson Cancer Center, University of Texas

Dr. John Mellor-Crummey
Dept. of Computer Science, Rice University

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I am grateful to my advisor, Dr. Lennart Johnsson, for his invaluable guidance and support that made this work possible. His knowledge and enthusiasm motivated me at every stage of my research endeavor.

I also want to express my gratitude to Dr. Jaspal Subhlok and Dr. Dragan Mirkovic for their guidance, without which this project would never have reached fruition. I would also like to thank Dr. John Mellor-Crummey, Dr. Barbara Chapman and late Dr. Ken Kennedy for serving on my dissertation advisory committee. Their comments and suggestions were very helpful in improving the quality of this work.

Many thanks to Erik Engquist, Mark Huang, Bo Liu, Peter Tang and Greg Henry for participating in the discussions and for sharing their experiences on several occasions.

Support from the Texas Learning and Computation Center (TLCC), the Texas Advanced Computing Center (TACC), the TAMU supercomputing facility, the National Supercomputing Center (NSC) at Linkoping, Sweden is hereby gratefully acknowledged.

I am grateful to my wife, Ambreen, for her patience and interest in my PhD adventure. Above all, I wish to thank my parents, who helped me attain my goals through their aspirations and prayers.

ADAPTIVE DYNAMIC SCHEDULING OF FFT ON HIERARCHICAL MEMORY AND MULTI-CORE ARCHITECTURES

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Ayaz Ali

May 2008

Abstract

In this dissertation, we present a framework for expressing, evaluating and executing dynamic schedules for FFT computation on hierarchical and shared memory multiprocessor / multi-core architectures. The framework employs a two layered optimization methodology to adapt the FFT computation to a given architecture and dataset. At installation time, the code generator adapts to the microprocessor architecture by generating highly optimized, arbitrary size micro-kernels using dynamic compilation with feedback. At run-time, the micro-kernels are assembled in a DAG-like schedule to adapt the computation of large size FFT problems to the memory system and the number of processors.

To deliver performance portability across different architectures, we have implemented a concise language that provides specifications for dynamic construction of FFT schedules. The context free grammar (CFG) rules of the language are implemented in various optimized driver routines that compute parts of the whole transform. By exploring the CFG rules, we were able to dynamically construct many of the already known FFT algorithms without explicitly implementing and optimizing them. To automate the construction of best schedule for computing an FFT on a given platform, the framework provides multiple low cost run-time search schemes. Our results indicate that the cost of search can be reduced drastically through accurate prediction and estimation models.

With its implementation in the UHFFT, this dissertation provides a complete methodology for the development of domain specific and portable libraries. To validate our methodology, we compare the performance of the UHFFT with FFTW

and Intel's MKL on recent architectures - Itanium 2, Xeon Clovertown and a second generation Opteron. Our optimized implementations of various driver routines compare favorably against the FFTW and MKL libraries. Our experiments show that the UHFFT outperforms FFTW and MKL on most architectures for problems too large to fit in cache. Moreover, our low-overhead multithreaded driver routines deliver better performance on multi-core architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The large gap between the speed of processors and main memory that has developed over the last decade and the resulting increased complexity of memory systems to ameliorate this gap has made it increasingly harder for compilers to optimize an arbitrary code within an acceptable amount of time. Though issues of heat removal have considerably slowed the rate of increase in processor clock frequencies, the industry response to continue delivering higher performance is chips with an increasing number of processor "cores" (multi-core chips), which further adds to the complexity of code optimization. There is a need for the development of domain specific, high performance frameworks that aid the compilers through generation of optimized compute-intensive kernels for performance critical applications.

The fast Fourier transform is one such application that has gained importance in many fields of science. In fact, the FFT algorithm is reported to be one of the top ten algorithms of the century [19]. A considerable research effort has been devoted to optimization of FFT codes over the past four decades. The algorithm presented by Cooley and Tukey [17, 37], reduced the algorithm complexity of computing the DFT from $O(n^2)$ to $O(n \log n)$, which is viewed as a turning point for applications of the Fourier transform. For FFT to remain an effective computation tool, it is important to have efficient implementations for modern computer systems. The FFT continues to present serious challenges to high performance library developers for every new generation of computer architectures due to its relatively low ratio of computation per memory access and non-sequential memory access pattern. Due to these unique features, scheduling FFT computation on modern platforms, given their complex architectures, is a serious challenge compared to the simple data reference and control structure of Basic Linear Algebra Subroutines (BLAS).

Current state-of-the art scientific codes use re-usable components and a layered design to adapt to the computer architecture. At the lower level, the components may use highly optimized sets of parameterized codes (micro kernels) that are automatically generated to adapt to microprocessor architecture. At the higher level, the parameterization allows a search strategy to determine the optimal assembly of kernels to make the most effective use of memory bandwidth for larger problem sizes. A search scheme can be driven by a model or it can be empirical [65]. Most auto-tuning libraries including ATLAS [18], FFTW [25] and SPIRAL [44] employ empirical search favoring performance over one-time search cost. Although empirical

search generally does a better job at finding the fastest code schedule for computing, it may take orders of magnitude more time than the actual execution of code depending on the number and size of tuning parameters. For FFT, search may require enumerating and evaluating an exponential number of combinations of factorizations and algorithms.

Although the FFT has been a well-studied topic since its publication in 1965, its progress has been retarded by lack of a standard notation and API - to express the FFT computation in a concise language that is easy to understand for computer scientists. "The simplicity and intrinsic beauty of many FFT ideas is buried in research papers that are rampant with vectors of subscripts, multiple summations, and poorly specified recursions" [37]. Indeed, this has led to poor understanding of the correlation between a FFT formula and its performance on modern architectures.

1.2 Contributions of the Dissertation

The goal of this dissertation is to address the challenges described above by developing a framework for expressing, evaluating and executing dynamic schedules for FFT computation on hierarchical and shared-memory multiprocessor architectures. To support this dissertation, we redesigned the UHFFT library [40, 39, 5, 7, 6] to deliver higher performance across a range of modern architectures. The UHFFT consists of four main components; a code generator, a formal language to express and construct FFT algorithms, an FFT executor containing serial and parallel driver routines and a scheduler that implements multiple run-time search and estimation

methods to find the best schedule of execution. In the following, we list the main contributions of this dissertation.

a. Code Generator

We have implemented an automatic empirical optimization mechanism in the FFT code generator, which has been integrated in the UHFFT library. The code generator was enhanced to automatically generate arbitrary size micro-kernels, including the so called *twiddle* codelets, rotated PFA codelets, coupled in-place codelets and loop-vectorized codelets. Additionally, the code generator also supports generations of short vector (SIMD) variants of all the codelet types. To our knowledge this is the only code generator that generates optimized, arbitrary size rotated codelets that can be used as part of the prime factor algorithm (PFA). Moreover, unlike the approach taken in the FFTW, our twiddle codelets multiply more twiddle factors per function call resulting in fewer calls to the expensive twiddle codelets.

b. The FFT Schedule Specification Language

The code generated at installation time is assembled together at run-time to solve large FFT problems. We have implemented a concise language based on simple context free grammar rules for specifying a variety of FFT schedules and algorithms. A similar effort has been implemented in the SPIRAL code generator system; their signal processing language (SPL) provides a flexible encoding for manipulating the

mathematical algorithms. The formula generated in SPL goes through multiple optimization stages to generate the best schedule for computing the given problem. On the contrary, the FFT schedule specification language (FSSL) provides a dynamic mechanism to express the schedule, including the formula and most importantly the blocking of FFT computation. This makes FSSL particularly well-suited to hierarchical memory and multi-core architectures since it specifies both the schedule and blocking of the computation. The language also helps in better understanding the correlation between an FFT schedule (algorithm and factorization) and its performance on the given architecture, which helps in the development of intuitive models (heuristics) that can be implemented in run-time search schemes to avoid expensive empirical analysis.

c. Serial and Parallel FFT Driver Routines

In the UHFFT, we have implemented efficient serial and parallel FFT driver routines for each of the FSSL rules. Our implementations of various driver routines compare favorably against the other popular FFT libraries - FFTW and Intel's MKL. Unlike the other two libraries, the UHFFT also implements prime factor algorithm (PFA) driver routines, which perform particularly well on the non-powers-of-two sizes that can be factorized in co-prime factors. Our experiments show that the UHFFT outperforms FFTW and MKL on most architectures for problems too large to fit in cache. Moreover, our low-overhead multithreaded driver routines deliver better performance on multi-core architectures.

d. Multiple Run-time Search Methods

The low cost search methods are particularly useful in libraries like the UHFFT and FFTW, which perform run-time optimizations. Traditionally, the libraries have performed such optimizations in a pre-computation stage that may take orders of magnitude longer to find the best schedule (plan) than to execute it. In this dissertation we have implemented three low cost run-time search methods in the scheduler to automate the process of constructing the best schedule. We show that the expensive run-time empirical search can be avoided by generating domain specific models (offline) without compromising the quality of schedules generated. Our preliminary results indicate that the model driven search can be accurate in predicting the best performing FFT schedule for a minimum cost of initialization.

e. Methodology for the Development of Domain Specific Adaptive Libraries

The methodology of this dissertation has been validated by developing a set of tools, a software library and by performing benchmarks on modern CPU architectures including SMPs using our FFT package, the UHFFT. Although the tool pertains to the FFT domain, the methodology presented in this dissertation may be employed in the development of other domain specific and portable libraries.

1.3 Dissertation Outline

This dissertation is organized as follows. In Chapter 2, we present background material that discusses the current trends in architectures, compiler technology and automatic tuning methodologies. Chapter 3 discusses the related work and presents an overview of the fast Fourier transform. In Chapter 4, we present the design and specifications of the UHFFT library. The chapter also describes the testbeds used for benchmarking. Chapter 5 presents the automatic code generation and optimization strategy implemented in the UHFFT library. We also evaluate the generated code on a variety of architectures. Chapter 6 presents a formal language for the construction of dynamic execution schedules for FFT computation on recent CPU and shared-memory architectures. In Chapter 7, we describe the run-time search strategies for finding the best computation schedule of a given FFT problem on the target architecture. We also compare the performance results of the UHFFT with that of other adaptive FFT libraries on various architectures. Finally in Chapter 8, we outline the conclusions and contributions of this work and discuss future work.

Chapter 2

Background

This chapter presents background material for the ideas presented in later chapters. We present a survey of the trends in computer architectures, optimizing compilers and automatic tuning methodologies. The purpose of this chapter is to refresh the fundamental concepts in the above-mentioned topics. For a full treatment of these topics, we refer the reader to two excellent books on computer architecture [43] and optimizing compilers [35].

2.1 Modern Computer Architectures

We have seen tremendous growth in the processor speed since the inception of microprocessor based computers in the late 1970s. The growth has been largely sustained by the decreasing size of transistors, increased clock rates and our ability to put more transistors on a chip. The increased number of transistors has in part been used to

enhance flexibility and predicative execution capacities of processors. The resulting complex architectures have evolved from scalar uniprocessors to superscalar multi-core architectures capable of executing multiple instructions out-of-order on each core.

2.1.1 Instruction-level Parallelism

Microprocessors in the early 1980s were relatively simple architectures called scalar processors. The scalar architectures issued and executed one instruction at a time on a single CPU. However, they were soon replaced by more complicated superscalar processors with multiple pipelined functional units capable of issuing and executing multiple instructions at a time as shown in Figure 2.1. This type of parallelism

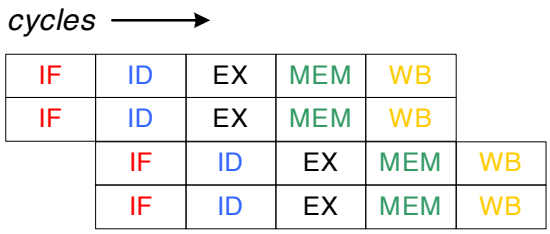


Figure 2.1: Simplified view of a superscalar pipeline execution

is known as instruction-level parallelism (ILP). Most modern processors are superscalar processors with deep pipelines capable of executing multiple instruction per machine cycle. However, dependences between instructions make it very difficult for a processor to achieve its theoretical peak performance. Figure 2.2a shows a kind of data dependence, which causes the pipeline to stall until a prior instruction is

```
LD    R1, 0(R2)          LD    R1, 0(R2)
ADD   R3, R1, R4         LD    R5, 0(R6)
LD    R5, 0(R6)          ADD   R3, R1, R4
ADD   R7, R5, R4         ADD   R7, R5, R4
```

(a) Original sequence (b) Optimized sequence

Figure 2.2: An example of instruction scheduling to avoid single pipeline stalls

complete. Notice that the ADD instruction can not execute until LD operation completes. In many cases, the independent instructions could be scheduled to avoid stalls as shown in Figure 2.2b. Indeed, most superscalar processors rely on special hardware to dynamically schedule instructions at run-time. Since the resulting execution order of instructions is different from the actual schedule, it is also known as "out-of-order" execution. Some modern processors, known as explicitly parallel instruction computer (EPIC) employ static (compile time) scheduling to package independent (parallel) instructions in packets called very long instruction words (VLIW).

SIMD: The single instruction single data (SISD) superscalar architectures exploit ILP to perform multiple instructions in parallel. On the contrary, the single instruction multiple data (SIMD) is a form of data parallelism that is used to perform identical operations on multiple data. Most general purpose processors support some type of SIMD extensions because of the performance benefit possible on well structured data (stream), which is common in multimedia and image processing applications. Compared to vector processors, today's microprocessor architectures are constrained

by such features as the length of vector registers, packing/unpacking of data and vector memory access. Due to these difficulties, most compilers are only moderately successful in generating optimized SIMD ("simdized") code.

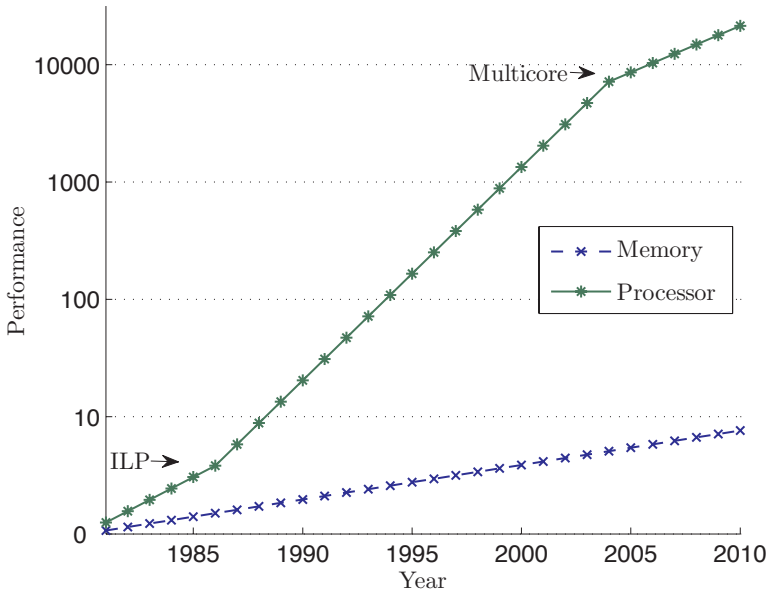


Figure 2.3: Starting with 1980 as a baseline, the gap in the performance (inverse of latency) of memory and processor has been growing with time[43]

2.1.2 Hierarchical Memory

Even though today's processors can theoretically execute billions of instructions per second, the main memory system is not capable of supplying (or receiving) data at such a rate. While the speed of processors has been growing rapidly, the improvement in memory latency and bandwidth has been much slower. The diverging rates of improvement in the main memory and processing speeds as shown in Figure 2.3,

have created a performance bottleneck in many scientific applications. This barrier, commonly referred to as the *memory wall*, has continued to present serious challenges to the high performance computing (HPC) community. In an effort to ameliorate these latencies, memory systems are organized in a hierarchy on most modern day computers as shown in Figure 2.4. The level of memory that is closest to the pro-

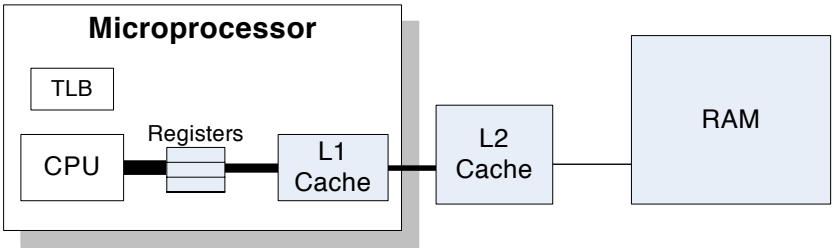


Figure 2.4: Memory hierarchy

cessor, is the fastest and also the smallest. This allows for the working dataset to be fetched in faster levels of memory thus hiding the long latency to main memory. This also improves the effective memory bandwidth for the dataset that is reused from faster levels. As a consequence of deeper and complicated memory hierarchies, depending entirely on compiler technology for optimization of code written without regard to underlying architecture is often not realistic. Developers of high performance software often need to produce "architecture aware" source code, for standard compilers to produce high quality codes, or use special purpose compilers for critical functions. Fortunately, most programs exhibit locality of reference, i.e., recently accessed items (data and instructions) are likely to be accessed again in the near future (*temporal locality*) and items whose addresses are near one another tend to be referenced close together (*spatial locality*). A typical hierarchical memory design

consists of registers, two to three levels of cache and an address mapping structure called translation lookaside buffer (TLB).

Register: In a typical RISC (reduced instruction set computer) architecture, registers act as a placeholder for operands before the operation can be executed. Registers are closest to processor and a load/store operation typically takes one clock cycle. Most modern processors have multiple data paths between the registers and the functional units enabling concurrent loads and stores. There are separate registers for floating point and general purpose instructions with total capacity generally in the range of 1KB to 1MB.

Cache: In a hierarchical memory architecture, registers are backed by multiple levels of cache, starting with on-chip cache called L1 (level one) cache. Cache memory acts as a temporary storage for datasets brought in from slower levels of memory. Most architectures have two levels of cache with sizes in the range of few MBs. However, some high-end computers use upto three levels of cache with L3 size in tens of MBs. When a processor references a memory location, it is first searched in cache starting with the fastest (L1) to slowest level (L2/L3). If a data item is found, the reference is called a *cache hit* and the memory access is completed in a few clock cycles. However, if the data item is not present in any level of cache, a *cache miss* occurs. The requested item is fetched from the main memory in a fixed size block called *cache line*. This block not only contains the requested item but a few adjacent items as well; thus maintaining spatial locality. The efficiency of a cache design also depends on how the fetched block is mapped in the cache. Because when

a cached item is referenced, it needs to be located in the cache for retrieval. Most caches implement a (n-way) set-associative design, which maps the block addresses to restricted locations using modulo arithmetic:

$$(Block\,address)\ |\ (Number\,of\,sets\,in\,cache)$$
$$\Rightarrow (Block\,address)\ |\ (\tfrac{Number\,of\,blocks\,in\,cache}{associativity})$$

In a simple *direct mapped* cache (1-way set associative), the block can be mapped to exactly one location. In a *fully associative* cache (n-way set) , the block can be placed in any of the available n blocks, which makes the retrieval more expensive. Caches are typically organized in a four and eight way set associative design. Apart from the cache associativity, the write policy is also an important distinguishing factor in the design of caches. There are two possible options when writing to the cache [43]. In *write through* (WT) cache, when a write occurs, the data is written to both cache and lower level memory. On the contrary, *write back* (WB) cache employs lazy protocol, i.e, the modified cache block is written to main memory only when it is replaced.

Translation Lookaside Buffer (TLB): Most architectures use virtual memory to let multiple programs run in their own address space. Each process usually sees more memory than the size of available physical memory. In such a scenario, an address translation mechanism is provided through a page table that maps the virtual addresses to physical memory. The page table resides in main memory so accessing the page table for every memory reference can be quite expensive. Therefore, following the principle of locality, address translations are kept (cached) in a special cache

called translation lookaside buffer (TLB) as shown in Figure 2.4.

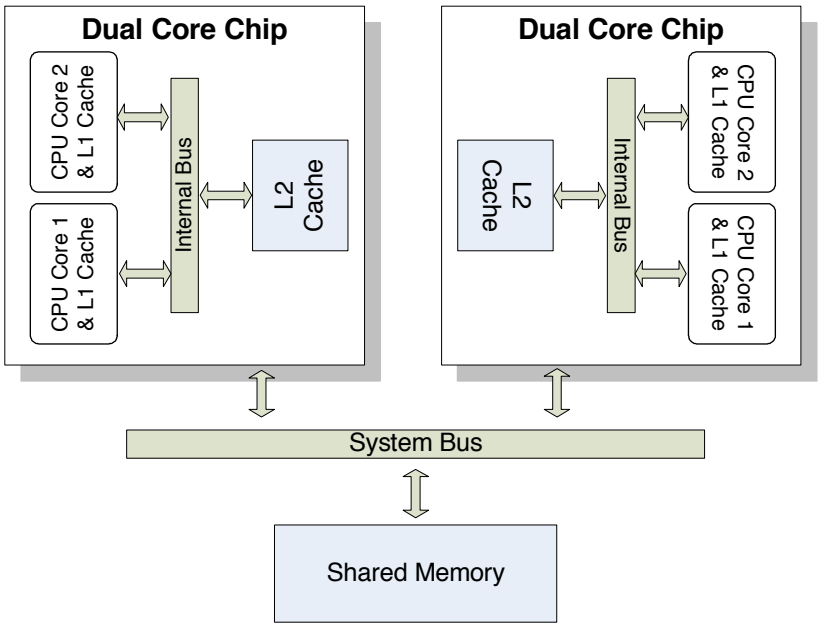


Figure 2.5: Shared-memory multiprocessing

2.1.3 Shared-memory Parallel Architectures

Although uniprocessor technology continued to advance through the 90s, issues related to heat dissipation and power requirement grew worse in the early 21st century. Significant improvement in processor speeds was no longer possible by increasing the clock rate. Moreover, increased complexity of processors also adds to the power consumption. This barrier is commonly known as the *power wall*. The industry response to this problem has been multi-core chips, which has extended the life of "Moore's law". Modern high performance computers contain hundreds to several thousands of nodes. Each node is a symmetric shared-memory multiprocessor (SMP) board,

typically containing two to four sockets, with each socket commonly containing up to four processing cores (chip multiprocessors) as shown in Figure 2.5. In a shared-memory multiprocessor architecture, tasks running on multiple cores or processors are able to share their data and code. Such tasks are light weight processes called *threads*. In order to produce correct results, threads tend to communicate, share and synchronize during the execution of a parallel program. Consistency of shared data is maintained in a transparent manner through implementation of cache coherence protocols in the hardware. The term cache coherence means that any variable shared among the processors should have consistent value. Two popular cache coherence protocols are the snoopy protocol and the directory based protocol. In the *snoopy protocol*, caches listen on the common bus (broadcast medium) and invalidate or update their local copies when another processor writes to shared location. In the *directory based protocol*, the state of the cache is maintained in a centralized data structure (directory). The cost of a cache coherence conflict affects the overall performance of a multithreaded application since the data may need to be fetched from a remote processor's cache.

Nonuniform Memory Access (NUMA): In a symmetric shared-memory system all processors are symmetrically organized with respect to a single shared main memory. These systems are also called uniform memory access (UMA) machines because all processors have uniform access to main memory. In contrast to UMA, there is another class of shared-memory architectures in which each processor has its own private memory module that may or may not be shared by a remote processor. Apart from that, there is a global shared memory that can be referenced

by all the processors. Since the two memories have different access latencies, these architectures are called nonuniform memory access (NUMA) machines.

Chip Multiprocessors (Multi-cores): Multi-core technology is the semiconductor industry's solution to continued adherence to "Moore's law" scalability, while preserving power consumption. Each core is capable of running one or more tasks (threads or processes) concurrently, providing greater performance. On a typical mutli-core chip, the cores share a single main memory and I/O bus. On some multi-core processors, the highest level (L2 or L3) of cache is also shared among cores but L1 cache is typically separate (private). Naturally, added cache complexity poses a challenge to compilers and programmers. High-end application developers in particular need to be aware of the memory complexity and access schedule of the tasks besides finding a load-balanced parallel implementation.

2.2 Compiler Optimization Technology

Compiler technology has made major strides since the early days when a compiler's main purpose was to translate programs written in high level language to executable object code. Today, performance of compiled programs is a major criteria in the use of compilers. In general, compilers implement multiple levels of optimization to transform a source code to achieve higher performance. We will briefly discuss some of the optimizations that are most important for scientific applications.

2.2.1 Loop Transformations

In many scientific applications, the bulk of the execution time is spent in loops. Thus, many compiler optimization techniques target loops to optimize the performance of an application. Loop transformations play an important role in efficient use of the memory hierarchy.

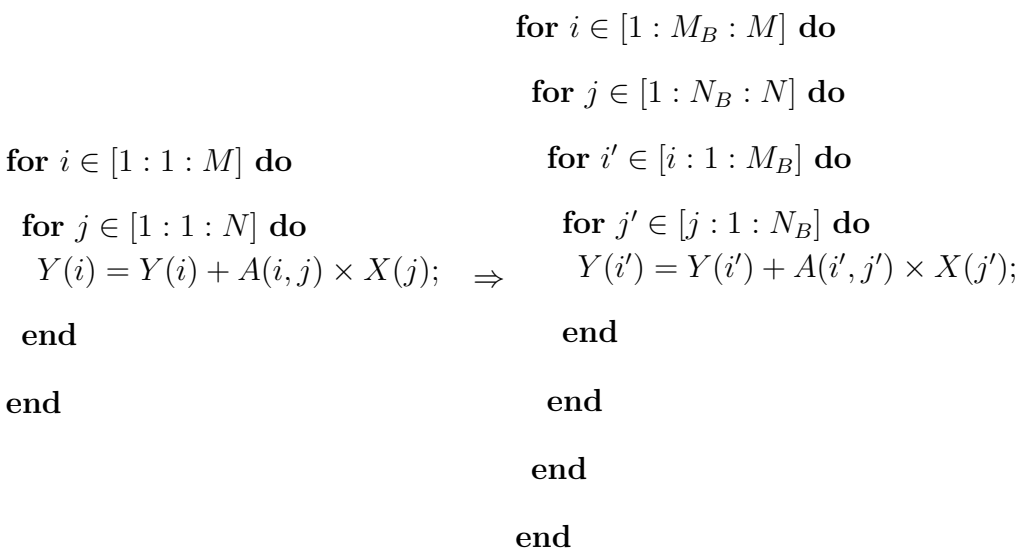
Loop Interchange: In this transformation, an inner loop is interchanged with an outer loop in a nested structure. Loop interchange is a very useful transformation that improves the cache locality of a code block. For example, consider the following loop nest (left), which accesses a matrix of size $M \times N$ in a column-wise order. Assuming that the matrix is stored in a row major order, the loop interchange transformation (loop nest on the right) would improve spatial locality by accessing the elements at a memory stride. In the following code block, the loop bounds are represented in Matlab notation.

for $j \in [1:1:N]$ **do**		**for** $i \in [1:1:M]$ **do**
for $i \in [1:1:M]$ **do**		**for** $j \in [1:1:N]$ **do**
$A(i,j) = K;$	$\Longrightarrow$	$A(i,j) = K;$
end		**end**
end		**end**

Loop Interchange

Loop Tiling: Loop tiling or blocking is one of the most important optimizations in scientific codes, particularly in the linear algebra domain. This transformation

partitions long iteration spaces of nested loops in smaller chunks (blocks/tiles) to maintain data locality. Loop blocking allows reuse of data by performing the computation on blocks of data that fit in some level of cache. For many algorithms, their block variants have been implemented to achieve higher performance on hierarchical memory as well as parallel architectures. An implementation of matrix-vector multiplication and its blocked implementation is given below:



Loop Blocking

Unroll-and-jam: Loop unroll-and-jam transformation improves efficiency of pipelined functional units by unrolling the body of outer loop and fusing the resulting inner loops together. As a result, the overhead of testing the loop condition and jump instruction is decreased. Apart from that, efficient blocking of registers and scalar replacement reduces the number of register loads and stores within the loop body. Consider the original matrix-vector multiplication loop nest given above. Assuming

the matrix contains an even number of rows, unrolling the outer loop by a factor of two would result in the following code, which saves $\frac{N}{2}$ loop conditions and an equal number of register loads.

for $i \in [1 : 2 : M]$ **do**

  for $j \in [1 : 1 : N]$ **do**

    $R1 = X(j);$

    $Y(i) = Y(i) + A(i, j) \times R1;$

    $Y(i + 1) = Y(i + 1) + A(i + 1, j) \times R1;$

  end

end

Unroll-and-jam

2.2.2 Scalar Code Optimizations

Although loops are the main target of optimizations, many scientific codes contain computational kernels that are straight-line blocks of code. Since these kernels are often called inside a loop body, significant performance improvements can be achieved by optimizing these blocks. Two of the most important optimizations that are relevant to our discussion are briefly described below.

Instruction Scheduling: As discussed in Section 2.1, all modern architectures have multiple pipelined functional units capable of executing multiple instructions. In order to take full advantage of available resources, instructions need to be scheduled so that they can be executed in parallel without resource conflicts and pipeline

bubbles. Moreover, the efficiency can be further improved by avoiding dependences among instruction that lead to stalls. Most compilers perform this optimization by creating a directed acyclic graph (DAG) of instructions in a basic block. The independent instructions in a DAG can be reordered to obtain an optimized schedule of execution.

Register Allocation: Most instructions need the operands to be loaded in registers before an operation can be performed. Multiple instructions executing in parallel may cause register conflicts that can lead to stalls. Independent instructions using the same registers may be rescheduled or they could be allocated different registers to avoid such conflict. However, due to a limited number of available registers this can easily lead to register spills. As a tradeoff, dependent instructions that reuse the registers should be scheduled closer together to minimize the register spills.

2.2.3 Prefetching and Copying

Prefetching and memory copying are two important optimizations that improve effective memory bandwidth and latency. Unlike the transformations discussed in previous sections, these optimizations do not change the schedule of computation. Prefetching hides memory latency by predicting and fetching the data from main memory to cache before it is referenced. The data is fetched in blocks, typically equal to the cache line. When the data is referenced, the load instruction takes fewer cycles to service it from the cache. Prefetching can significantly improve the performance of programs that exhibit high spatial locality. However, if a program

makes strided access to the input data, prefetching may decrease the effective memory bandwidth by loading unwanted elements in cache. Many scientific applications use blocking algorithms to improve effective memory bandwidth. When the input data is not stored in a favorable layout and sufficient reuse of data is guaranteed, copying can be used to reorder data so that subsequent memory references can be made at unit stride. Copying not only ameliorates cache conflict misses, it also avoids TLB misses, which can be very expensive. However, one of the disadvantages of these optimizations is that more instructions need to be executed as a result of additional prefetching and copying instructions. Therefore, the profitability of these optimizations must be carefully analyzed so that they are inserted only if the benefit outweighs the execution overhead.

2.3 Automatic Performance Tuning

Compiler technology has made remarkable progress in the past few decades. But the pace of development in computer architecture keeps making it hard for compilers to optimize arbitrary pieces of code in palatable amount of time. For high-end applications, hand tuning of codes has generally delivered better performance than compiler generated code, specially in the early years of new architectures. However, one of the drawbacks of hand tuning is the lack of performance portability across architectures, requiring re-implementation for every new generation of architectures. Over the past decade, the high performance computing community has turned to automatic performance tuning strategies for performance critical applications. Domain specific

libraries and code generators have been developed that implement domain specific optimizations successfully because of a priori knowledge of code function. Some optimizations also adapt the code to various features of the underlying architecture. The code generator systems generate parameterized code blocks (microkernels) that are automatically tuned to the architecture. At run-time, the parameterization allows for optimal scheduling of computations and data access patterns with respect to the memory hierarchy.

2.3.1 Search

One of the major ingredients of a two-layered adaptation methodology is run-time analysis and search schemes. Even in optimizing compilers and code generators, search plays an important role. The main objective of search is to find the right combination of values for optimization parameters that yield the best performance in the generated code. Depending on the type of code, these parameters may include loop tiling, unroll factor and register reuse distance. In signal processing codes, including the fast Fourier transform, search may be performed over a large space of formulas, which include different factorizations and algorithms. A search scheme can be driven by a model or it can be empirical [65]. Empirical search generally does a better job at finding the optimal code at the cost of additional search time. Most auto-tuning libraries, i.e., ATLAS[18], FFTW[25] and SPIRAL[44] favor empirical search scheme, preferring performance over one-time search cost. However, depending on the complexity and structure of code, efficient performance models can be generated that reduce the cost of search without severely impacting the quality

of the generated code. Recent studies (Yotov *et al.* [64]) have shown that codes generated through model driven search can match the performance of those that are generated through empirical search.

2.3.2 Feedback-directed Tuning

Using static analysis and simple heuristics that minimize the number of operations is not sufficient to generate highly efficient codes; especially on modern, complex architectures. Among other factors, instruction schedule and pipelining play an important role in the overall performance. Unfortunately, finding the optimal combination of loop parameters and instruction schedules is not trivial [3]. To overcome the limitations of static analysis, dynamic techniques, i.e. iterative compilation and feedback-directed optimization have been studied [36, 28, 61, 15]. Feedback-directed optimization automates much of the process of performance tuning by empirically evaluating the transformed code to guide the optimization process. The iterative empirical technique is known to yield significant performance improvement in linear algebra kernels [61].

ATLAS: The automatically tuned linear algebra software (ATLAS) [18] is an adaptive system for generating highly tuned parameterized basic linear algebra subroutines (BLAS). There are three levels of BLAS routines generated by ATLAS, i.e. BLAS level 1-3, where the level reflects the number of nested loops. ATLAS is an excellent example that effectively employs the iterative compilation technique. The methodology called automatic empirical optimization of software (AEOS) [60], uses

iterative empirical evaluation of many variants to choose the best performing BLAS routines. Overall performance of the generated codes depends on a number of parameters including loop block size and unroll factor. The best combination of these parameters are searched to generate code that is tuned to a given architecture.

2.3.3 Cache-oblivious Strategies

Two popular approaches have been researched for adapting performance critical applications to the memory hierarchy of architectures. In the cache-oblivious approaches [27], code is optimized for memory hierarchy without knowing the architecture parameters, such as cache capacity, block size or associativity; hence the name cache-oblivious. "Cache-conscious" approaches use architecture parameters to guide the optimization. The parameters are generally discovered automatically on a given architecture by performing a global empirical search. The cache-oblivious scheme relies on recursive divide-and-conquer restructuring of code. The basic idea is to divide the problem in successively smaller subproblems so that it will eventually fit in some level of the memory hierarchy. Such an implementation has asymptotically optimal I/O complexity [33, 27] in a theoretical cache model. The cache-oblivious strategy has been successfully employed in FFTW [25], which is a state-of-the-art FFT computation library. However, since cache-oblivious strategies are implemented using recursive codes, they inherit the performance problems associated with such codes [66]. Since most compiler optimizations are performed on loops and interprocedural analysis is typically hard, there is very little opportunity for compilers to perform optimizations on recursive codes. Apart from that, the calling overhead due

to recursive calls may degrade the performance further.

Chapter 3

The Fast Fourier Transform

The fast Fourier transform (FFT) is a divide-and-conquer method for quick evaluation of the discrete Fourier transform (DFT). The FFT is one of the most ubiquitous algorithms in the modern world of high speed digital communications. It also has a wide range of applications from digital signal processing to astronomy and weather forecast. According to some, the modern world began in 1965 when Cooley and Tukey published their famous paper[17, 37] on FFT, which reduced the complexity of DFT computation from $O(n^2)$ to $O(n \log n)$. Since then a considerable research effort has been devoted to the problem of optimizing the FFT algorithm on different architectures. In the following, we briefly discuss the main algorithms for computing the DFT. For a thorough treatment of the material and the notation used in our discussion, we refer the reader to [37, 59].

3.1 Algorithms

3.1.1 Discrete Fourier Transform

The discrete Fourier transform (DFT) of a complex vector of length n is the product of input vector X with an $n \times n$ DFT matrix W_n:

$$\begin{aligned} y_k &= \sum_{j=0}^{n-1} \omega_n^j \times x_j, \qquad \forall k \in [0, n-1] \\ Y &= W_n \times X \end{aligned} \tag{3.1}$$

The inverse discrete Fourier transform (IDFT) is given by:

$$\begin{aligned} x_j &= \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{-j} \times y_k, \qquad \forall j \in [0, n-1] \\ X &= W_n^{-1} \times Y \end{aligned} \tag{3.2}$$

where ω_n is the n^{th} root of unity:

$$\begin{aligned} \omega_n &= exp\left(\frac{-2\pi i}{n}\right) \\ &= \cos\left(2\pi/n\right) - i \sin\left(2\pi/n\right) \end{aligned}$$

The DFT algorithm presented in equations (3.1-3.2) requires $O\left(n^2\right)$ arithmetic operations to compute the transform. The coefficients in W_n, commonly known as the *twiddle factors*, are generally precomputed and stored in memory for reuse.

3.1.2 General Radix Algorithms

The periodicity of ω_n introduces an intricate structure in the DFT matrix W_n; many coefficients of the matrix are either 1, -1, i, $-i$, or can be derived from each other

using these factors:

$$
\begin{aligned}
W_1 &= \left(\omega_1^0\right) = (1) \\
W_2 &= \begin{pmatrix} \omega_2^0 & \omega_2^0 \\ \omega_2^0 & \omega_2^1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\
W_4 &= \hat{\omega_4} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & 2 & 4 & 2 \\ 0 & 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}
\end{aligned}
$$

Taking advantage of the periodicity in W_n, it is possible to factorize the matrix into a small number of sparse factors. This sparse factorization of the DFT matrix forms the basis for the FFT idea. If $n = r \times m$, we can write a two dimensional mapping of X to Y by re-indexing input and output array indices as:

$$
\begin{aligned}
j &= j_1 r + j_2 \qquad\qquad 0 \le j_2,\ 0 \le j_1 \\
k &= k_1 + k_2 m \qquad\qquad k_1 < m,\ k_2 < r
\end{aligned}
\tag{3.3}
$$

such that $x_j = x_{j_1 r + j_2} = x_{j_2}^{j_1}$ and $y_k = y_{k_1 + k_2 m} = y_{k_1}^{k_2}$. Then,

$$
\begin{aligned}
y_{k_1}^{k_2} &= \sum_{j_2=0}^{r-1} \sum_{j_1=0}^{m-1} \omega_n^{(k_1 + k_2 m)(j_1 r + j_2)} \times x_{j_2}^{j_1} \\
&= \sum_{j_2=0}^{r-1} \omega_n^{j_2 k_2 m} \omega_n^{j_2 k_1} \sum_{j_1=0}^{m-1} \omega_n^{j_1 k_1 r} \omega_n^{j_1 k_2 r m} \times x_{j_2}^{j_1} \\
&= \sum_{j_2=0}^{r-1} \omega_{n/m}^{j_2 k_2} \omega_n^{j_2 k_1} \sum_{j_1=0}^{m-1} \omega_{n/r}^{j_1 k_1} \omega_{n/rm}^{j_1 k_2} \times x_{j_2}^{j_1}
\end{aligned}
\tag{3.4}
$$

we know that $\omega_{n/rm} = \omega_{n/n} = 1$ and $n/r = m$ and $n/m = r$. Therefore, Eq. (3.4) can be written as:

$$\begin{aligned} y_{(k_1+k_2m)} &= \sum_{j_2=0}^{r-1} \left(\left(\sum_{j_1=0}^{m-1} x_{(j_1r+j_2)} \times \omega_m^{j_1k_1} \right) \omega_n^{j_2k_1} \right) \omega_r^{j_2k_2} \\ Y &= (W_r \otimes I_m) \times T_m^n \times (I_r \otimes W_m) P_r^n \times X \end{aligned} \tag{3.5}$$

The algorithm given in Eq. (3.5) is the "Kronecker Product" formulation of the Cooley Tukey *mixed-radix* algorithm, where factors r and m are referred to as the radix. Application of a single step of this splitting generates the output in transposed order requiring an extra sorting step given by the mod-r sort permutation matrix P_r^n. In Eq. (3.5), T_m^n is the diagonal "twiddle factor" matrix:

$$T_m^n = diag\left(I_m, \Omega_{n,m}, \ldots, \Omega_{n,m}^{r-1}\right) = \begin{pmatrix} I_m & & & \\ & \Omega_{n,m} & & \\ & & \ddots & \\ & & & \Omega_{n,m}^{r-1} \end{pmatrix},$$

where $\Omega_{n,m} = diag\left(1, \omega_n, \ldots, \omega_n^{m-1}\right)$.

To illustrate the Cooley Tukey algorithm given in Eq. (3.5), let us consider a DFT of size $n = 4$. Using radix-2 factorization, i.e., $r = 2,\ m = \frac{n}{r} = 2$, we get:

$$\begin{aligned} Y &= W_4 \times X \\ Y &= (W_2 \otimes I_2) \times T_2^4 \times (I_2 \otimes W_2)\, P_2^4 \times X \\ Y &= \begin{pmatrix} I_2 & I_2 \\ I_2 & -I_2 \end{pmatrix} \times T_2^4 \times \begin{pmatrix} W_2 & 0 \\ 0 & W_2 \end{pmatrix} P_2^4 \times X \end{aligned}$$

Substituting the diagonal twiddle factor matrix T_2^4:

$$T_2^4 = diag\left(I_2, \Omega_{4,2}\right) = \begin{pmatrix} I_2 & \\ & \Omega_{4,2} \end{pmatrix} = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & \omega_4^1 \end{pmatrix},$$

and the mod-2 sort permutation matrix P_2^4:

$$P_2^4 \times X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{pmatrix},$$

we get:

$$Y = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \times \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & \omega_4^1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{pmatrix}$$

In the Cooley Tukey mixed-radix algorithm (3.5), the permutation step is performed on input data (time), this is known as *decimation-in-time* factorization. A similar factorization can be obtained by performing the sorting on output data (frequency) at the end of a transform, which results in *decimation-in-frequency* factorization. The Gentleman-Sande algorithm [29] is an example of a DIF FFT, which is obtained by transposing the Cooley Tukey factorization:

$$\begin{aligned} Y &= W_n \times X = W_n^T \times X \\ &= P_r^n \times (I_r \otimes W_m) \times T_m^n \times (W_r \otimes I_m) \times X \end{aligned} \tag{3.6}$$

Other efficient algorithms have also been developed that alternate between the DIT and DIF to compute a FFT [42, 47].

In the following, we give the arithmetic complexity analysis of the Cooley Tukey algorithm. To simplify the analysis, we assume **radix-2** splitting, i.e., $r = 2$ and that n is a power of 2:

$$\begin{aligned}
\text{Time complexity of DFT of size } n &= T(n) \\
T(n) &= 2T(m) + O(n) + mT(2) \\
T(n) &= 2T\left(\frac{n}{2}\right) + O(n) + \left(\frac{n}{2}\right)T(2) \\
T(n) &= 2T\left(\frac{n}{2}\right) + O(n)
\end{aligned}$$

Solving the recursion we get

$$T(n) = O(n \log n) = Kn \log n + O(n) \quad (3.7)$$

In Eq. (3.7), constant K depends on radix of the algorithm. For example, radix-2 factorization yields $K = 5$ and radix-4 and 8 yield $K = 4.25$ and $K = 4.08$ respectively. When size n is a power of 2, arithmetic operations can be further reduced by splitting the problem in one $m = n/2$ DFT and two $r = n/4$ DFTs. This idea, called the *split-radix* algorithm [63, 20, 49, 46], yields $K = 4$.

3.1.3 Prime Factor Algorithm

The prime factor algorithm (PFA), originally developed by Good and Thomas [30, 58] and later by Winograd [62], works on sizes that can be split into two co-prime factors, i.e., $n = m \times r$ where $gcd(m, r) = 1$. In the mixed-radix algorithm a non-trivial

amount of computation is associated with the twiddle factor multiplication step. The PFA algorithm given by:

$$Y = \Gamma_{r,n} \times (W_r \otimes W_m) \times \Upsilon_{r,n}^T \times X \tag{3.8}$$

avoids this step by mapping the one dimensional arrays to two dimensional arrays using complex prime factor mappings (PFM)[37]. To illustrate this, let us consider a DFT of size $n = 6$ with two co-prime factors, i.e., $r = 3$ and $m = 2$. We know that,

$$W_6 = \Gamma_{3,6} \times (W_3 \otimes W_2) \times \Upsilon_{3,6}^T = \Gamma_{3,6} \times (W_6^2 \otimes W_6^3) \times \Upsilon_{3,6}^T$$

$$\hat{\omega_6} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 2 & 4 & 0 & 2 & 4 \\ 0 & 3 & 0 & 3 & 0 & 3 \\ 0 & 4 & 2 & 0 & 4 & 2 \\ 0 & 5 & 4 & 3 & 2 & 1 \end{pmatrix} = \Gamma_{3,6} \times \hat{\omega_6} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 3 & 0 & 3 \\ 0 & 0 & 2 & 2 & 4 & 4 \\ 0 & 3 & 2 & 5 & 4 & 1 \\ 0 & 0 & 4 & 4 & 2 & 2 \\ 0 & 3 & 4 & 1 & 2 & 5 \end{pmatrix} \times \Upsilon_{3,6}^T$$

Notice that $W_3 \otimes W_2$ is a row/column permutation of W_6. Indeed, the $\Upsilon_{r,n}^T$ permutation is based on the *Ruritanian mapping* while the output permutation $\Gamma_{r,n}$ is based on the *Chinese remainder theorem* mapping as shown in Table 3.1. In general, the Ruritanian map is given by:

$$j =< rj_1 + mj_2 >_n \tag{3.9}$$

where $j_1 =< \alpha_1 j >_m$ and $j_2 =< \alpha_2 j >_r$ and the CRT map is given by:

$$k =< \alpha_1 r k_1 + \alpha_2 m k_2 >_n \tag{3.10}$$

Table 3.1: Prime factor index mappings for $r = 3$ and $m = 2$

		j_1	
		0	1
	0	0	3
j_2	1	2	5
	2	4	1

(a) Ruritanian

		k_1	
		0	1
	0	0	3
k_2	1	4	1
	2	2	5

(b) CRT

where $k_1 =< k >_m$, $k_2 =< k >_r$. The constants α_1 and α_2 are called *rotations* where $0 < \alpha_1,\ 0 < \alpha_2$, such that

$$< \alpha_1 r >_m \ = \ 1 \tag{3.11}$$

$$< \alpha_2 m >_r \ = \ 1 \tag{3.12}$$

For example, if $n = 6,\ r = 3$ and $m = 2$, we get $\alpha_1 = 1$ and $\alpha_2 = 2$. To ensure an in-place algorithm that is also self-sorting, Burrus [13] suggested using Ruritanian mapping for both input and output permutation. However, Temperton [54] suggested using CRT mappings, which simplifies the index computation by avoiding the modulo operations. Once the indices have been properly initialized, the next row or column of indices can be calculated by cyclically shifting positions in the previous row or column followed by an increment [16]. An implication of using identical mappings is that it requires special PFA (rotated) modules $W_r^{(\alpha_2)}$ and $W_m^{(\alpha_1)}$ [13, 55]. The in-place in-order formulation of the PFA algorithm (3.8) can be written as:

$$Y \ = \ \Gamma_{r,n} \times (W_r^{(\alpha_2)} \otimes W_m^{(\alpha_1)}) \times \Gamma_{r,n}^T \times X \tag{3.13}$$

Further discussion on efficient implementations of the PFA algorithm can be found in a series of papers by Burrus [13] and Temperton [54, 57].

3.1.4 Prime Size (Rader) Algorithm

For large prime size FFTs, Rader [45, 37] developed an algorithm that uses convolution to reduce a problem of prime size n to two non-prime size $n-1$ FFTs, for which we may use any other algorithm. It uses a number theoretic permutation of W_n that produces a skew-circulant submatrix of order $n-1$. The Rader factorization can be written as:

$$W_n = P^T_{n,r^{-1}} \times \begin{pmatrix} 1 & \mathbf{1^T_{n-1}} \\ \mathbf{1_{n-1}} & S_{n-1} \end{pmatrix} \times P_{n,r} \tag{3.14}$$

where $\mathbf{1_{n-1}}$ is a vector of all ones, $P_{n,r^{-1}}$ and $P_{n,r}$ are exponential permutations associated with the primitive root r. If n is prime, then an integer r, with $2 \leq r \leq n-1$ such that

$$\{2, 3, \ldots, n-1\} = \left\{< r >_n, < r^2 >_n, \ldots, < r^{n-2} >_n\right\}$$

is called the *primitive root* of n. If $1 \leq r \leq n-1$, then there exists an inverse r^{-1}, with $1 \leq r^{-1} \leq n-1$ such that $< r \times r^{-1} >_n = 1$. If r is a primitive root of n then the $n \times n$ permutation matrix $P_{n,r}$ can be given by:

$$\begin{aligned} Z &= P_{n,r} \times X \ , \\ Z(k) &= \begin{cases} X(k) & if\ k = 0, 1 \\ X\left(< r^k >_n\right) & if\ 2 \leq k \leq n-1 \end{cases} \end{aligned}$$

For example, if $n = 5$, then $r = 2$ and $r^{-1} = 3$, the permutations associated with $P_{5,2}$ and $P_{5,3}$ are $[0\ 1\ 2\ 4\ 3]^T$ and $[0\ 1\ 3\ 4\ 2]^T$ respectively. In Eq. (3.14), S_{n-1} is a skew-circulant matrix, which is computed by performing two FFTs of size $n-1$, i.e.,

$$\begin{aligned} S_{n-1} &= W_{n-1} \times D \times W_{n-1} \\ D &= diag\left(W_{n-1}^{-1} C\right) \\ C &= \left[\omega_n, \omega_n^r, \ldots, \omega_n^{n-2}\right]^T \end{aligned}$$

Notice that if multiple prime size vectors need to be transformed, the diagonal matrix D can be precomputed and stored in memory for repeated use[2].

3.2 Data Flow Visualization

Before we discuss the variations (implementations) of FFT algorithms, we will present an intuitive way to visualize the flow of data in the FFT. Many authors use a signal flow graph called the *butterfly* to visualize an FFT. For example a radix-2 DIT FFT can be depicted as shown in Figure 3.1. In general, a butterfly diagram of any mixed-radix FFT can be drawn by combining many small-radix butterflies. When the size of the FFT is a power of two ($n = 2^i$), there are $n\log(n)/2$ radix-2 butterflies arranged in a multi-level diagram. There are i levels in the butterfly diagram, which are also referred to as *dimension* or *rank*. An example of a size $n = 8$ FFT butterfly diagram is shown in Figure 3.2. In the diagram, there are $n/r = 8/2 = 4$ small butterflies in each of $\log(n) = \log(8) = 3$ ranks. Solid filled circles imply multiplication with a twiddle factor. In an FFT computation, data flows from left (lower rank) to

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 & \omega_2 \\ 1 & -\omega_2 \end{pmatrix} \times \begin{pmatrix} a \\ b \end{pmatrix}$$

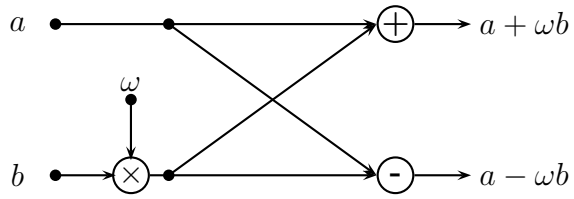


Figure 3.1: Radix-2 FFT butterfly

right (higher rank). Moreover, the computation may follow a breadth first (iterative) or a depth first (recursive) flow. Iterative and recursive schedules are discussed in the next section.

An alternative method to visualize the FFT is through multidimensional arrays or meshes. Each level of splitting in a Mixed Radix splitting converts a one dimensional FFT into a two dimensional FFT with the "twiddle factor" multiplication sandwiched in the middle. If a radix $r = 2$ splitting is used, the diagram is referred to as hypercube of i dimensions. As an example of a size $n = 8$ FFT hypercube diagram is shown in Figure 3.3. In the diagram, there are twelve edges (four in each dimension), each of which represents a single butterfly computation. Solid lines indicate multiplication with twiddle factors.

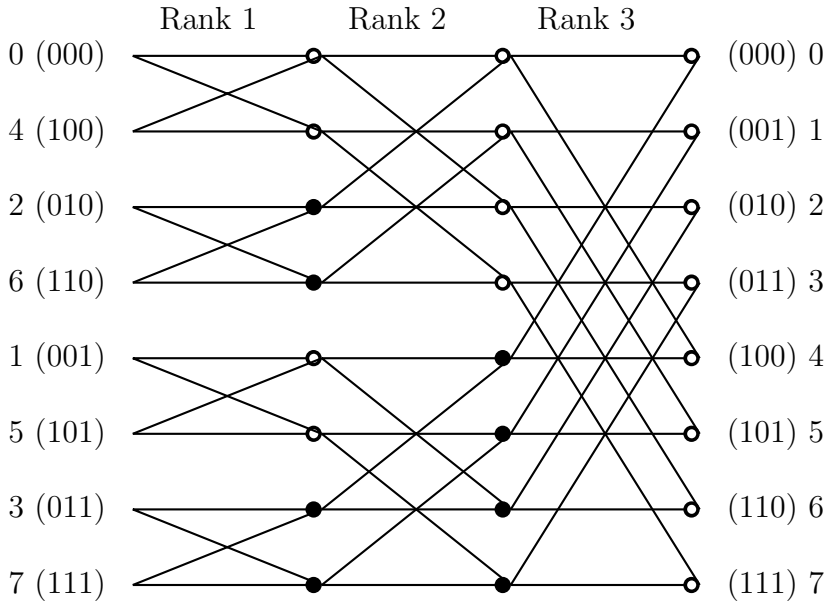


Figure 3.2: Butterfly diagram of 8-point DIT FFT with input in bit-reversed order

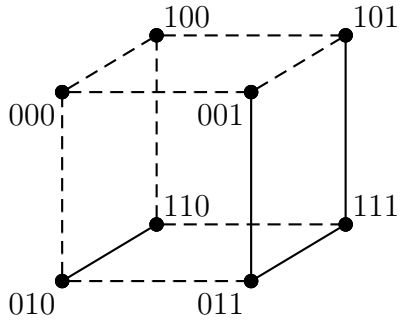


Figure 3.3: Hypercube diagram of 8-point DIT FFT with input in bit-reversed order

3.3 Schedules

Initial research efforts mentioned in Section 3.1 focused on the design of algorithms that minimized the number of arithmetic operations. Although the FFT algorithm

reduces asymptotic complexity of computing a DFT from $O(n^2)$ to $O(n \log n)$, it introduces complex strided memory access patterns (Figure 3.2) that poses optimization challenges on modern hierarchical memory architectures [12, 37, 25]. On these architectures, the optimization goal has shifted toward extracting maximum memory locality through efficient scheduling of operations and memory accesses. Overall performance of an FFT implementation is a complex function of many variables including factorization tree, algorithm selection, ordering of small butterfly computations, availability of output and workspace buffers, and layout of precomputed twiddle factor multipliers. At each level of splitting, there are numerous choices for each of these variables, resulting in an exponential space of combinations, which we refer to as the *schedules*. Considering the mixed-radix factorization alone there are ${}^{n}/_{2} = O(n)$ different combinations (trees) that can be used to compute an FFT of size $n = 2^i$ as shown in Figure 3.4. Note that actual computation takes place at leaves (non-filled circle) of the factorization trees.

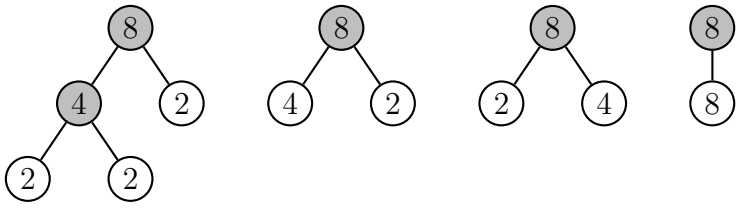


Figure 3.4: Mixed-radix factorization trees for 8-point FFT

3.3.1 Iterative Schedules

Iterative implementations follow a breadth first (rank-wise) schedule of computations. Algorithm 4 is an iterative implementation of the radix-2 Cooley Tukey algorithm [37]. The algorithm description specifies one of many possible schedules to

$x \leftarrow P_n x$; $t \leftarrow \log(n)$;

for $q = 1 : t$ **do**
  $L \leftarrow 2^q$; $r \leftarrow n/L$; $L_* \leftarrow L/2$;

  for $j = 0 : L_* - 1$ **do**
    $\omega \leftarrow \cos(2\pi j/L) - i \sin(2\pi j/L)$;

    for $k = 0 : r - 1$ **do**
      $\tau \leftarrow \omega.x(kL + j + L_*)$;

      $x(kL + j + L_*) \leftarrow x(kL + j) - \tau$;

      $x(kL + j) \leftarrow x(kL + j) + \tau$

    end

  end

end

Algorithm 4: Iterative implementation of radix-2 Cooley Tukey algorithm

compute the FFT of size $n = 2^i$. We can transpose the Cooley Tukey algorithm to obtain another schedule of computation called the Pease framework[37]. It allows the two inner loops to be fused into a single loop but uses additional workspace to store intermediate data in transposed order. Due to the non-negligible cost of the "bit-reversal" permutation[34, 67] involved in the Cooley Tukey algorithm, many implementations exist that perform the unscrambling step inside the butterfly. Stockham's [37] autosort framework performs the ordering in each rank of the butterfly

by copying intermediate data back and forth between a workspace array. There are many variants of Stockham's framework, which differ in the layout of data storage in intermediate ranks and the order of inner loops (j and k). Indeed, the two loops can be ordered (interchanged) to allow small stride access, which makes them particularly suitable for vector computers [50, 9, 11]. The above-mentioned frameworks are examples of iterative breadth first schedules. A breadth first or rank-wise schedule makes at least $\log_r n$ passes over the data, making no reuse of data within ranks (temporal locality) when n is quite large. However, iterative schedules can be implemented to preserve spatial locality.

3.3.2 Recursive Schedules

Due to lack of memory locality and non-trivial cost of copies, iterative breadth first implementations are rarely used on modern architectures. Instead, recursive depth first implementations are preferred [25, 6, 7] because of their cache optimality[33]. When an output vector is available (out-of-place FFT), the unscrambling step can be performed at the leaves of the recursion tree, i.e. inside the first rank of the butterfly network. Performing the computation "in-place", i.e. without a separate output vector or temporary array, poses a very difficult problem for self-sorting FFTs. For some transform sizes, explicit data reordering is unavoidable in computing in-place and in-order FFTs. However, for most sizes including powers-of-two, Singleton [48] and later Burrus [14] and Temperton [56] developed an algorithm that is both self sorting and in-place. The algorithm avoids explicit sorting by performing implicit ordering inside the first $\lfloor \frac{\log n}{2} \rfloor$ ranks. A recursive formulation of the algorithm was

later given by Hegland [31]. This formulation is suited to parallel and vector architectures. Indeed, the "divide and conquer" property of recursive implementations makes them an attractive option on hierarchical and shared-memory multiprocessor architectures.

3.4 Libraries

One of the early attempts to distribute a portable FFT code was done through the popular FFTPACK [50] package by P.N. Swartzrauber. FFTPACK contains Fortran code for complex-complex, real-complex, complex-real, sine and cosine transforms. The Fortran FFT codes by Clive Temperton [57] and the Fortran Split-Radix FFT code by H. Sorenson [49] were other popular public domain codes. The Sorenson code computes real and complex transforms for sizes that are powers-of-two and uses the Split Radix algorithm. The Temperton codes were designed for any powers of 2,3 and 5, i.e., for $N = 2^p 3^q 5^r$ and used the Prime Factor FFT algorithm. These codes were designed to perform well on vector architectures, and for many years were the fastest codes on such architectures. Takahashi's FFTE [51, 52] is a more recent package to compute Discrete Fourier Transforms of 1-, 2- and 3- dimensional sequences of length $N = 2^p 3^q 5^r$. The CWP numerical library by David Hale from the Center for Wave Phenomena at the Colorado School of Mines was one of the first to use a microkernel based approach to optimize the FFT over multiple platforms. This library has a fixed number of codelets and works only on a limited set of array sizes. More recently, the research focus has shifted to adaptive code generators and libraries that tune

themselves with respect to performance on different architectures and for different sizes of transforms. The main idea is to dynamically construct the factorization of the DFT matrix using a combination of the above-mentioned algorithms, depending on the size of the transform. In the following, we discuss two examples of current adaptive FFT libraries and code generators in the public domain.

FFTW: FFTW (Fastest Fourier Transform in the West) [25, 24, 26] is a state-of-the-art adaptive library for the efficient computation of FFT of real and complex data of arbitrary dimensions and sizes on many architectures. It employs two-stage adaptation methodology to adapt to microprocessor architecture and memory hierarchy. At the installation time, the code generator automatically generates highly optimized small DFT code blocks called *codelets*. Apart from the regular DFT codelets, the code generator also generates special non-leaf (*twiddle*) codelets, which multiply the precomputed twiddle factors with the input vector before computing the DFT. As discussed in Chapter 5, this approach is different from the one implemented in the UHFFT. At run-time, the pre-generated codelets are assembled in a plan to compute large FFT problems. The space representing various compositions of factorizations and algorithms for a given size FFT is explored to find the best plan of execution. The FFT plan in FFTW is expressed in terms of internal structures; I/O tensors and dimensions. Although they offer a flexible design to solving a complex problem, these representations do not provide sufficient abstraction to the mathematical and implementation level details of FFT algorithms. FFTW planner considers a small number of simple plans, which are used to compose larger FFT plans. The plan initialization stage adds a one-time cost to the actual computation of the FFT. Once

a tuned plan is generated it can be repeatedly used to compute FFTs of the same size on the given architecture.

SPIRAL: SPIRAL (Signal Processing Algorithms Implementation Research for Adaptive Libraries) [44, 23] is a code generator system for generating optimized signal processing codes including the DFTs. Automatic tuning is performed in three different levels. In the first level, the formula generator uses mathematical rules and identities to expand and optimize the formula for a given transform. The various signal processing algorithms, rules and formulas are expressed in terms of vectors, matrices and tensors in a special purpose pseudo-mathematical language called SPL (Signal Processing Language). In the second level, the optimized SPL formula is translated into source code, which can be compiled by general purpose C or Fortran compilers. At this level, various scalar and loop level optimizations are performed to reorder the code for better efficiency. Moreover, for certain architectures that support special instructions, such as short vector SIMD and FMA instructions, the formula translator identifies and translates the segments that can be mapped to such instructions. Finally, in the third level, the source code is compiled and evaluated. The performance (evaluation) is interpreted by the "search and learning" module, which controls the formula selection and optimization decisions made in the first two levels. The central goal is to iteratively evaluate the different code variants for the same transform and generate the best code by guiding the code generation process. SPIRAL avoid exhaustive search through dynamic programming and evolutionary search methods.

Unlike FFTW and UHFFT, SPIRAL does not generate microkernels of codes; instead the source code is generated for the whole transform. As a result, it takes longer for SPIRAL to generate the target code and the code needs to be compiled and linked before it can be used for the computation of an FFT. Moreover, the maximum size of transform is also limited because of higher memory and time requirements for generating the whole transform. In contrast, the standard distribution of FFTW contains a large set of pre-generated codelets to be used at run-time as part of the execution plan. Expert users, who wish to generate different codelets, need to write a high-level mathematical description of the DFT algorithm in OCaml language, which is used by the code generator (`genfft`) to generate optimized source code. Unlike FFTW, the UHFFT code generator (`fftgen`) is capable of generating an optimized arbitrary size codelet without any user intervention. Unfortunately, the codes generated by the three code generators do not use a standard interface. Therefore, it is very difficult, for example, to use the codes generated by SPIRAL and FFTW within UHFFT run-time framework.

Both FFTW and SPIRAL employ expensive optimization strategies to generate the best plan or code for computation of an FFT problem. In the UHFFT, we attempt to avoid, as much as possible, the run-time empirical search. We have implemented multiple low-cost search schemes that employ accurate performance models and estimation techniques to search the best schedule of execution. Through better understanding of the correlation between the schedules and their performance on modern architectures, we have developed effective heuristics that result in a pruned

search space. In order to help understand the data and control flow of FFT computations, we have developed a high level language that is exposed through the UHFFT application programming interface (API). It allows to explore alternative schedules and to construct novel implementations at run-time. Indeed, considerable effort has been devoted in this dissertation to the design of consistent and flexible interface for the computation of FFTs.

Apart from the publicly available open-source libraries, most hardware vendors also offer mathematical libraries that include interfaces for computing the DFT. Typically, these libraries are specifically tuned to the vendor's architectures, e.g. AMD's Core Math Library (ACML) is tuned to AMD architectures. Intel's Math Kernel Library (MKL) is one of the fastest libraries for computing FFT on x86 and Intel architectures. MKL provides a comprehensive interface (DFTi [53]) for computing DFTs of different size, dimension, and precision etc.

Chapter 4

UHFFT: An Adaptive FFT Library

The UHFFT [40, 39, 5, 7, 6] is an adaptive FFT library that maintains performance portability across different architectures. Performance portability in FFT codes is achieved by adapting the computation schedule to various features of the underlying architecture, including memory hierarchy and processor level parallelism (SIMD and multi-core). The UHFFT comprises of two main components, i.e., a code generator and a run-time system, as shown in the block diagram of the UHFFT library (Figure 4.1). At installation time, the code generator produces microkernels of straight-line parameterized DFT code blocks (*codelets*) in C. For each codelet size, a limited number of variants are generated on the basis of different tuning parameters, including factorization tree, algorithm and register blocking. The variants are compiled and empirically evaluated to select the best codelet. At run-time a user may input any

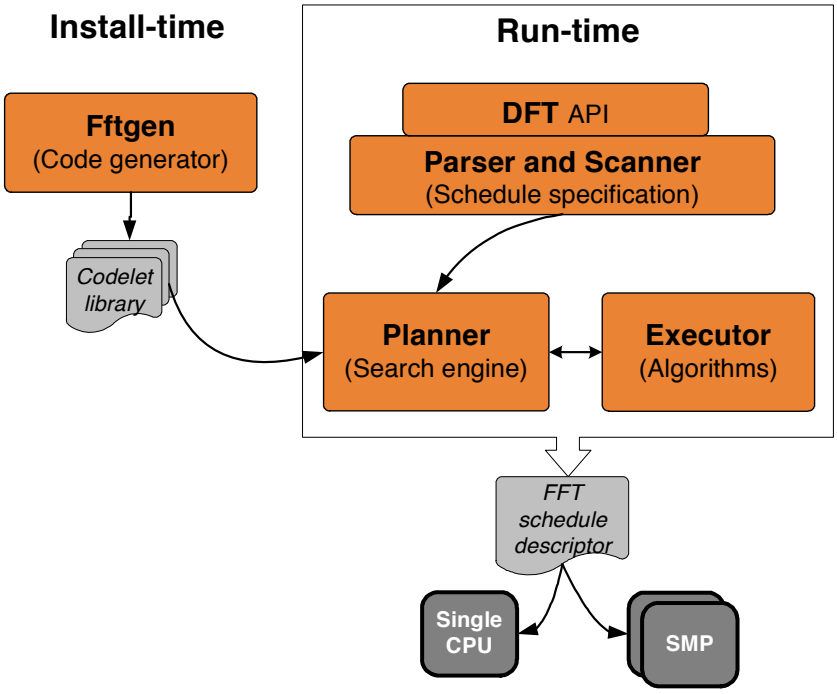


Figure 4.1: The UHFFT library design overview

arbitrary size FFT problem through the DFTi API. Given an FFT problem descriptor, a combination of parameterized *codelets* that has low computation time for the FFT is dynamically searched. The current version (2.0.1) of the UHFFT[4] supports the following features for computing an FFT of arbitrary size:

Data type complex to complex

Direction forward and inverse

Precision single and double

Placement in-place and out-of-place

Ordering ordered

Dimensionality single and multi-dimensional

The two stage adaptation methodology of the UHFFT is similar to the one employed by FFTW. However, there are some differences that set the two libraries apart as shown in Table 4.1.

Table 4.1: Differences between UHFFT and FFTW

	UHFFT	**FFTW**
Code generator		
Implementation	C	Objective Caml
Specifying Codelets	Any set of codelets can be generated by simply specifying them in a script file	Expert user needs to generate the code using OCaml dialect
Adaptation Methodology	Code is empirically optimized through feedback directed code generation	Tuning parameters and optimizations are hard coded in the code generator
Run-time		
API	Intel's DFTi	Multiple interfaces
Search	Multiple low cost search schemes	Multiple search schemes with varying cost
Specifying the FFT schedule	FFT schedule specification language allows bypassing search	Plans encrypted in internal data structures

4.1 Installation

One of the main goals of the UHFFT library is to maintain portability across different platforms. The run-time and code generator have been implemented in ANSI C, which makes it easy to build the library on any platform that supports the standard C compiler. Building the UHFFT on systems that support the GNU build system is as simple as:

```
.\configure
make
make install
```

The sequence of commands given above, builds the UHFFT with default options. It is also possible for the user to specify a different set of options through the `configure` script. A list of installation options is given in Table 4.2. The UHFFT library has been installed and tested on various platforms in order to validate its performance portability. The performance benchmarks reported in this dissertation have been performed on current microprocessors and high-end SMPs. The processor/SMP architectures have been selected on the basis of their relevance and popularity as established or emerging high performance computing platforms. In terms of software architectures, the library has been built and tested on various operating systems and compilers. The build system automatically selects the vendor compiler if it is found on the target system; otherwise, the gcc compiler is used. The library can be installed seamlessly on any flavor of Linux or Unix systems including cygwin so long as it supports GNU build tools. In the following, we give detailed specifications of

Table 4.2: UHFFT library installation options

Optional Features	**Default**	**Description**
`--enable-float`	disabled	Builds library in single precision
`--enable-single`	disabled	Same as `--enable-float`
`--enable-simd`	disabled	Generate code that supports SIMD extensions
`--enable-aeos`	disabled	Generate code using feedback directed empirical optimization technique
`--enable-threads`	disabled	Use native threads for multithreaded execution (default=OpenMP)
`--disable-openmp`	enabled	Use native threads for multithreading instead of OpenMP
`CC`	auto	Compiler is automatically selected unless specified
`CFLAGS`	auto	Compiler optimization flags are automatically selected unless specified

the testbeds selected for this dissertation.

Table 4.3: Xeon Woodcrest 5100 Specification

Processors	2×dual CMP/SMP
CPU speed	2.66GHz
CPU Peak (DP,SP)	10.64Gflop/s, 21.28Gflop/s
FP Registers	16 (128bit)
Memory BW, Latency	≈10.6GB/s, 70-380 cycles
ICache	32K
L1 DCache	32K/core, 64B, 8way, WB, 3 cycles
L2 I+DCache	4M/dual, 64B, 16way, WB, 14 cycles
Primary Compiler	icc-10.1

4.1.1 Intel Xeon Woodcrest 5100

The first testbed is one of the compute nodes in the Lonestar cluster at the Texas Advanced Computing Center. Lonestar nodes contain two Xeon Woodcrest dual core 64-bit processors (4 cores in all) on a single board, as an SMP unit. Each core is capable of executing four double precision floating point operations per cycle using 128 bit SIMD operation per add and multiply unit.

4.1.2 Intel Xeon Clovertown 5300

The Neolith cluster at the National Supercomputing Center in Linkoping Sweden consists of Xeon Clovertown processors. Each node consists of two quad core Clovertown processors. Xeon Clovertown is successor of the Woodcrest consisting of two dual-core Woodcrest processors per chip.

Table 4.4: Xeon Clovertown 5300 Specification

Processors	2×quad CMP/SMP
CPU speed	2.33GHz
CPU Peak (DP,SP)	9.32Gflop/s, 18.64Gflop/s
FP Registers	16 (128bit)
Memory BW, Latency	≈10.6GB/s, 80-280 cycles
ICache	32K
L1 DCache	32K/core, 64B, 8way, WB, 3 cycles
L2 I+DCache	8M/quad, 64B, 16way, WB, 14 cycles
Primary Compiler	icc-10.1

4.1.3 Intel Itanium 2 (Madison)

The Itanium node of the Eldorado cluster at the Texas Learning and Computation Center, used for benchmarking in this dissertation consists of four Itanium 2 processors in an SMP configuration. Itanium's architecture is significantly different from x86 based architectures (Pentium, Xeon, Opteron etc.). The Itanium 2 issues two instructions per cycle as very long instruction words (VLIW), each of which encodes three independent instructions. Unlike x86 based systems, the scheduling of instructions is not done dynamically by hardware rather it is performed by the compiler at compile time. The Itanium 2 has two floating point units capable of executing fused multiply and add (FMA) operations. Hence, given the perfect blend of instructions the processor can achieve a throughput of four floating point arithmetic operations per cycle.

Table 4.5: Itanium 2 (Madison) Specification

Processors	4×(SMP)
CPU speed	1.50GHz
CPU Peak	6.0 Gflop/s
FP Registers	128 (FP)
Memory BW,Latency	≈6.4GB/s, 165 cycles
ICache	16K
L1 DCache (Integer data)	16K, 64B, 4way, WT, 1 cycle
L2 DCache	256K, 128B, 8way, WB, 6 cycles
L3 DCache	6M, 128B, 12way, WB, 13 cycles
Primary Compiler	icc-10.1

4.1.4 AMD Opteron 285

The Opteron contains one add and one multiply unit working in a superscalar mode, generating two floating point results per cycle. The 128-bit SIMD instructions are implemented as two 64-bit operations that are internally pipelined. The SIMD instructions provide a theoretical single precision peak throughput of two additions and two multiplications per clock cycle, whereas x87 instructions can only sustain one addition and one multiplication per clock cycle. The SSE2 and x87 double-precision peak throughput is the same, but SSE2 instructions provide better code density [8].

Table 4.6: AMD Opteron 285 Specification

Processor	2×dual CMP/SMP
CPU speed	2.60 GHz
CPU Peak(DP,SP)	5.2 Gflop/s, 10.4 Gflop/s
FP Registers	16 (128bit)
Memory BW, Latency	≈6.4GB/s, ≈150 cycles
ICache	64K
L1 DCache	64K/core, 64B, 2way, WB, 3 cycles
L2 I+DCache	1M/core, 64B, 16way, WB, 12 cycles
Primary Compiler	gcc-3.4.6

4.1.5 IBM Power5+

The IBM Power5+ nodes used for benchmarking form part of the Hydra cluster at the Supercomputing facility of Texas A&M University. Each node consists of 8 dual core Power5+ processors. Each core is capable of running two threads simultaneously (SMT) to feed the pipeline. This results in increased efficiency provided the threads do not compete for cache resources. Each core consists of two floating point units that can perform fused multiply and adds, resulting in throughput of four floating point operations per cycle if the code contains a perfect blend of instructions.

Table 4.7: IBM Power5+ Specification

Processor	8×dual CMP/SMP
SMT	2 per core
CPU speed	1.90 GHz
CPU Peak	7.6 Gflop/s
FP Registers	120 (FP)
Memory BW, Latency	≈25 GB/s, 220 cycles
ICache	64K
L1 DCache	32K/core, 128B, 4way, WT, 2 cycles
L2 DCache	1.9M/dual, 128B, 10way, WB, 12 cycles
L3 DCache	36M/dual, 256B, 12way, WB, 80 cycles
Primary Compiler	xlc-8

4.2 API

The run-time part of the UHFFT has been redesigned to support the DFT interface (DFTi), which is implemented in the Intel Math Kernel Library (MKL). For a detailed specification of the DFTi API and its usage, see [53]. Computing a DFT of an input sequence typically involves three stages, i.e., problem description and setup, actual computation, and finally the cleanup stage. The sequence of five steps for computing a DFT using the DFTi API is as follows:

1. Initialize the problem (descriptor)

2. Optionally, set problem description parameters

3. Prepare and initialize schedule of execution

4. Compute the DFT of input sequence

5. Clean up the descriptor

The first three steps are part of the setup stage. The fourth step performs the transformation while the fifth step cleans up the data structures and buffers associated with the problem descriptor. Note that fourth step can be called repeatedly on different data sets of the same size and description. In the following, we discuss the DFTi calling interface associated with each of these steps. After completion, each of the functions returns error status [53] or `DFTI_NO_ERROR`, if no error was encountered.

4.2.1 Initializing the DFT Descriptor

In the first step of the DFT computation, the user needs to create the problem descriptor using the following interface in C.

```
long DftiCreateDescriptor(DFTI_DESCRIPTOR_HANDLE dfti_desc,
    DFTI_CONFIG_PARAM precision, DFTI_CONFIG_PARAM domain,
    long dimensions, void length)
```

The function call takes one output parameter, i.e., `DFTI_DESCRIPTOR_HANDLE` and four essential input parameters, which include precision of the transform, domain of FFT computation, number of dimensions and length of each dimension.

Precision The precision of the library is set at installation time through the `configure`

script. Therefore, the precision parameter must be equal to the preselected setting, i.e., `DFTI_SINGLE` if `--enable-float` or `--enable-single` were turned on, otherwise `DFTI_DOUBLE`.

Domain In its current version, the UHFFT only support complex to complex FFTs. Therefore this parameter should be set to `DFTI_COMPLEX`.

Dimensions The dimension parameter may take any integer value greater than 0.

Length Depending on the number of dimensions, length may take a scalar or array parameter. For a multidimensional transform, length is expected to be the address of the array.

Note that the creation of the descriptor `DFTI_DESCRIPTOR_HANDLE` does not perform any computation on the input sequence. Instead, it only allocates and initializes the descriptor with default values corresponding to the four essential parameters.

4.2.2 Setting Configuration Parameters

The UHFFT is an ongoing project; although it supports the core functionality for computing a DFT it does not support all the features of the DFTi specification. However, the UHFFT offers some additional features outside the standard DFTi, which have been implemented as extensions. Configuration options of a descriptor can be set through the following interface:

```
long DftiSetValue(DFTI_DESCRIPTOR_HANDLE dfti_desc, DFTI_CONFIG_PARAM
param, void value)
```

where `dfti_desc` is the descriptor that has already been created and `param` is the parameter that needs to be set. In the following, we discuss a few important configuration parameters supported by the UHFFT:

Placement The UHFFT supports both in-place ($x \leftarrow F(x)$) and out-of-place ($y \leftarrow F(x)$) execution of the FFT. If the transformed result is required in a separate array, the placement parameter `DFTI_PLACEMENT` should be set to `DFTI_NOT_INPLACE` value; otherwise, it should be set to `DFTI_INPLACE`, which is the default setting when a descriptor `DFTI_DESCRIPTOR_HANDLE` is created.

Search The search scheme can be selected by setting the `DFTI_INITIALIZATION_EFFORT` parameter to one of three values, i.e., `DFTI_HIGH`, `DFTI_MEDIUM` or `DFTI_LOW`. `DFTI_HIGH` takes longer than the other two search schemes because it empirically evaluates most options for a given FFT computation. However, the other two search schemes have been found to generate good schedules reasonably quickly, and thus be preferable in many situations. A further discussion on search schemes is given in Chapter 7.

Stride A user may want to transform the data that is stored in a non-contiguous manner at strided memory locations. When the input or output data needs to be accessed or stored in a strided fashion, the information can be provided by setting the `DFTI_INPUT_STRIDES` or `DFTI_OUTPUT_STRIDES` parameters. In its current version, the UHFFT supports integer values for the strides, i.e., when the number of dimensions is greater than one, all dimensions need to have the same stride.

4.2.3 Preparing Execution Schedule

Once a problem descriptor has been initialized, it needs to be committed so that a computation schedule may be prepared. The following function call is used to commit a descriptor that has already been created:

```
long DftiCommitDescriptor(DFTI_DESCRIPTOR_HANDLE dfti_desc)
```

During this step, the bulk of the time is spent in searching for the best combination of factors and algorithms to compute the given DFT problem. The time taken usually depends on the value of the `DFTI_INITIALIZATION_EFFORT` parameter.

4.2.4 Computing the Transform

A committed DFT descriptor is ready to be used for transforming input sequences. The descriptor can be used repeatedly on any number of sequences as long as the input sequences are of same size and description. The input sequence `in` of complex data can be transformed from time domain to frequency domain or vice versa using the following DFTi interface:

```
long DftiComputeForward or DftiComputeBackward
(DFTI_DESCRIPTOR_HANDLE dfti_desc, void *in, void *out)
```

The function writes the result in `out` sequence. If the result needs to be written in-place, i.e., the `DFTI_INPLACE` parameter is selected, then `out` needs to point to

the same location as `in`. For architectures that support SIMD extensions, the performance of the computation can be significantly improved by aligning the (start of) the two arrays `in` and `out` in memory to appropriate boundaries (typically 16B). If the arrays are aligned, this information must be passed to the search engine. A user may specify the alignment flag through the precision parameter of the `DftiCreateDescriptor` interface, i.e., `precision=precision|DFTI_ALIGNED`. The UHFFT provides two interfaces for allocating aligned structures (scalar) or arrays (vector) in memory:

```
scalar(void **var, long size_of_var)
vector(void **var, long length, long size_of_element)
```

4.2.5 Freeing the Descriptor

The following interface frees up all the memory associated with the descriptor:

```
long DftiFreeDescriptor (DFTI_DESCRIPTOR_HANDLE dfti_desc)
```

4.3 Benchmarking

Performance benchmarks reported in this dissertation are collected using an auxiliary utility `bench` that is installed with the UHFFT library. This utility can be used to benchmark various types and sizes of the FFT on target architectures. The utility allows reporting of performance in several metrics, i.e., time, "MFLOPS" and

hardware counters. The "MFLOPS" (Million Floating Point Operations Per Second) metric is calculated from execution time and commonly used arithmetic complexity of the FFT,i.e.,

$$\text{MFLOPS} = 5 \times n \log(n) / time(\mu s) \tag{4.1}$$

where n is the size of the transform. Note that the MFLOPS number is an inflated count compared to the actual number of operations executed; the actual number of operations is lower than $5 \times n \log(n)$. Similarly, efficiency can be computed from the performance in MFLOPS as follows:

$$\begin{aligned} Efficiency &= \%\ of\ Theoretical\ Peak \\ &= \frac{\text{MFLOPS} \times 100}{(FPU\ Throughput \times ClockRate)} \end{aligned} \tag{4.2}$$

where "$FPU\ Throughput$" is the theoretical number of floating point operations that can be retired per cycle and $ClockRate$ is the CPU frequency in MHz. Note that the peak performance is a theoretical number which can only be achieved if the code contains an ideal mix of floating point instructions. These efficiency and performance measures are useful in comparing the relative performance of FFTs.

Hardware counters data is collected through a popular third party tool - PAPI [41]. The tool should already be installed on the system and its path needs to be set using the UHFFT installation script `configure` as shown in Table 4.8. Besides PAPI, the `bench` utility also works with FFTW and MKL. The advantage of integrating the three FFT libraries in a single `bench` utility is that it brings consistency in the performance comparison of the libraries. In the following, we briefly describe our

Table 4.8: `bench` utility options

Third party tool	**Description**
`--with-fftw3=FFTW3_PATH`	Supports benchmarking of FFTW3 through bench utility
`--with-mkl=MKL_PATH`	Supports benchmarking of MKL through bench utility
`--with-papi=PAPI_PATH`	Allows collection of performance benchmarks in terms of hardware counters

benchmarking methodology for collecting data on execution time and accuracy.

4.3.1 Speed

The speed of the DFT computation is given by the execution time or the "MFLOPS" metric, which can be calculated from the execution time of a DFT problem. However, in order to collect stable samples of execution time, higher resolution clocks may be required. Most operating systems provide timing routines with micro seconds precision. To calculate execution time of routines that take shorter than one μs, the execution must be repeated in a loop. In general, the sampling error is minimized by executing a problem significantly longer than one μs. The `bench` utility uses Algorithm 5 to generate a stable measurement of DFT execution time. Every reading is performed for an execution time of at least 0.01 seconds. If the execution

$t_{factor} \leftarrow 1$; $t_{min} \leftarrow 0.01$;

$doiter \leftarrow 1$; $niter \leftarrow 1$;

repeat

$t_1 \leftarrow$ GET_TIME();

for *i=0* **to** *niter* **do** EXECUTE DFT;

$t_{elapsed} \leftarrow$ GET_TIME()-t_1;

if $t_{elapsed} = 0$ **then** $t_{factor} \leftarrow 100$;

else $t_{factor} \leftarrow \text{MIN}(100, \text{MAX}(1.1, t_{min}/t_{elapsed}))$;

$niter \leftarrow t_{factor} \times niter$;

$doiter \leftarrow doiter + 1$;

until $t_{elapsed} < t_{min}$ *and* $doiter <$ MAXDO ;

return $t_{elapsed}/niter$;

Algorithm 5: Benchmarking methodology for calculating the execution time

time is smaller than the clock resolution, the number of iterations is increased by a factor of 100. Otherwise, if the elapsed time is smaller than the minimum execution time (0.01), the number of iterations is increased by 10%.

4.3.2 Accuracy

"Accuracy is the degree of conformity of a measured or calculated quantity to its actual (true) value". The `bench` utility can be used to determine accuracy of the result produced by the UHFFT compared with the true value. `bench` supports four types of accuracy tests; constant, square, ramp and inverse. In each test case, the

normalized error is calculated as follows:

$$Error = \frac{\|Y - \hat{Y}\|_p}{\|\hat{Y}\|_p}$$

where Y is the computed solution and $\hat{Y}$ is the true solution and $\|e\|_p$ is known as the L_p norm of e:

$$\|e\|_p = \sum_{i=0}^{n-1} (|e_i|^p)^{1/p} \tag{4.3}$$

where $p \geq 1$ and n is the size of transformed sequence. We use the L_2 norm to calculate the error between two sequences. In the following, we present the generating functions of our test cases and their true solutions.

Constant In this accuracy test, the input sequence is generated by a constant function, i.e.,

$$f(x_j) = 1 + i \qquad \forall\ 0 \leq j < n$$

The true solution of the DFT of f is given by:

$$\hat{f}(x_j) = \begin{cases} n & \text{if } j \ = \ 0 \\ 0 & \text{otherwise} \end{cases}$$

Square In this accuracy test, the input sequence is generated by a square wave function, i.e.,

$$f(x_j) = \begin{cases} 1 & \text{if } j \textit{ is even} \\ -1 & \text{if } j \textit{ is odd} \end{cases}$$

The true solution of the DFT of f is given by:

$$\hat{f}(x_j) = \begin{cases} 1 + i\tan(\pi j/n) & \text{if } n \textit{ is even} \\ \begin{cases} n & \text{if } j = (n+1)/2 \\ 0 & \text{otherwise} \end{cases} & \text{if } n \textit{ is odd} \end{cases}$$

Ramp In this accuracy test, the input sequence is generated by a ramp (step) function, i.e.,

$$f(x_j) = j/n$$

The true solution of the DFT of f is given by:

$$\hat{f}(x_j) = \begin{cases} (n-1)/2 & \text{if } j = 0 \\ -0.5 + i\,(0.5/\tan(\pi j/n)) & \text{if } j > 0 \end{cases}$$

Inverse In this test case, the DFT problem is computed on a random sequence in both forward and inverse direction and the result is compared with the original input sequence, i.e,

$$\begin{aligned} y &= F(x) \\ u &= F^{-1}(y) \\ Error &= \frac{\|u - x\|_2}{\|x\|_2} \end{aligned}$$

Chapter 5

Code Generation

5.1 Introduction

The UHFFT library is comprised of two layers, a code generator (`fftgen`) and a run-time system. The code generator produces highly optimized straight-line "C" code blocks called codelets at installation time. These codelets are parametrized blocks (microkernels) that compute parts of a given FFT problem at run time. Any standard "C" compiler can be used to compile these codelets, which ensures the ultimate portability. The automatic code generation approach is used because hand coding and tuning the FFT is a very tedious process for transforms larger than size five. It also allows for easy evaluation of different algorithms and code schedules. Because of the unique structure of FFT algorithms with output dependence between successive ranks (columns), loop level optimizations are not as effective as in the case of many other linear algebra codes. Choosing the best FFT formula (factorization

and algorithm) is vital to minimizing the number of operations required to compute the transform. However, using simple heuristics that minimize the number of operations is not always sufficient to generate the best performing FFT kernels; specially on modern, complex architectures. Among other factors, instruction schedule and register blocking play an important role in the overall performance. `fftgen` employs an aggressive optimization approach (AEOS) by iteratively generating, compiling and evaluating different variants. Apart from exploring the best FFT formula and register blocking, we try a few instruction schedules, translation schemes to probe and adapt to both microprocessor architecture and compiler. A user may disable the empirical optimization approach to expedite the installation process. In that case, default optimization parameters can be overridden by manually specifying them in a script file.

5.2 Codelet Types

`fftgen` is a very flexible code generator, capable of generating many types of codelets of arbitrary size. A simple script file can be used to specify the sets of codelets (types and sizes) to generate. The codelet sizes should typically be limited by the size of instruction cache and the number of registers on the target architecture. After the set of desired codelet sizes is specified in the script file, the code generator does not require any further user intervention to produce highly optimized micro-kernels of codelets for the platform. As shown in Figure 4, each codelet is identified by an identifier, which is a string of seven literals postfixed by the size. In the following,

we describe each of these literals.

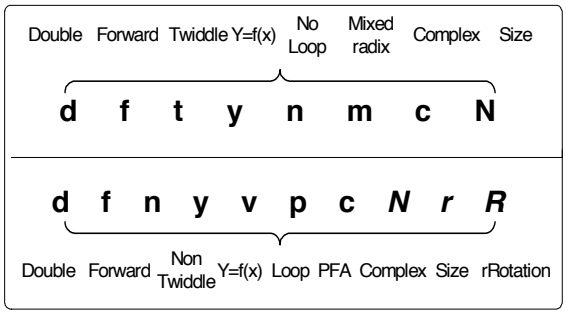


Figure 5.1: Codelet identifier string

Precision: As the name implies, this literal specifies the precision of the target codelet. It takes four possible values, `'s','d','q'` and `'z'`. Literals `'s'` and `'d'` stand for single (SP) and double precision (DP) respectively, while literals `'q'` and `'z'` are applicable to architectures that support SIMD extensions and use vectors of four SP or two DP elements as operands. SIMD-enabled codelets are discussed in detail in Section 5.3.2.

Direction: Depending on the domain (time, frequency) of the input vector, a codelet can compute the transform in forward `'f'` or inverse `'i'` direction, as indicated by the second flag.

Twiddle: The Cooley-Tukey mixed-radix FFT algorithm involves multiplication with a diagonal twiddle matrix between the two FFT factors. Similar to the approach

in FFTW[25], we generate special "*twiddle codelets*", which perform the twiddle multiplication inside the codelet to avoid extra loads. This flag takes two possible values, `'t'` for twiddle and `'n'` for non-twiddle. For a size n codelet, the n twiddle multipliers are fused at the end; multiplication is applied to the output of the codelet. This approach is different from the FFTW approach in which $n-1$ multipliers are fused at the beginning of a size n codelet. Note that the number of twiddle factor multiplications in a given FFT remains the same for both approaches. However, the UHFFT approach requires fewer twiddle codelet calls, which are more expensive than the non-twiddle codelets. As an example, Figure 5.2 shows four radix-2 codelet calls (one

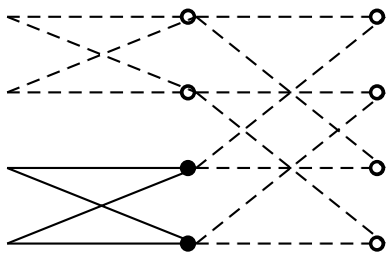


Figure 5.2: Difference between twiddle and non-twiddle codelets

for each radix-2 butterfly), one of which (in rank 1) requires twiddle multiplication at the end, depicted by filled circles. Notice though, that the twiddle multiplication can alternatively be deferred and be applied to the input of both codelets in rank 2.

Placement: This flag is also specific to the mixed-radix FFT algorithm. It specifies whether the codelet is part of an in-place or out-of-place (not-in-place) algorithm. As discussed in Section 5.3, the in-place algorithm requires partial transposes on

square block DFTs. An in-place codelet performs n DFTs of size n and transposes the result inside the codelet to avoid extra register loads. This flag takes two possible values; `'x'` for in-place ($x = f(x)$) and `'y'` for out-of-place ($y = f(x)$) computation.

Vector-ffts: In most cases multiple equi-distanced vectors of the same size need to be transformed. A codelet with the vector flag enabled performs multiple DFTs of the same size in a loop. The codelet call takes two additional parameters to let a user specify the strides (distance) between successive vectors and the number of loops. This is also a Boolean flag; it takes two possible values; `'v'` for vector codelets and `'n'` for straight-line codelets.

Algorithm: Different types of FFTs and algorithms may require different codelets. Apart from the codelets used in mixed-radix algorithms, `fftgen` generates special codelets that can be used in the PFA algorithm (Section 3.1.3) and trigonometric transforms. The algorithm flag takes four possible values, i.e., `'m','p','c'` and `'s'` representing mixed-radix, PFA, cosine(DCT) and sine(DST) algorithms, respectively.

Data: Both real and complex data codelets can be generated depending on the transform type. In this dissertation, we focus on complex data type codelets. The flag takes `'c'` for complex-to-complex type codelets and `'r'` for real-to-complex data codelets.

Rotation: This parameter is only applicable to *rotated codelets* that are used as part of the PFA algorithm. Note that the PFA codelet with rotation of one is

equivalent to a regular (non-PFA) codelet. Rotation is specified by appending `'r'` and (rotation) at the end of the codelet identifier string, as shown in Figure 5.1.

5.2.1 Codelet Data Structure

The data structure used for a codelet is of type `Module`. Apart from specifying the type, size and rotation of a codelet, `Module` also contains four function pointers to each of the forward/inverse and twiddle/non-twiddle codelet calls as shown in Figure 5.3. When a codelet is generated, the corresponding `Module` structure is initialized accordingly. Each library of codelets is defined by a global array of modules. The codelet modules initialized at code generation time are immutable objects. Therefore, when a module needs to be inserted in a schedule, the original codelet is cloned by performing a deep copy. This ensures that same codelet can be used in different contexts as part of different schedules. For example, a radix-2 schedule for an FFT of size $n = 2^i$ could contain a list of i modules, each of them a clone of the same codelet, i.e., of size 2. Although, each `Module` in the list will have the same type and size, the stride, twiddle array and next pointers would have different values depending on their position in the container schedule. Note that a "container schedule" itself is also represented by the `Module`; its type depends on the algorithm that it implements. Some schedules may contain a mix of different algorithms, in which case each sub-schedule is represented by a branch called *switch* module. The switch module is used to build schedules that are composed of different algorithms. For example, a PFA sub-schedule may be embedded inside a mixed-radix schedule or in case of Rader's algorithm the FFT for size $n - 1$ needs to be embedded in the prime module of size

n. Most of this will become clearer when we discuss implementation of schedules in the next chapter.

5.3 Codelet Libraries

Although many codelet types may be constructed using different string combinations, only a few are useful in the UHFFT run-time. Many string combinations (codelet types) do not make sense; for example, the twiddle flag is not applicable to rotated (PFA) codelets. Similarly, a PFA codelet does not need a separate inverse direction codelet.

In the following discussion, we concentrate on double precision complex-to-complex codelet libraries since the generated code is fundamentally the same for single precision. However, when the functionality is significantly different (as in case of SIMD codelets), we will point that out.

5.3.1 Scalar Codelets

Scalar or non-SIMD codelet libraries are architecture independent kernels that are generated in the "C" programming language. Although the codelets in this category are portable, they are generally not the most efficient on architectures that support SIMD extensions. That is because most compilers are not good at performing automatic simdization (vectorization) of straight-line codes. In the following, we describe four main codelet types used in the UHFFT library run-time.

```
typedef struct module {
   long               n;             /* The size of the module */
   long               type;          /* Module type */
   long               rot;           /* Module rotation */
   long               stride;        /* Module stride */
   long               v;             /* Reserved */
   CPLEX *            w;             /* Twiddle factors */
   long               ws;            /* Twiddle array stride */
   Module *           next;          /* Next module */
   PFV                call;          /* Forward routine */
   PFV                icall;         /* Inverse routine */
   PFV                twcall;        /* Forward twiddle routine */
   PFV                twicall;       /* Inverse twiddle routine */
   union advparams    advParams;     /* Switch module */
   CPLEX *            work;          /* Scratch buffer */
} Module;
```

Figure 5.3: Module data structure

DYNMC: This is the most basic type of straight-line complex-to-complex FFT codelets used in a mixed-radix algorithm. For each size n, four codelets are generated; forward non-twiddle (`dfnynmc`), inverse non-twiddle (`dinynmc`), forward twiddle (`dftnymc`) and inverse twiddle (`ditynmc`). The calling interfaces of twiddle and non-twiddle codelets of size 4, are:

```
d?nynmc4(CPLEX *X, CPLEX *Y, long is, long os)
d?tynmc4(CPLEX *X, CPLEX *Y, long is, long os, CPLEX *W)
```

`X` and `Y` are, respectively, the input and output arrays of structure (complex). The output vector may point to the same location as the input vector, in which case the input vector is overwritten. `is` and `os` are the input and output strides respectively. In terms of Matlab vector notation, this codelet is equivalent to the following operation:

$$Y\left[0 : os : n \times os\right] = \mathbf{fft_n}\left(X\left[0 : is : n \times is\right]\right)$$

For twiddle codelets, the twiddle factor array is accessed at unit stride.

DYVMC: This type of codelet is similar to the one discussed above, except that it performs the same DFT on multiple equi-distanced vectors. In a large FFT problem, a codelet needs to be called multiple times inside a loop. This codelet pushes that loop inside the subroutine to save the calling overhead. For each size n, four codelets are generated for every combination of direction and twiddle flags. The calling interfaces of twiddle and non-twiddle codelets of size 4, are:

```
d?nyvmc4(X, Y, is, os, long v, long vis, long vos)
d?tyvmc4(X, Y, is, os, long v, long vis, long vos, long vws, CPLEX *W)
```

The function call contains three additional parameters, which specify the number of vectors (loop iterations) `v`, and the distance between successive input and output vectors, `vis` and `vos`. In terms of functionality this codelet type is equivalent to the following code segment:

for $i = 0$ **to** v **do**
$ystart \leftarrow i \times vos$
$xstart \leftarrow i \times vis$
dfnynmc4$(X\,[xstart]\,, Y\,[ystart]\,, is, os)$
end

Although the same DFT is performed on all `v` vectors, each vector is multiplied by a different twiddle factor in twiddle codelets of this type. Contiguous twiddle factor arrays are expected to be stored at `vws` distance.

DXVMC: This codelet type is used in mixed-radix in-place algorithms. Straight-line code for this codelet type is not generated because multiple DFTs need to be performed in most cases. For each size n, four types are generated for different combinations of direction and twiddle flags. The calling interfaces of twiddle and non-twiddle codelets of size 4, are:

```
d?nxvmc4(X, is, dist, v, vis)
d?txvmc4(X, is, dist, v, vis, ws, W)
```

An in-place codelet performs n transforms of size n (non-contiguous 2D square blocks) and transposes the result to overwrite the input vector. `is` is the inter-element (row) stride in the vector (`X`), while `dist` is the distance between rows of the square blocks (assuming row major order). When multiple square blocks need to be transformed, `v` equals the number of iterations and `vis` specifies the stride between the first elements of square blocks. In terms of functionality, this codelet type is equivalent to the following code segment:

for $i = 0$ **to** v **do**
  $xstart \leftarrow i \times vis$
  dfnyvmc4$(X\,[xstart]\,, X\,[xstart]\,, is, is, 4, dist, dist)$
    `/* transpose matrix is=horizontal & dist=vertical stride */`
  transpose$(X\,[xstart]\,, is, dist)$
end

For twiddle codelets, each row of a square block is multiplied with the same twiddle array of size n, but multiple blocks (`v`>1) may be multiplied by different twiddle factor arrays, in which case a twiddle stride (`vws`>0) can be specified.

DYVPC: This codelet type is used in the Prime Factor Algorithm (PFA) [58, 62, 30]. Recall from the introduction to PFA in Chapter 3 that the algorithm does not require a twiddle multiplication step. Hence, a twiddle PFA codelet is not generated. Moreover, by taking advantage of special indexing, the PFA algorithm can generate output that is both in-place and in-order [13, 54]. However, in order for the algorithm to be more general, it requires special rotated codelets [54, 55]. The

rotation parameter depends on the co-prime factors $n_1, n_2 \ldots n_i$ of the FFT problem $N = \prod n_i$. For each factor n, j unique codelets are generated for all rotations r_j such that $0 < r_j < n$ and $gcd(r_j, n) = 1$. For example, if $n = 6$, two rotated codelets,i.e., $r_1 = 1$ and $r_2 = 5$ are generated. Note that a codelet of rotation $r = 1$ is equivalent to a regular mixed-radix codelet. The calling interfaces for PFA codelets of size 4, are:

```
dfnyvpc4r1(X, Y, is, os, v, vs)
dfnyvpc4r3(X, Y, is, os, v, vs)
```

The straight-line version of this codelet type `(dynpc)`, is not generated because a single rotated DFT is not required in the PFA algorithm. Furthermore, no inverse PFA codelet is generated because the direction of the PFA codelet is controlled through the rotation, i.e., if rotation $r_f = 1$ performs a forward rotated transform then $r_i = n - r_f = 3$ will perform an inverse rotated transform.

5.3.2 SIMD Codelets

For architectures (ISAs) that support SIMD extensions, exploiting architecture specific SIMD intrinsics is known to achieve better performance than scalar (non-SIMD) codes [32, 26, 22]. `fftgen` is capable of generating both single and double precision SIMD codelets. As per current microprocessor architecture trends, we generate SIMD codes for relatively short vectors of 128 bits (16 bytes), i.e., two double precision (DP) floats or four single precision (SP) floats. Such architectures also require that the data be aligned in memory at 16 byte boundaries for best performance.

Even though SIMD operations can potentially achieve double the performance of scalar operations, the code must exhibit the structure to enable paired arithmetic. In case of double precision complex-to-complex codelets, each complex element can be treated as a short vector and the SIMD operations can be performed on complex elements instead of separate real and imaginary parts. However, such a pairing may inhibit some of the arithmetic optimizations that could have been possible in scalar code. For example, let us consider a vector addition $\mathbf{y} = \mathbf{u} + i \times \mathbf{v}$, where $i = \sqrt{-1}$. Simple floating point (non-SIMD) code would require one addition and one subtraction:

$$y_r = u_r - v_i$$

$$y_i = u_i + v_r$$

while the SIMD version might require more than one operation to produce the result:

$$\mathbf{v} = \texttt{CONJUGATE}(\mathbf{v})$$

$$\mathbf{y} = \mathbf{u} + \mathbf{v}$$

where `CONJUGATE` is generally implemented using `shuffle` followed by `xor`.

For single precision SIMD codelets, two butterflies (at stride one) must be paired to perform the same operation on the two complex elements (four SP floats). `fftgen`'s flexible and modular design makes it easy to generate codes for different SIMD extension types. In fact, new extensions can be added simply by updating the macros in a header file. Some of these macros such as VLD, VST, VADD and VSUB map directly to the intrinsics or assembly instructions while others such as complex multiplication

(VZMUL) require multiple micro instructions to execute. Currently, `fftgen` generates optimized SIMD codes using SSE,SSE2 AND SSE3 extensions for Intel and AMD architectures and single precision SIMD, i.e., ALTIVEC for Power architectures.

ZYNMC: This codelet type is the SIMD-enabled (simdized) version of `dynmc`. Since only one butterfly is computed in `dynmc, synmc and zynmc`, there is no single precision counterpart of `zynmc`, i.e., there is no `qynmc` type codelet library, which requires paired butterfly computation. The calling interfaces of twiddle and non-twiddle codelets of size 4, are:

```
z?nynmc4(CPLEX *X, CPLEX *Y, long is, long os)
z?tynmc4(CPLEX *X, CPLEX *Y, long is, long os, CPLEX *W)
```

In the twiddle codelet, twiddle factors are expected to be stored in the same unit stride fashion as in `dynmc`. All other parameters also have the same interpretation as in scalar codelets. Figure 5.4 gives the listing of code generated for `zitynmc4` codelet.

ZYVMC: This codelet performs multiple transforms on equi-distanced vectors. The `zyvmc` codelet performs the same function as its scalar double precision counterpart. The calling interfaces of twiddle and non-twiddle codelets of size 4, are:

```
z?nyvmc4(X, Y, is, os, long v, long vis, long vos)
z?tyvmc4(X, Y, is, os, long v, long vis, long vos, long vws, CPLEX *W)
```

The parameters have identical interpretation as for `dyvmc`. However, the single precision SIMD codelet `qyvmc` differs significantly from `syvmc` and `zyvmc`. A `qyvmc` codelet call is equivalent to the following operations:

for $i = 0$ **to** v **do**
$ystart \leftarrow i \times vos$
$xstart \leftarrow i \times vis$
sfnynmc4$(X\,[xstart]\,, Y\,[ystart]\,, is, os)$
sfnynmc4$(X\,[xstart + 1]\,, Y\,[ystart + 1]\,, is, os)$
end

Two butterflies (odd and even) are computed together to fully utilize the length of a SIMD vector. Note that for the `qyvmc` codelet, `is,os,vis` and `vos` can not take odd values. Consider using `syvmc` instead if these restrictions can not be met. For twiddle codelets, the twiddle array is stored at unit stride, as usual. Internally, each twiddle load fetches both even and odd elements. For single precision SIMD calls, the twiddle array must be packed such that adjacent (paired) twiddle calls get the right set of twiddles.

ZXVMC: This codelet is used in the in-place in-order mixed-radix algorithms. It performs transforms on square blocks followed by a transpose as discussed previously. It is identical to `dxvmc` in terms of functionality and interface parameters. The calling interfaces of twiddle and non-twiddle codelets of size 4, are:

```
z?nxvmc4(X, is, dist, v, vis)
z?txvmc4(X, is, dist, v, vis, vws, W)
```

The single precision counterpart (`qxvmc`) of this codelet must meet the conditions as described above. In this case `is,dist,vis` and `vws` can not take an odd value.

ZYVPC: This codelet is identical to `dyvpc` in terms of functionality and interface parameters. It is used in the Prime Factor Algorithm, which requires special rotated codelets as described above. The calling interfaces of PFA codelets of size 4, are:

```
zfnyvpc4r1(X, Y, is, os, v, vs)
zfnyvpc4r3(X, Y, is, os, v, vs)
```

The single precision counterpart `qyvpc` of this codelet must meet the conditions as described above. In this case `is,os` can not take an odd value.

5.3.3 Direct Access to the Codelet Libraries

The codelet libraries generated at installation time are used in different driver routines of the UHFFT at run-time. The main user interface to the library is the DFTi API, which uses those driver routines to compute an FFT problem. Nevertheless, a user may bypass the DFTi API to access the codelets directly to solve a small FFT problem or use them in another run-time algorithm. Each codelet library can be accessed as an array of `Module` pointers. The PFA codelet library is a two dimensional array since each pointer points to an array of rotated modules of that size. There are two ways a user can get direct access to the codelets, as given below:

```
Module **DftiExGetLib(int LIBTYPE);
```

OR

```
extern Module *dynmcLib[];      /* SIMD and single precision
extern Module *dyvmcLib[];        codelets can be accessed
extern Module *dxvmcLib[];        in a similar manner */
extern Module **dyvpcLib[];
```

```
void zitynmc4(CPLEX *x, CPLEX *y, INT is, INT os, CPLEX *w)
{
     register V  tmp0, tmp1, tmp2, tmp3, tmp4, tmp5,
        tmp6, tmp7, tmp8, tmp9, tmp10, tmp11, tmp12;
     tmp0 = VLD(&x[0]);
     tmp1 = VLD(&x[is]);
     tmp2 = VLD(&x[2*is]);
     tmp3 = VLD(&x[3*is]);
     tmp4 = VADD(tmp0, tmp2);
     tmp5 = VSUB(tmp0, tmp2);
     tmp6 = VADD(tmp1, tmp3);
     tmp7 = VSUB(tmp1, tmp3);
     tmp8 = VCONJ(tmp7);
     tmp9 = VADD(tmp4, tmp6);
     tmp10 = VSUB(tmp4, tmp6);
     tmp11 = VADD(tmp5, tmp8);
     tmp12 = VSUB(tmp5, tmp8);
     VST(&y[0], VZMULI(tmp9, VLD(&w[0])));
     VST(&y[os], VZMULI(tmp11, VLD(&w[1])));
     VST(&y[2*os], VZMULI(tmp10, VLD(&w[2])));
     VST(&y[3*os], VZMULI(tmp12, VLD(&w[3])));
}
```

Figure 5.4: `zitynmc4` codelet

5.4 Code Generation Methodology

The UHFFT code generator (`fftgen`) has been implemented in "C" and its design allows it to be easily extended to other types of numerical codes. Apart from the complex-to-complex, real-to-complex and complex-to-real FFT codelets, the code generator was recently used to generate trigonometric transforms [5]. The formula of target numerical code is specified through a routine that describes the algorithm using scalar, vector or matrices of expressions. An expression can be one of seven basic types:

- Sum
- Product
- Negation (Complement)
- Assignment
- Variable
- Constant
- Block

For most types, the expression can take one or more (sub)expressions as operands. This allows recursive construction of expression lists using declarative style of programming. As an example, consider the following segment that generates code for an $O(n^2)$ DFT algorithm using matrix-vector multiplication.

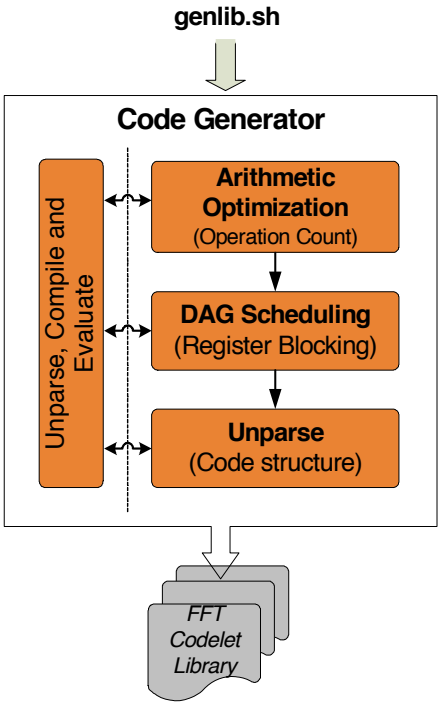


Figure 5.5: `fftgen` design diagram

```
ExprVec *out;
out = AssignExprVec( CPLEXOUTVAR,
       MultExprMatVec( GetDFTExprMat(n, FORWARD),
            GetExprVec( n, VAREXPR, CPLEXINVAR, 0)
       )
     );
```

The DFT matrix of rank n is multiplied with the input variable expression vector `X` and the resulting vector is assigned to the output variable expression vector `Y`. Using these built-in operators, a user may write driver routines to generate Basic Linear Algebra Subroutines (BLAS).

The code is generated in three main steps as shown in Figure 5.5. In each step, the optimization is performed independently. This is particularly helpful when automatic empirical optimization (AEOS) is enabled `--enable-aeos`, since it reduces the number of combinations that need to be evaluated. When the AEOS is disabled, `fftgen` uses default optimization parameters that are known to generate good code on most architectures.

5.4.1 Compiler Feedback Loop

The performance of generated codelets depends on many parameters that can be tuned to the target platform. Instead of using global heuristics for all platforms, the best optimization parameters can be discovered empirically by iteratively compiling and evaluating generated variants. The evaluation component, benchmarks the variants and tests the accuracy of generated code. In order to generate statistically stable measurements, each codelet is executed repeatedly. Codelets are called with non-unit strides keeping in view the context in which they are likely to be used as part of a larger FFT problem. The average of the performance measurements for different strides is used to represent the quality of a variant. The three main stages used to generate optimized code are discussed below.

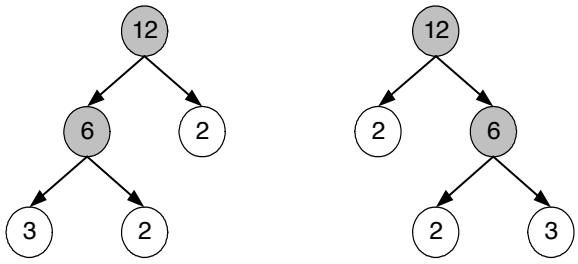
Stage 1: Formula Generation and Optimization

In the first stage, different FFT formulas are generated and evaluated to optimize for floating-point operation count. This phase generates the *butterfly computation*,

which is abstracted internally as a list of expressions. Simple arithmetic optimizations are applied to the list of expressions to minimize the number of operations. Four variants are generated depending on the factorization policy and Rader algorithm implementation as listed in Table 5.1. An illustration of two factorization policies is given in Figure 5.6.

Table 5.1: Stage 1 tuning parameters

Parameter	Value
Prime size algorithm	Partial or Full Skew Rader
Factorization policy	Right or Left recursive



(a) Left recursive (b) Right recursive

Figure 5.6: Factorization policies

The heuristics given in Figure 5.7 are used at each level of the factorization tree to select the algorithm and factors. In the simplest case when n is equal to 2 or when n is a prime number, the algorithm returns the size n as a factor, selecting an appropriate algorithm.

if $n = 2$ **then** $algo \leftarrow$ DFT and $r \leftarrow 2$

else if IsPrime(N) **then** $algo \leftarrow$ RADER and $r \leftarrow n$

else /* Use Heuristics */

$k \leftarrow$ GetMaxFactor (n)

$algo \leftarrow$ MR and $r \leftarrow k$

if GCD $\left(k\,,\,\frac{n}{k}\right) = 1$ **then**
$algo \leftarrow$ PFA and $r \leftarrow k$

else if $n > 8\,\&\,4 \mid n$ **then**
$algo \leftarrow$ SR and $r \leftarrow 2$

else if $n > k^3 and\ k^2 \mid n$ **then**
$algo \leftarrow$ SR and $r \leftarrow k$

end

if $FactorizationPolicy =$ RightRecursive **then** $r \leftarrow \frac{n}{r}$

Figure 5.7: FFT factorization and algorithm selection heuristics

Stage 2: Expression Scheduling

The second phase performs scheduling and blocking of expressions by generating a directed acyclic graph (DAG). The main purpose of this scheduling is to minimize register spills by performing a topological sort of instructions. Currently, only instructions within the same block are permuted, i.e., instructions are not moved outside the basic block scope even if it is safe to do so. However, larger basic blocks may be chosen in order to enhance the scope of the DAG scheduler. A study [3] conducted by the author revealed that even for very small straight-line codelets the performance variation due to varying instruction schedules could be as much as 7%.

The study also showed that different compilers responded differently to the same high level instruction schedule. In keeping with those findings, three different block sizes, 2,4 and 8, are tried as given by Table 5.2. In total, six different variants of schedules are generated and the best is selected after empirical evaluation.

Table 5.2: Stage 2 tuning parameters

Parameter	Value
Schedule	Cache-oblivious or CO+Topological sort
Basic Block	Size $\in$ {2,4,8}

Stage 3: Unparsing

In the third stage, the sorted expression list is unparsed (translated) to the target language. Although we generate code in "C", the unparser can be easily extended to output the code in other languages, including FORTRAN. Depending on the type of codelet, SIMD or scalar, different unparsers are used. For SIMD codelets, there is only one variant generated in this step. However, for scalar codelets six different variants are tried. The scalar codelets compute complex FFT on the input vectors, which are represented as arrays of structures. To generate code that results in the best performance, we tried two different representations for input and output vectors of complex type as given in Figure 5.8. In the first representation, input and output vectors are accessed as arrays of structure (`CPLEX`). In the second scheme, the complex vector is represented by two separate real and imaginary arrays of `REAL` data type. Apart from the two array translation schemes for complex type codelets,

two more variants are tried for all types of codelets, as given in Table 5.3. For some compilers, `gcc-3.4` for example, we noticed that an explicit step of replacing I/O vector elements in temporary scalar registers performed better in most cases. However, it did increase the total size of code in terms of lines of "C" code.

Table 5.3: Stage 3 tuning parameters

Parameter	Value
Scalar replacement	On or Off
I/O Data type	`CPLEX` or `REAL`

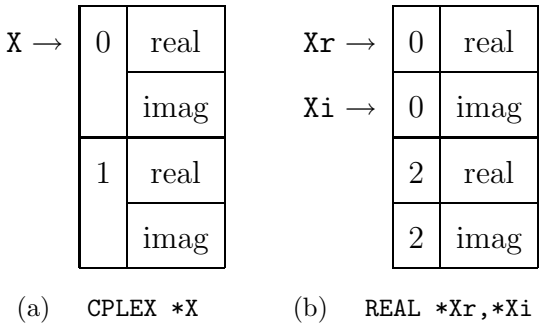


Figure 5.8: I/O data structure access

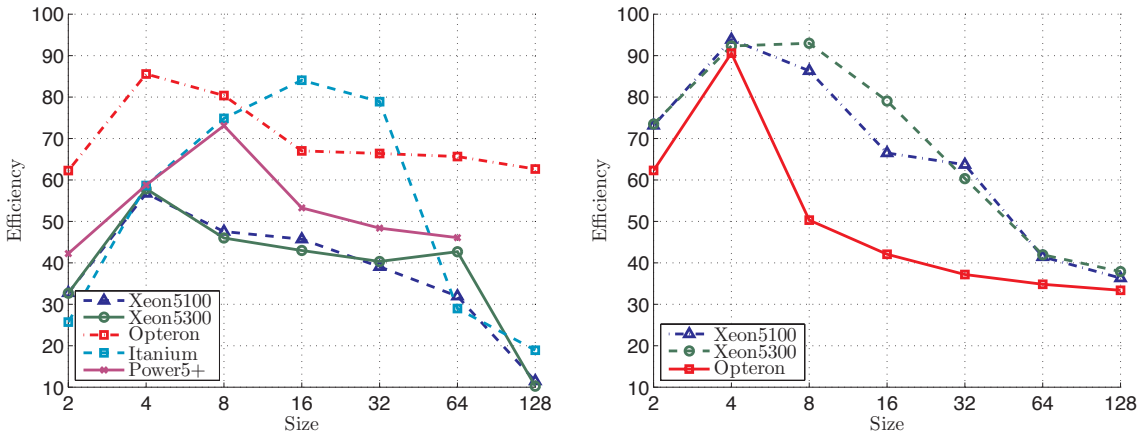
5.5 Results

In this section, we analyze the performance of generated codelets on some of the most recent architectures. For a complete list and specifications of the platforms, please refer to Section 4.1. The benchmarks are collected by running a codelet multiple times and taking the average as representative time of execution. Note that the timing results are expected to contain some error, especially for very small size codelets, where function call and loop iteration overhead could be significant compared to the actual floating point code. A detailed discussion on the benchmarking methodology is given in Section 4.3.

5.5.1 Selecting the Maximum Size of Codelets to Generate

In the first set of results, we benchmarked codelets of powers-of-two sizes for all target architectures. In each case, input and output blocks of memory equal to the size of 128 double precision complex elements were allocated to execute the maximum codelet size using unit stride. The plots in Figure 5.9 show that the performance of codelets increases with the size of the transform and then starts deteriorating once the code size becomes too big to fit in the instruction cache and registers. The performance increase is due to the fact that larger size codelets contain a sufficient number of arithmetic instructions to neutralize the overhead of a function call. Moreover, the codelets of size 2 and 4 do not contain any multiplication instruction, thus, only the addition functional unit(s) are utilized. Interestingly, the performance decline on the Opteron processor was not as sharp as found on other architectures. We

believe that it is due to the bigger instruction cache on the Opteron (64K) compared to the Itanium 2 (16K) and Xeons (32K). At installation time, appropriate sets of codelets can be selected by paying attention to the size of the instruction cache and the number of floating point registers. In the default set of codelets, size 64 is the largest codelet generated.



(a) Non-SIMD codelets

(b) SIMD codelets

Figure 5.9: Performance of powers-of-two size codelets (unit stride)

5.5.2 Performance of Single and Double Precision Codelets

In this section, we compare the performance of single and double precision codelets. When the architecture supports SIMD extensions, we present the performance of both SIMD and non-SIMD variants. We have selected only powers-of-two sizes because they are the most commonly used codelets. The benchmarking results are reported in pseudo-"MFlops", which is derived from the execution time and the standard floating-point complexity of radix-2 FFT algorithms, i.e., $5n\log(n)$. Note

that the actual number of operations is generally lower than the standard complexity.

Table 5.4: Single and double precision codelets performance (MFlops) on Xeon

Codelet	double		single	
Size	dyvmc	zyvmc	syvmc	qyvmc
2	3483	7783	3482	17673
4	6034	9987	6570	17126
8	5059	9185	5309	15389
16	4861	7075	5266	14032
32	4155	6776	5105	12527
64	3400	4408	4614	7175
128	1220	3865	1200	6618

(a) Woodcrest 2.66GHz

Codelet	double		single	
Size	dyvmc	zyvmc	syvmc	qyvmc
2	3046	6848	3049	15467
4	5392	8604	5147	14951
8	4284	8664	4149	15061
16	4006	7367	4335	12327
32	3759	5624	4220	10563
64	3978	3910	4175	6292
128	954	3530	1015	5759

(b) Clovertown 2.33GHz

Intel Xeon Woodcrest

Each core of the Xeon Woodcrest is capable of executing two 128-bit floating point operations (flops) each cycle, i.e, four double precision or eight single precision flops can be executed in parallel if the right blend of SIMD instructions is available. However, as discussed in previous chapters, achieving theoretical peak performance is generally not possible even though the compiler may be able to simdize some part of the "C" code. Table 5.4 gives the performance results of both single and double precision codelets of powers-of-two sizes, where `dyvmc` and `zyvmc` refer to double

precision floating point and SIMD codelets respectively. Similarly, `syvmc` and `qyvmc` refer to single precision variants. As shown in the table, the scalar (non-SIMD) code is considerably slower than the SIMD code, which computes an operation on the full vector length, potentially doubling the performance. Note that single precision floating point (non-SIMD) code does not achieve any meaningful speedup over its double precision counterpart. As the size of the codelets increase, so does the register pressure, which impacts the performance of both SIMD and non-SIMD codelets. Also note that as the size of the codelets increase, the performance benefit of SIMD code over the non-SIMD variant decreases except for size 128 codelets. The reduction in performance benefit is due to the fact that operating on SIMD vectors sometimes inhibits the performance optimizations that would otherwise be possible in scalar (non-SIMD) code as discussed in Section 5.3.2. This results in higher than the ideal (half) number of operations in SIMD code. On the other hand, the performance of a very large size codelet (`dfnyvmc`128 contains approximately 3K line of "C" code), improves drastically due to better density found in the simdized codes.

AMD Opteron 285

Compared to the Xeon Woodcrest, the second generation Opteron did not achieve significant performance improvement due to the simdization of codelets, as shown in Table 5.5. This behavior is expected and has been explained in Section 4.1. According to the AMD64 manual [8], the double precision SIMD code is not likely to achieve any improvement because the instruction is internally broken into scalar operations. As a matter of fact, the double precision SIMD code turned out to

Table 5.5: Single and double precision codelets performance (MFlops) on the Opteron

Codelet	double		single	
Size	dyvmc	zyvmc	syvmc	qyvmc
2	3238	3238	2507	6476
4	4449	4710	3813	8813
8	4178	2615	3644	4534
16	3483	2187	3200	4054
32	3451	1933	3182	3650
64	3413	1811	2950	3449
128	3257	1735	2864	3158

be slower than the (non-SIMD) double precision scalar code. As explained in the previous sections, the slowdown is due to missed optimization opportunities as a result of vector operations. This behavior is somewhat evident in Table 5.10, which shows that the number of measured floating point operations of the SIMD code is higher than that of the scalar code. Also notice that the SIMD code contains lower number of instructions than the scalar code. It is not clear why the number of operations in the "C" code and the "PAPI" measured counts are different for the Opteron.

Itanium 2 (Madison)

In Table 5.6, we compare the performance of single and double precision code generated on the Itanium 2. Notice that the performance is almost identical in both cases, which is expected since each register can hold a single floating point element; there

Table 5.6: Single and double precision codelets performance (MFlops) on Itanium 2

Codelet Size	`dyvmc`	`syvmc`
2	1543	1216
4	3519	3741
8	4491	4490
16	5044	5042
32	4734	4676
64	1738	1811
128	1135	1126

is no SIMD floating point unit. However, the Itanium 2 supports fused multiply and add (FMA) operations. Hence, the efficiency of code depends on the instruction mix and compiler's ability to generate FMA instructions from straight-line code.

IBM Power5+

The Power5+ architecture also contains two FMA units each capable of performing two arithmetic operations per cycle. As expected, there is no significant performance improvement in single precision code compared to double precision as shown in Table 5.7.

Table 5.7: Single and double precision codelets performance (MFlops) on Power5+

Codelet Size	`dyvmc`	`syvmc`
2	3210	3210
4	4465	4470
8	5553	6331
16	4047	4221
32	3675	3708
64	3501	3488

5.5.3 Performance Impact of Empirical Optimization (AEOS)

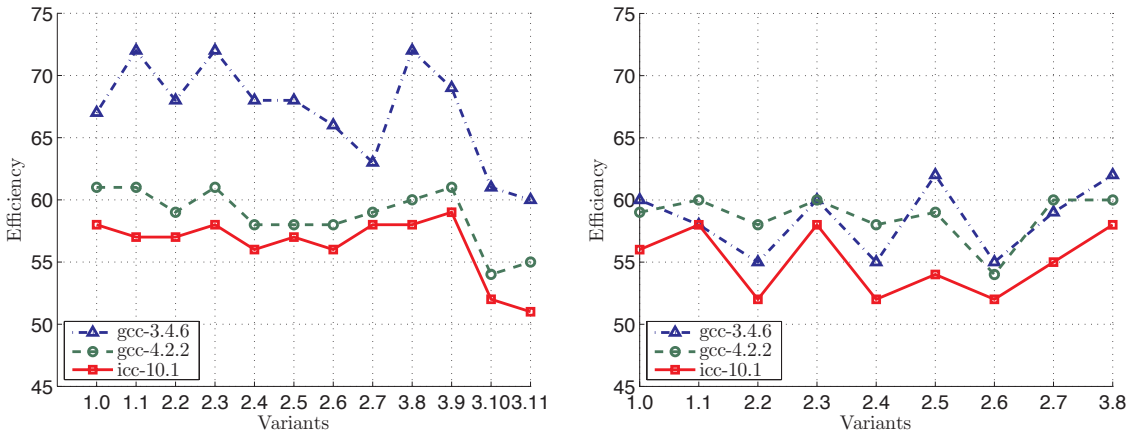
In Section 5.4.1, we discussed the code generation methodology implemented in `fftgen`. The methodology employs aggressive empirical optimization technique using a compiler feedback loop to adapt to both the architecture and the compiler. In this section, we evaluate the effectiveness of this approach versus using hard-coded (default) values for tuning parameters. Note that when AEOS is enabled (`--enable-aeos`), the optimization starts with the default values for the parameters as given in Table 5.8. The default values have been selected based on testing of various parameter combinations. Figure 5.10 shows the variation in performance in each optimization stage for codelets of size 16 on the Opteron 285 processor. First, notice that `gcc-3.4.6` performs better than both `gcc-4.2.2` and `icc-10.1` compilers on the Opteron 285 processor. Also note that the performance variation due to the compiler feedback loop in the SIMD codelets is lower than for the scalar codelets. This is expected because SIMD codelets are generated using architecture

intrinsics, which translate directly into assembly, obviating the need for extensive compiler optimization. Since the empirical optimizations include all combination of

Table 5.8: Tuning parameter values for the codelet `dfnyvmc16`

Stage	#	Factor. Policy	Rader Algorithm	Instruction Schedule	Block Size	Scalar Repl.	I/O Array
1	0	**RRF**	**SkewP**	CO+TS	2	Off	Cplex
1	1	**LRF**	**SkewP**	CO+TS	2	Off	Cplex
2	2	LRF	SkewP	**CO**	**2**	Off	Cplex
2	3	LRF	SkewP	**CO+TS**	**2**	Off	Cplex
2	4	LRF	SkewP	**CO**	**4**	Off	Cplex
2	5	LRF	SkewP	**CO+TS**	**4**	Off	Cplex
2	6	LRF	SkewP	**CO**	**8**	Off	Cplex
2	7	LRF	SkewP	**CO+TS**	**8**	Off	Cplex
3	8	LRF	SkewP	CO+TS	2	**Off**	**Cplex**
3	9	LRF	SkewP	CO+TS	2	**Off**	**Real**
3	10	LRF	SkewP	CO+TS	2	**On**	**Cplex**
3	11	LRF	SkewP	CO+TS	2	**On**	**Real**

tuning parameters (including default values), we expected the performance to be at least as good as that with hard-coded values. Figure 5.20 shows that to be true in most cases; however, in some cases the empirical optimization did not select the best combination of parameters. On most clusters, the compilers and build tools are only available on the login node(s), which are shared by all users of the system. Since the code is generated at installation time on login nodes, the timing results are not expected to be accurate in all cases. In the future, we plan to use the median of the



(a) Non-SIMD codelet

(b) SIMD codelet

Figure 5.10: Performance variation in the codelet `dfnyvmc16` due to compiler feedback. The experiment was performed on the Opteron 285 using three compilers

execution times instead of the mean to avoid the statistical noise in benchmarks.

5.5.4 Codelets Performance Models

A large size FFT problem is composed of many small codelets, which access the input and output memory at non-unit stride. Therefore, it is important to evaluate the codelets' performance for a range of input and output strides. For all platforms considered, the performance decreases considerably for large data strides. If two or more data elements required by a particular codelet are mapped to the same physical block in cache, then loading one element results in the expulsion of the other from the cache. This phenomenon known as *cache thrashing* occurs most frequently for strides of data that are powers-of-two because data that are at such strides apart are mapped to the same physical blocks in cache depending on the type of cache that is

used by the architecture. On a typical machine with simple cache design, thrashing occurs for each level of cache when:

$$\begin{aligned}
\frac{stride \times size_{datapoint} \times size_{codelet}}{size_{block}} &> sets_{cache} \\
\frac{stride \times size_{datapoint} \times size_{codelet}}{size_{block}} &> \frac{blocks_{cache}}{associativity} \\
\frac{stride \times size_{datapoint} \times size_{codelet}}{size_{block}} &> \frac{size_{cache}}{size_{block} \times associativity} \\
stride &> \frac{size_{cache}}{associativity \times size_{datapoint} \times size_{codelet}}
\end{aligned} \tag{5.1}$$

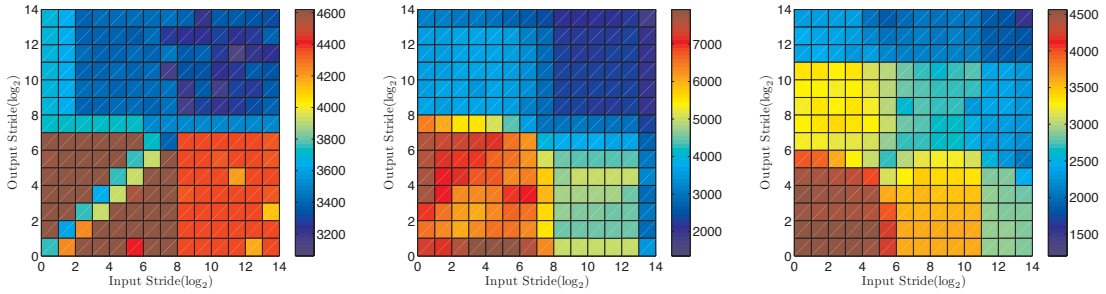
where $size_{cache}$ is the size of cache, $associativity$ is the cache associativity, $size_{datapoint}$ is the size of each element in the vector; for double precision, complex type, the $size_{datapoint}$ is equal to 16 bytes. The main objective of this experiment was to develop performance models that can help in the selection of factors at various levels of the factorization tree.

Intel Xeon Woodcrest

Both the Xeon Woodcrest and Clovertown possess similar cache hierarchy, i.e., two levels of cache, where the L1 cache size and associativity is 32K and 8 respectively. The L2 cache size for Woodcrest and Clovertown is 4M per dual and 8M per quad respectively, while the associativity is 16 for both Xeons. Figures 5.11 and 5.12 show the performance variation of three codelets as a function of input and output strides. Only SIMD codelets have been selected because scalar codelets are significantly slower on Xeons. Note that input and output vectors are separate arrays. Both architectures exhibit similar behavior as a function of strides, which is expected

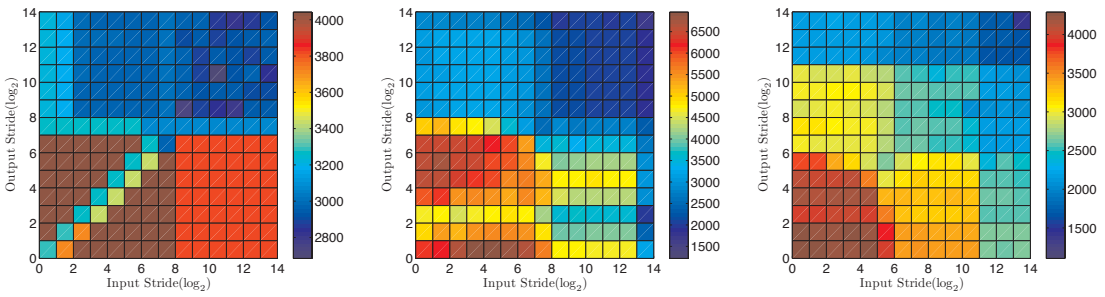
because Clovertown is based on the Woodcrest architecture. Observe that the performance of codelets exhibits a plateau-like pattern, which indicates cache thrashing at multiple levels of the cache hierarchy. For smaller codelets, we also observed that when input and output strides were equal, the performance dropped dramatically as indicated by the diagonal pattern for the codelet of size 4. We believe that this behavior is due to conflict between input and output vectors. Since both are aligned to 128-bit boundaries in the main memory, it is likely that the vectors get mapped to same addresses in the cache causing cache misses.

Figure 5.13 shows the performance of powers-of-two size codelets for various strides. In this experiment, the input and output strides are equal and both vectors point to the same location. Given a power of two size FFT, the size $N = 2^i$ can be factorized using any mix of factors shown in the surface plot. In a greedy algorithm, the most efficient codelet (size 8 in Figure 5.13) can be selected as long as it divides the size of the transform.



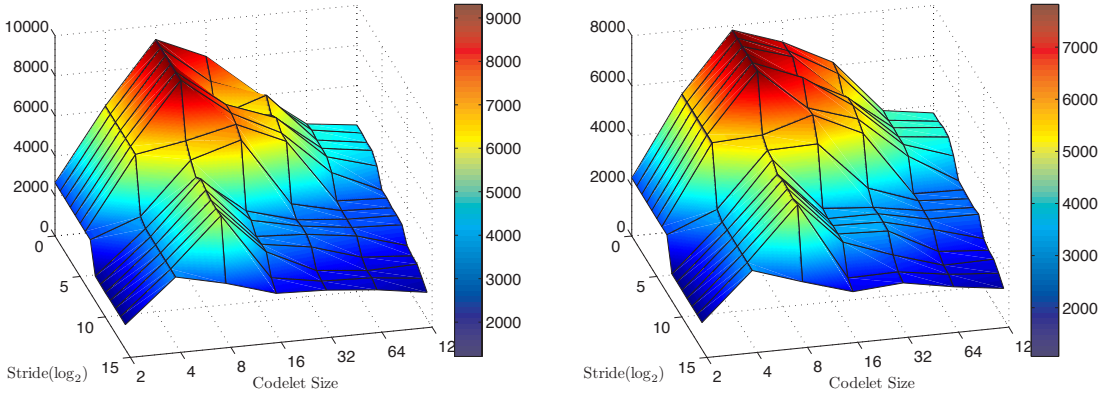
(a) `zfnyvmc4` (b) `zfnyvmc16` (c) `zfnyvmc64`

Figure 5.11: The Xeon Woodcrest 5100 performance model for a range of strides



(a) `zfnyvmc4` (b) `zfnyvmc16` (c) `zfnyvmc64`

Figure 5.12: Xeon Clovertown 5300 performance model for a range of strides



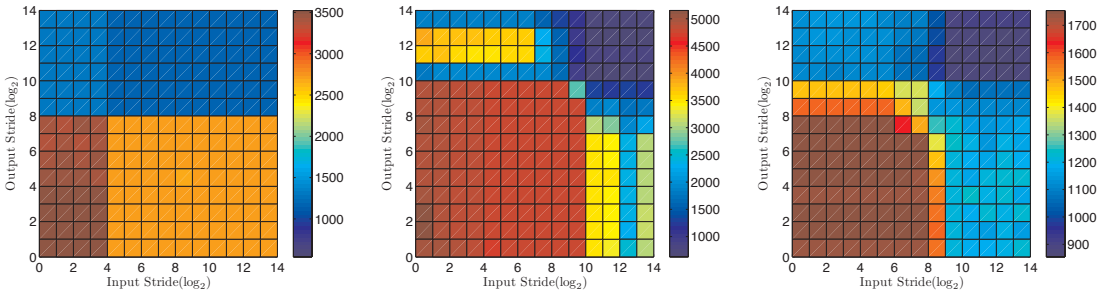
(a) Xeon5100 (b) Xeon5300

Figure 5.13: Xeon: Performance variation of powers-of-two size codelets as a function of stride ($input = output$)

Intel Itanium 2

Itanium 2 contains three levels of cache, but only the second and third levels are used for floating point data. Both the L2 and L3 caches are bigger that the L1 and L2 caches of Xeon. As shown in Figure 5.14, this results in wider performance plateaus. Due to the larger size of the register file, the best performing codelets are bigger

than for the Xeons. As shown in Figure 5.15, the peak for the Itanium 2 is for the codelets of size 16 and 32, whereas for the Xeon this peak occurs for size 8.



(a) `dfnyvmc4` (b) `dfnyvmc16` (c) `dfnyvmc64`

Figure 5.14: Itanium 2 cache performance model for a range of strides

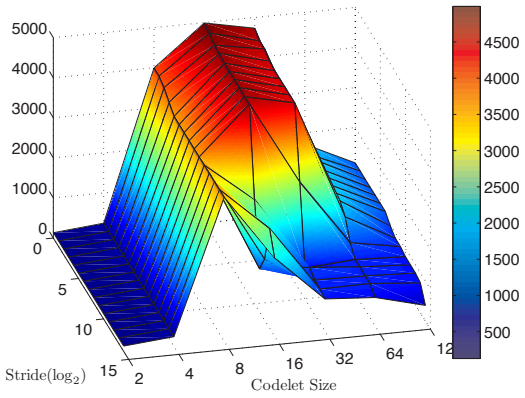
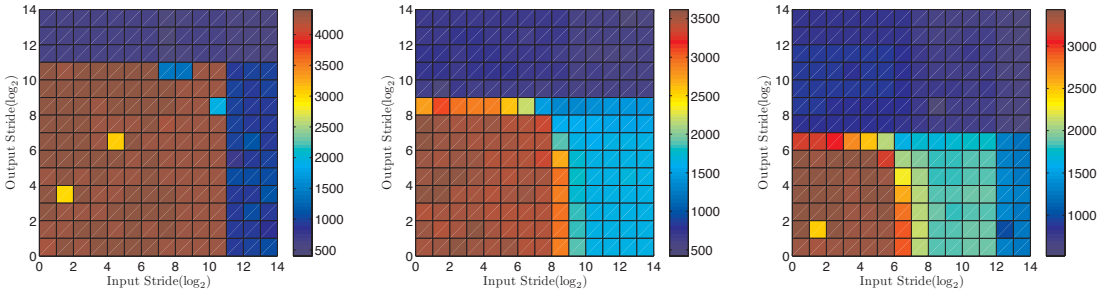


Figure 5.15: Itanium 2: Performance variation of powers-of-two size codelets as a function of stride ($input = output$)

AMD Opteron 285

The cache performance on Opteron was observed to be more predictable than for the other architectures. Indeed, the model given in Eq. (5.1), can be used to predict the stride boundaries of each level of cache, as shown in Figure 5.16. Similarly, the performance model given in the surface plot (Figure 5.17), shows that as we increase the size of codelets by a factor of 2, the boundary (given by stride) recedes by the same factor. Observe that the best performing codelet for the Opteron 285 processor is the same as for the Xeon (size 8), which is expected because both architectures have the same size of the register file. Unlike for other architectures, larger codelets do not suffer major performance degradation for the Opteron 285 processor, as shown in Figure 5.17. This could be because of the larger size of the instruction cache.



(a) `dfnyvmc4` (b) `dfnyvmc16` (c) `dfnyvmc64`

Figure 5.16: Opteron 285 cache performance model for a range of strides

IBM Power5+

The Power5+ not only contains a bigger register file compared to both the Opteron and Xeon, it also contains bigger instruction cache (64K). This is indicated in Figure

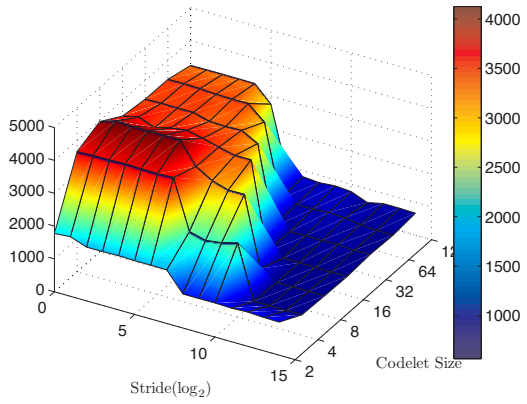
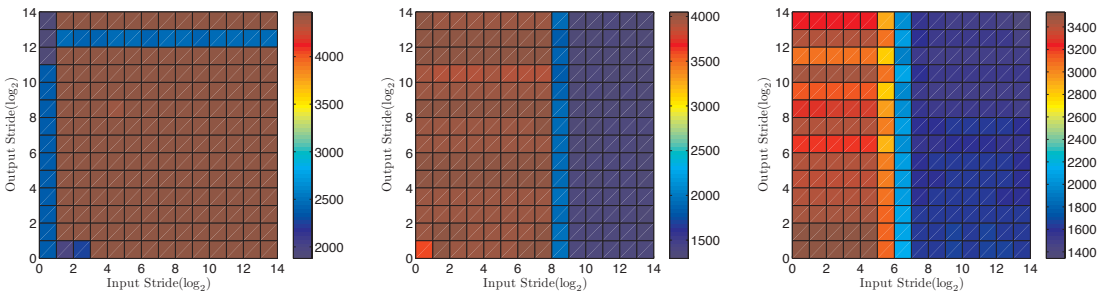


Figure 5.17: Opteron 285: Performance variation of powers-of-two size codelets as a function of stride ($input = output$)

5.19 by a slightly flatter surface as we increase the size of the codelets. One of the most striking differences in the performance of the Power5+ compared to the other architectures can be witnessed in Figure 5.18. Notice that the performance difference due to increase in input and output stride is not identical. Increasing the input stride causes a performance decrease but increasing the output stride does not create a performance decrease. We believe this is due to the fact that the Power5+ implements a write-through cache unlike other architectures discussed here. When the output is generated, it is written directly to memory.

In the discussion above, we investigated the performance of various codelets as a function of strides. When the stride is sufficiently small, the input memory block fits in the higher levels of cache, resulting in good efficiency. However, for larger strides cache conflicts cause thrashing resulting in a dramatic decrease in performance. In order to avoid thrashing, a data reordering step (transpose) may be performed. As



(a) `dfnyvmc4` (b) `dfnyvmc16` (c) `dfnyvmc64`

Figure 5.18: Power5+ cache performance model for a range of strides

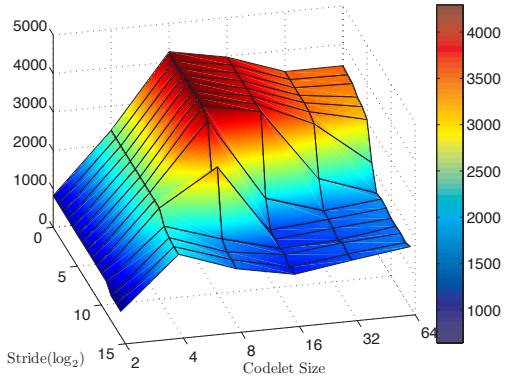
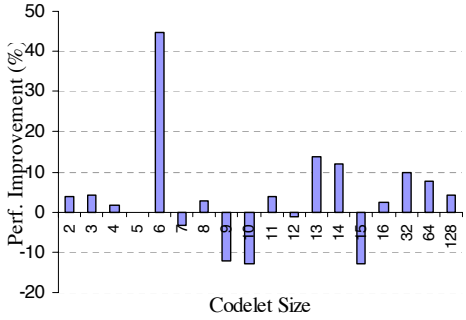


Figure 5.19: Power5+: Performance variation of powers-of-two size codelets as a function of stride ($input = output$)

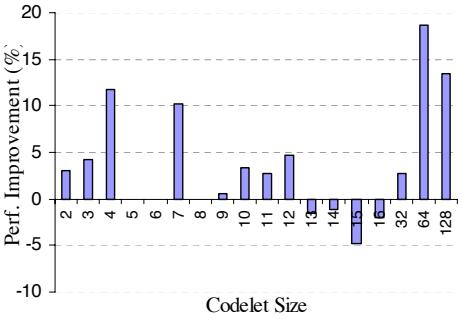
long as it amortizes the cost associated with the cache misses, a performance benefit can be achieved. We also observed that complex cache architectures make it hard to come up with a unified model for predicting the cache performance as a function of stride, codelet size and limited cache parameters. Thus, instead of relying on cache parameters, we will record the stride parameters (for all levels of cache) directly for each architecture through one-time empirical analysis.

For each architecture, some codelets are more efficient than others. The codelets of medium size generally performed best because they contained sufficient workload to amortize the calling overhead. The larger size codelets utilize both addition and multiplication units because they contain a mix of both operations unlike codelets of size 2 and 4. By selecting the best performing (factor) codelet in a greedy fashion followed by fast transpose, we can find the best combination of codelets to solve a larger size FFT. However, due to the high cost of data movement on modern architectures, such reordering rarely achieves much performance benefit. When an efficient transpose is not available, a greedy algorithm may be used for larger strides as well. In the surface plots, it can be observed that when the stride becomes sufficiently large to fit in any level of cache, the fastest codelet is typically equal to the associativity of the fastest cache, i.e., size 8 for Xeons and Itanium, size 2 for Opteron and size 4 for Power5+. This is because when the size of a codelet is smaller than or equal to the cache associativity, the conflicts are unlikely to occur because every element fits in the set. This behavior will be revisited in Chapter 7.

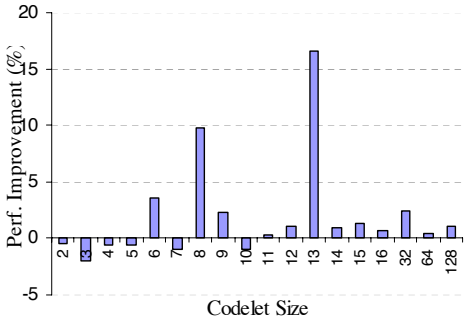
(a) Xeon5100



(b) Xeon5300



(c) Itanium 2



(d) Opteron

(e) Power5

Figure 5.20: Performance improvement due to AEOS

Table 5.9: Instruction Counts on Xeon

Codelet Size	"C" Code Instr. Count			"PAPI" Counts (Measured)			
				Scalar Code		SIMD Code	
	FP Adds	FP Mults	FP Total	OPS	INS	OPS	INS
2	4	0	4	4	4	2	2
3	12	4	16	16	16	8	8
4	16	0	16	16	16	8	8
5	32	12	44	44	44	22	22
6	36	8	44	44	44	22	22
7	60	36	96	96	96	48	48
8	52	4	56	56	56	30	30
9	80	40	120	120	120	60	60
10	84	24	108	108	108	54	54
11	140	100	240	240	240	120	120
12	96	16	112	114	114	58	58
13	216	76	292	294	294	148	148
14	148	72	220	224	224	114	114
15	156	56	212	216	216	110	110
16	144	24	168	172	172	92	92
32	372	84	456	476	476	258	258
64	912	248	1160	1220	1220	668	668
128	2164	660	2824	2990	2990	1630	1630

Table 5.10: Instruction Counts on Opteron

Codelet Size	"C" Code Instr. Count			"PAPI" Counts (Measured)			
				Scalar Code		SIMD Code	
	FP Adds	FP Mults	FP Total	OPS	INS	OPS	INS
2	4	0	4	6	14	6	7
3	12	4	16	21	34	27	19
4	16	0	16	24	40	29	22
5	32	12	44	61	86	72	47
6	36	8	44	61	86	72	50
7	60	36	96	122	179	151	95
8	52	4	56	78	124	120	102
9	80	40	120	162	221	224	174
10	84	24	108	146	216	174	111
11	140	100	240	271	447	444	435
12	96	16	112	155	229	195	125
13	216	76	292	373	568	556	390
14	148	72	220	285	439	470	369
15	156	56	212	283	412	351	221
16	144	24	168	222	360	341	313
32	372	84	456	573	955	908	856
64	912	248	1160	1410	2440	2300	2200
128	2164	660	2824	3320	5970	5550	5250

Chapter 6

Dynamic Code Schedules for FFT Computation

6.1 Introduction

The UHFFT employs a two layered methodology to adapt FFT computations to a given architecture and dataset. The previous chapter described the generation of FFT microkernels, which are optimized for microprocessor architectures by reducing the number of floating point operations as well as register spills. At run-time, codes generated by `fftgen` are assembled into a schedule to compute FFTs of sizes larger than generated by `fftgen`. An FFT problem (descriptor) can be computed in several different ways due to the recursive, divide and conquer nature of FFT algorithms. Apart from the factorization trees and algorithms, placement and ordering of input and output data vectors play an important role in the selection of schedules. Before

we discuss the mechanics of FFT schedules, we devise a strategy to express the FFT schedules in a compact representation. *"Unfortunately, the simplicity and intrinsic beauty of many FFT ideas is buried in research papers that are rampant with vectors of subscripts, multiple summations, and poorly specified recursions. The poor mathematical and algorithmic notation has retarded progress and has led to a literature of duplicated results"* [37]. The most commonly used, "Kronecker product" formulation of FFT is an efficient way to express sparse matrix operations. SPIRAL [44] generates optimized DSP codes using a pseudo-mathematical representation that is based on the Kronecker product formulation. FFTW [25] uses an internal representation of DFT problems using various data structures such as, I/O tensors and I/O dimensions etc. Although they offer an efficient design to solving a complex problem, these representations do not provide sufficient abstraction to the mathematical and implementation level details of FFT algorithms. One of the standard ways to visualize an FFT computation is through a signal flow diagram called the "butterfly" representation. This representation has been adopted universally by various disciplines of science. Though such graphical representation can help the understanding of control and data flow for small problems, a formal description is necessary for machine manipulation, optimization and code generation. The UHFFT run-time system implements a limited set of rules that can be used to express a wide range of serial and parallel FFT schedules. In order to express those schedules, we have designed a language called FFT schedule specification language (FSSL), which is generated from a set of context free grammar (CFG) productions. The grammar provides a direct and compact translation of the FFT butterfly representation and is easy to

understand for computer scientists. It also facilitates the machine manipulation of expressions, optimizations and code generation.

6.2 The FFT Schedule Specification Language

An execution schedule determines the codelets that will be used for the computation and also the order (schedule) in which they will be executed. An FFT schedule is described concisely using the grammar given in Table 6.1. It allows different algorithms to be mixed together to generate a high performance execution schedule based on properties of the I/O vector including the size and its factors. Indeed, by implementing a minimal set of rules, dynamic schedules can be constructed that suit different types of architectures.

6.3 One-dimensional Serial FFT Schedules

The UHFFT supports both unordered and in-order computation of the FFT, i.e., the output can be generated in a bit-reversed or sorted order. Computation of FFT in scrambled order derives naturally from the mixed-radix algorithm. We focus our attention on the in-order FFTs because that is the most prevalent form of FFT computation.

Table 6.1: FFT Schedule Specification Language grammar

#	CFG Rules
1-2	ROOT → MULTIDFFT | PARALLELFFT
3-4	MULTIDFFT → [NDFFT, FFT] | FFT
5-6	NDFFT → NDFFT, FFT | FFT
7	PARALLELFFT → (mrp$P$, MULTIDFFT)N
8-9	FFT → FFT mr MODULE | MODULE
10-12	MODULE → (rader$N$, MODULE)
	| ORDEREDFFT | CODELET
13-15	ORDEREDFFT → (outplace$N$, FFT)
	| (inplace$N$, FFT)
	| (pfa$N$, PFAMODULE pfa CODELET$'$)
16-17	PFAMODULE → PFAMODULE pfa CODELET$'$ | CODELET$'$)
18	CODELET → n
19	CODELET$'$ → n^{rot}

6.3.1 Out-of-place In-order Mixed-radix FFT

Recall from Chapter 3 that the mixed-radix FFT algorithm requires an explicit digit reversal step to bring the output "in order". When a separate output vector is available (`DFTI_PLACEMENT = DFTI_NOT_INPLACE`), the permutation can be fused in the first rank of FFT computation as shown in Figure 6.1. Notice that the ranks 2 through $\log(n)$ are computed in-place with the same I/O vectors and strides. Typically, these in-place ranks are small codelets pre-generated at the installation time. However, it is possible to use any MODULE production in the grammar shown in Table

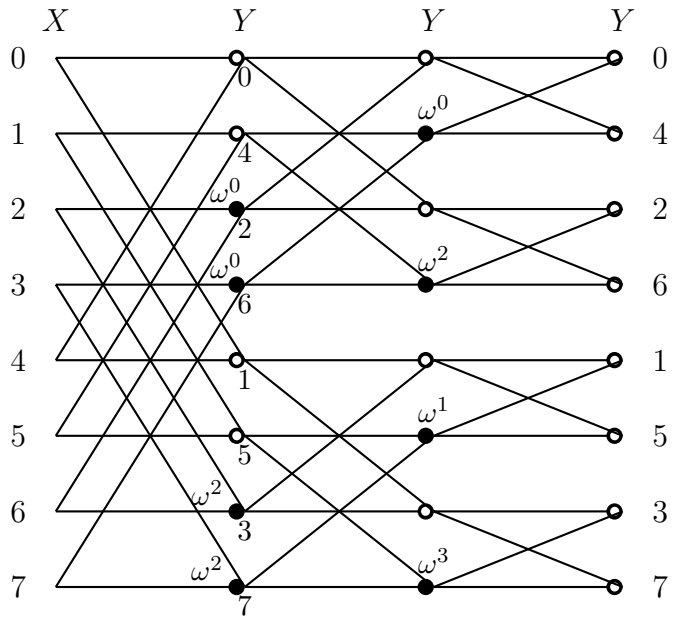


Figure 6.1: Butterfly diagram of 8-point out-of-place in-order FFT

6.1, which generates an ordered output using output or temporary workspace. Notice that an "out-of-place in-order" schedule may contain other schedules that implement a different algorithm. In fact, an `outplace` micro-schedule may be embedded inside a larger `outplace` schedule. To illustrate that, let us consider an example of size 8 out-of-place in-order FFT. The schedule depicted by Figure 6.1 can be expressed in FSSL as follows:

```
Schedule 0: (outplace8,2mr2mr2)
```

The same schedule can also be expressed as:

```
Schedule 1: (outplace8,(outplace4,2mr2)mr2)
```

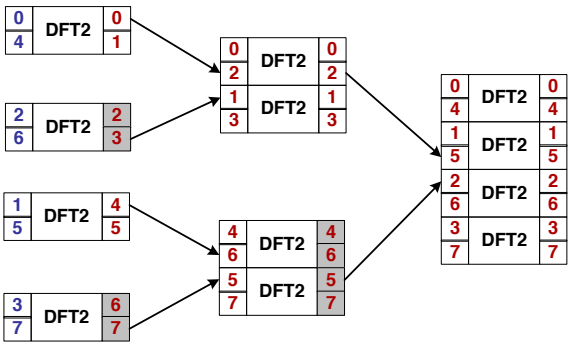


Figure 6.2: Codelet call pattern for `(outplace8,(outplace4,2mr2)mr2)`

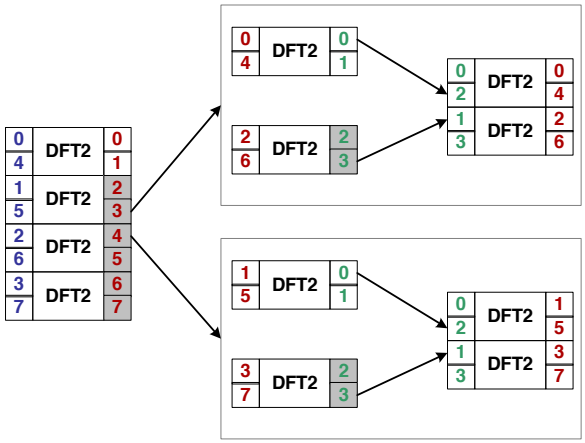


Figure 6.3: Codelet call pattern for `(outplace8,2mr(outplace4,2mr2))`

which uses the same call pattern to perform the FFT as shown in Figure 6.2. However, the following schedule computes the same FFT but uses a different access pattern and indeed results in different number of floating point operations:

`Schedule 2: (outplace8,2mr(outplace4,2mr2))`

The codelet call pattern for this schedule is shown in Figure 6.3. Although both schedules consist of the same number of codelet calls, the second schedule requires five

twiddle codelet calls compared to four calls in the first schedule. Moreover, the first schedule uses a depth first schedule, which results in better cache locality compared to the second schedule, which computes the first rank completely before moving to the next rank. Note also that the second schedule requires a temporary workspace to store intermediate result for computing `(outplace4,2mr2)`. Due to these drawbacks, the second schedule is likely to perform poorer than the first schedule. Nonetheless, embedded `outplace` schedules (also called *replace* schedules) may be unavoidable in the computation of "in-place in-order" FFTs.

Twiddle Factors Array Layout

The mixed-radix FFT algorithm requires multiplication by a diagonal matrix of twiddle factors. The array of twiddle factors is generally precomputed and stored in a table to be reused during the butterfly computation. As shown in Figure 6.1, the set of twiddle factors required by lower ranks is a subset of the factors required by higher ranks of the butterfly. Sharing of twiddle factors between ranks results in smaller memory footprint but introduces strided access in the precomputed twiddle table. As a tradeoff, copies of the twiddle factors may be stored at unit stride at the expense of additional memory. For example, let us consider a radix-4 FFT of size 64 given by `(outplace64,4mr4mr4)`. The butterfly of this schedule requires, $(4-1) \times 16 = 48$ twiddle multipliers between the second and third ranks and $(4-1) \times 4 = 12$ twiddle multipliers between the first and second ranks (for size 16 sub-FFT). The access pattern of the twiddle factors in the two ranks is given in Figure 6.4. The figure shows exponents of the twiddle factor ω_{64} reordered to enable unit stride access. Notice

that the twiddles in the 2nd rank may be reused in the first rank at a vector stride of 16. We evaluated the impact of compressed and replicated twiddle factor arrays on the performance of the FFT execution. As shown in Figure 6.5, three different size FFTs were evaluated on two architectures. In each case only size 4 codelets were used. The amount of twiddle factors sharing is given by horizontal axes. Notice that, using replicated twiddle factors results in better performance because each rank owns its own copy of twiddles that can be accessed at unit stride. In the UHFFT, we have disabled sharing of twiddle factors between ranks to maximize the performance.

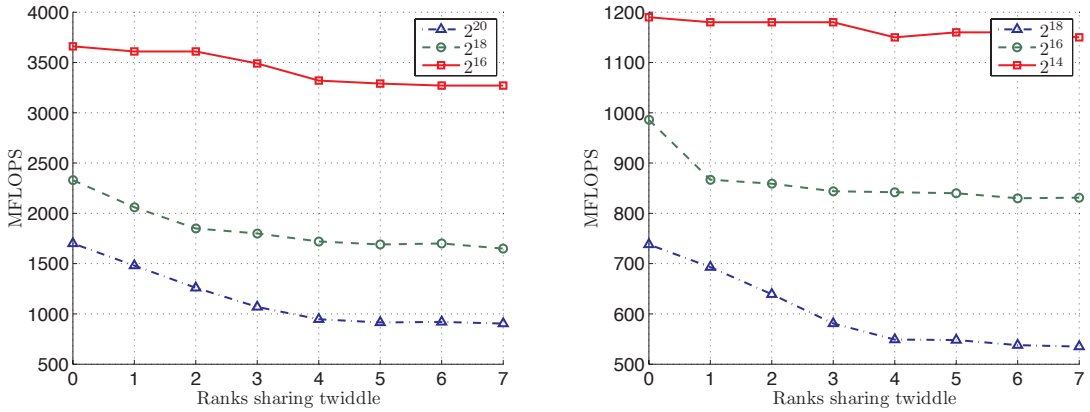
0	4	8	12
0	8	16	24
0	12	24	36

(a) 1st Rank

0	4	8	12	0	8	16	24	0	12	24	36
1	5	9	13	2	10	18	26	3	15	27	39
2	6	10	14	4	12	20	28	6	18	30	42
3	7	11	15	6	14	22	30	9	21	33	45

(b) 2nd Rank

Figure 6.4: Access pattern of twiddle factors in a radix-4 FFT of size 64. The twiddle codelets are executed in ranks 1 and 2



(a) Xeon Clovertown

(b) Opteron 285

Figure 6.5: Performance variation due to memory layout of twiddle factors

6.3.2 In-place In-order Mixed-radix FFT

Performing the computation "in-place", i.e. without a separate output vector or temporary array, poses a very difficult problem for self-sorting FFTs. For most transform sizes, data reordering is unavoidable in computing in-place and in-order FFTs due to lack of a separate workspace. For sizes N that can be factorized in a "palindrome", i.e., $N = r \times m \times r$ or $N = r \times r$, in-place in-order FFT can be computed by performing partial digit-reversal in each of the first $\lfloor \frac{\log(n)}{2} \rfloor$ ranks as shown in Figure 6.6. In the UHFFT, specialized codelets (`xvmc`) are generated that perform this reordering inside the codelet by computing the DFT on a square block followed by a transpose. This scheme saves loads and stores since the result is written to the proper location (input vector) inside the codelet. This saving can only be made if the required codelet has been pre-generated. If a factor r has not been generated, it needs to be dynamically constructed using one of the production rules

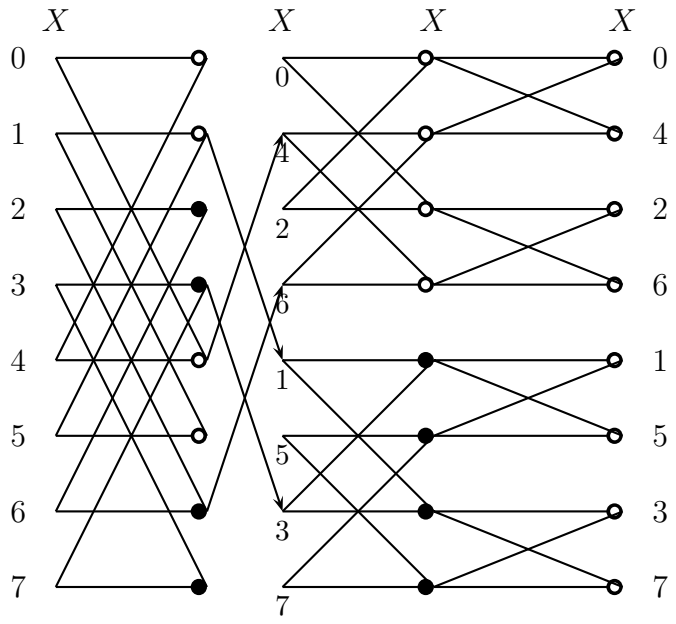
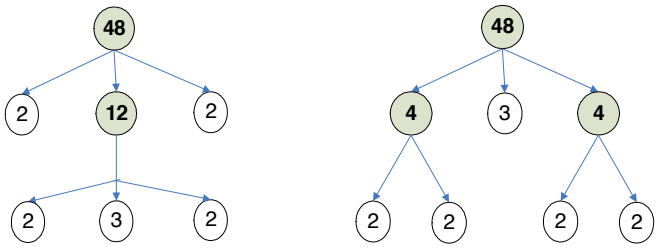


Figure 6.6: Butterfly diagram of 8-point in-place in-order FFT

(10-15) of the FSSL grammar (Table 6.1), in which case, the transpose is performed as a separate step. Like the `outplace` schedules, `inplace` schedules can be embedded within larger (container) schedules.

For a given transform size, many palindrome factorizations exist. Figure 6.7 illustrates two factorization schemes for a transform of size 48 (assuming only codelets of size 2 and 3 have been generated). Even though the two factorizations ultimately use the same factors (codelets), the index mapping or data access pattern is significantly different. Moreover, the number of twiddle multiplications required in the two schedules is also different, i.e., 124 for the schedule given in Figure 6.7(a) and 116 complex twiddle multiplications for the schedule given in Figure 6.7(b). Notice that scheme (a) recurses *outward* by first selecting the largest square factors, while scheme



(a) Inward (b) Outward

Figure 6.7: Factorization schemes for in-place in-order FFT of size 48, that form a palindrome. (a) `(inplace48,2mr(inplace12,2mr3mr2)mr2)`, (b) `(inplace48,(inplace4,2mr2)mr3mr(inplace4,2mr2))`

(b) recurses *inward* by choosing the largest factor in the middle of two small factors. The *outward* recursion generates larger blocks that can be executed independently or in large vectors at unit strides. However, in order to maintain spatial locality, the blocks will need to be executed in a breadth first fashion thereby reducing the potential for temporal locality between successive ranks. On the other hand, the *inward* recursion allows better blocking that maintains both temporal and spatial locality in a cache-oblivious manner. In the UHFFT, the planner uses the *inward* recursion strategy to factorize and execute self-sorting in-place algorithms.

6.3.3 The Four Step FFT

David Bailey's four step algorithm [10] is a popular algorithm for computing FFT requiring external memories. The algorithm follows recursively by dividing the size N problem in two equal (close to $\sqrt{N}$) factors. In the first step, the first rank is computed, followed by the second step, where twiddle factors are multiplied. In the

third step the data is transposed. And finally, the last rank is computed. Although, the four step algorithm is not specifically implemented in the UHFFT, it can be constructed using a combination of `inplace` and `outplace` (*replace*) schedules. To illustrate this, let us consider an example of size 16 FFT, where $r = \sqrt{N} = 4$. Assuming only codelet of size 2 is available, the four step algorithm can be given by following schedule:

```
(inplace16,(outplace4,2mr2)mr(outplace4,2mr2))
```

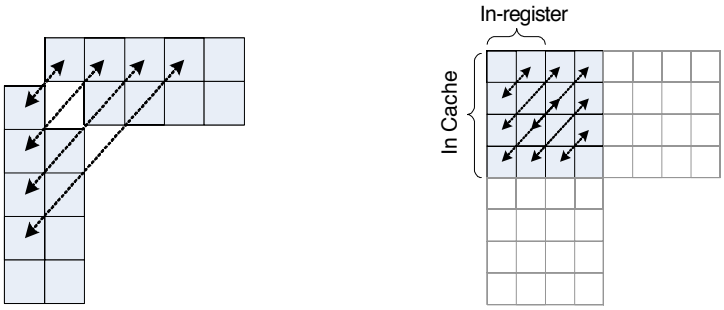
The schedule computes four out-of-place (*replace*) in-order FFTs of size four and multiplies the result with twiddle factors. The `inplace` schedule performs a transpose in the middle followed by another vector execution of an `outplace` schedule of size 4. The schedule requires a buffer to compute both *replace* (`outplace`) schedules, however, extra workspace can be avoided by using the following schedule:

```
(inplace16,(inplace4,2mr2)mr(inplace4,2mr2))
```

Square Transpose

The in-place in-order FFT algorithm described above uses square (sub-matrix) transposes in the first $\lfloor \frac{\log(n)}{2} \rfloor$ ranks to compute ordered FFT. `fftgen` generates specialized codelets that perform the transpose in registers. However, if the required codelet has not been pre-generated or the factor is too large (usually the case in the four step algorithm), then the transpose needs to be performed as a separate step. Depending on the rank where the transpose is performed, the sub-matrix may be accessed at

non-unit stride. We implemented and evaluated three different algorithms to perform square transposition. The naive algorithm transposes strided data in a sub-matrix

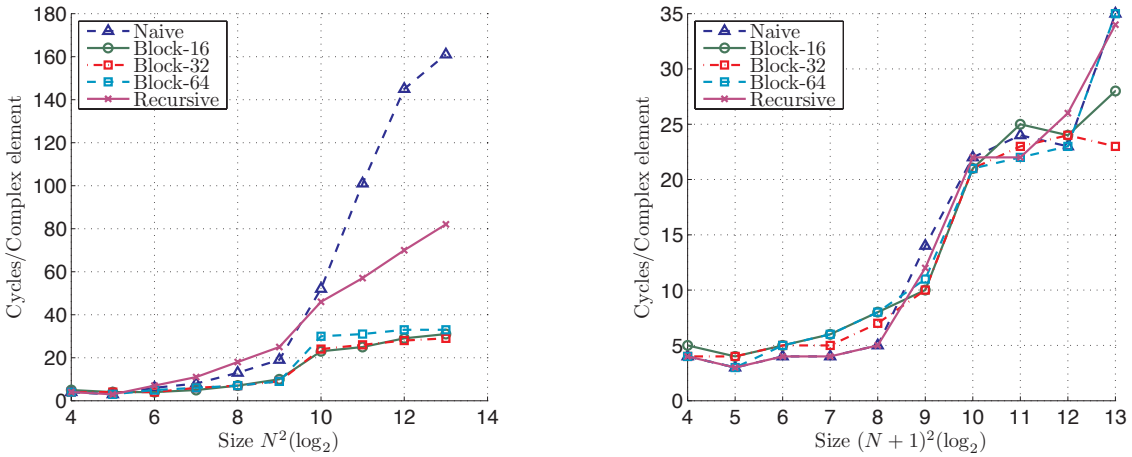


(a) Naive Transpose (b) Block Transpose

Figure 6.8: Iterative square transpose algorithms

by iterating element-wise using two loops as shown in Figure 6.8(a). The naive iterative algorithm can be improved using multilevel blocking as shown in Figure 6.8(b). The block implementation contains four loops, where the outer two loops iterate over blocks and inner two loops iterate inside the blocks. Moreover, internal loops are unrolled by a factor of two (each) to improve register blocking and reduce the number of loop condition checks. A more detailed account of transpose optimization can be found in [38], which uses three level blocking, including optimization for TLB. Both naive and block iterative implementations can be used to transpose sparse square matrices (strided sub-matrices). A third implementation based on Eklundh's algorithm [21] uses recursive implementation to transpose square matrices. The algorithm uses `memcpy` based swaps and requires the data to be contiguous. We include a performance comparison of the three algorithms, although only the first two have been included in the UHFFT library. Figures 6.9 and 6.10 show the performance

of the three algorithms for various square matrix sizes on two architectures. The block algorithm was evaluated for three different block sizes. We observed that the naive iterative algorithm performed slightly better for small matrices while the block algorithm (block size=32) performed best for big matrices. The recursive algorithm did not perform well for large sizes because it touches non-diagonal elements multiple times. Notice that the performance variation for odd size matrices is relatively smaller. This is because the cache conflict misses are fewer since the column stride (size of row) is a non-power of two.

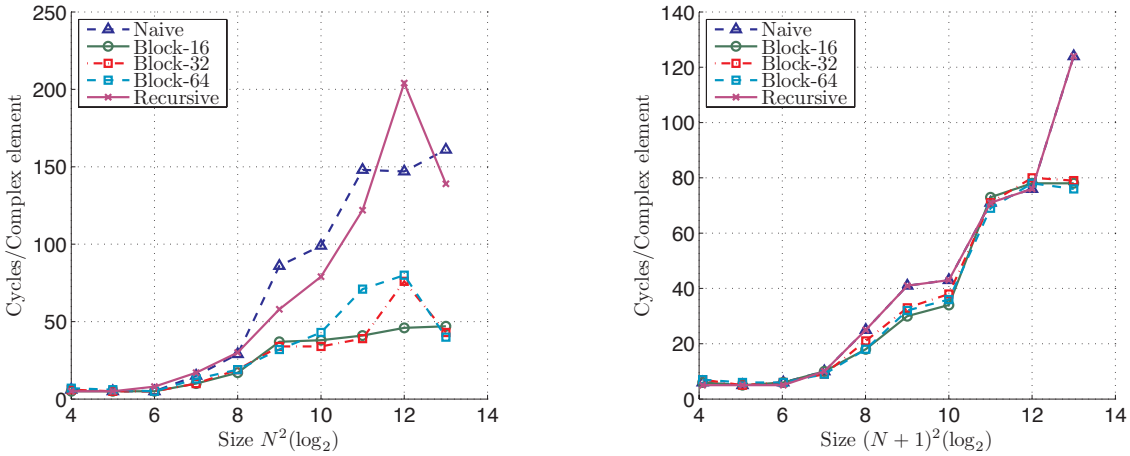


(a) $N \times N$ (b) $(N+1) \times (N+1)$

Figure 6.9: Performance of square matrix transpose on Xeon Clovertown

6.3.4 Prime Factor (PFA) FFT

The mixed-radix algorithm relies on the splitting of size N FFT into two smaller FFTs n_1 and n_2 where $N = n_1 \times n_2$. In this splitting a non-trivial fraction of



(a) $N \times N$ (b) $(N+1) \times (N+1)$

Figure 6.10: Performance of square matrix transpose on Opteron 285

computation involves multiplication with twiddle factors. The prime factor algorithm also relies on a similar splitting but it works on sizes that can be split into two co-prime factors, i.e., $gcd(n_1, n_2) = 1$. The prime factor algorithm avoids the twiddle factor multiplication by mapping one dimensional arrays to two dimensional arrays using complex prime factor mappings (PFM)[37] as discussed in Section 3.1.3. Using special rotated codelets, the PFA algorithm generates the result in-place and in-order. These properties make a PFA schedule a particularly attractive alternative to the mixed-radix schedules described in previous sections. Even though the algorithm is only applicable to a limited set of sizes that have co-prime factors, it can still be useful when embedded inside a larger `outplace` or `inplace` schedule. For example, let us consider a schedule for an in-place in-order FFT of size 60. Assuming that a size 15 codelet has not been generated, $N = 60$ can not be factored in a palindrome of codelet factors. Therefore, we need to use a *replace* schedule of size 15, which

requires a workspace to hold intermediate results:

```
(inplace60,2mr(outplace15,5mr3)mr2)
```

Alternatively, we can use an embedded PFA schedule:

```
(inplace60,2mr(pfa15,5pfa3)mr2)
```

which is likely to perform better since it avoids both workspace and twiddle multiplication. A performance comparison between the mixed-radix (`outplace` schedule) and the PFA algorithms is given in Tables 6.2(a-b).

Table 6.2: Performance (MFlops) comparison of `outplace` and `pfa` schedules

Size	`outplace`	`pfa`
24	3300	3070
63	3310	3290
520	3100	3100
1008	3590	3920
2288	2960	3050
32760	3090	3120
65520	3090	3210
240240	2350	2720
720720	1880	1870

(a) Xeon Clovertown

Size	`outplace`	`pfa`
24	2130	2410
63	1840	2090
520	1790	2230
1008	1980	2430
2288	1500	1820
32760	1160	1370
65520	1320	1490
240240	936	1120
720720	860	1140

(b) Opteron 285

6.3.5 Prime Size (Rader) FFT

The schedule types discussed above depend on the condition that the problem can be factorized in smaller factors recursively to the point where a codelet is available. In order to solve a problem of prime size, a strategy is needed for computing the FFT using sparse factorization. Such a strategy was suggested by Rader that proposed recursive FFT computations for size $N-1$. The `rader` schedule contains an embedded MODULE of size $N-1$ and requires a work-buffer of size N. For further details on the mathematical framework of Rader's algorithm, see Section 3.1.4. In the following, we present the execution steps of the algorithm:

1. Permute the input vector X is using rader's permutation (Eq. 3.15), which is generated during the descriptor setup step.

2. Apply an embedded FFT of size $N-1$ to the permuted vector elements 1 through $N-1$.

3. Compute the skew circulant matrix using the diagonal matrix D in Eq (3.15), and multiply it with the result of step 2. Note that D is precomputed during the descriptor setup step.

4. Apply an embedded FFT of size $N-1$ to the vector elements 1 through $N-1$ of step 3.

5. Un-permute the vector using the array that was used to permute the input vector in step 1.

6.4 Multidimensional Serial FFT Schedules

Multidimensional FFT schedules are very similar to the mixed-radix schedules. In a mixed-radix algorithm, each splitting step turns a one dimensional FFT of size N into a two dimensional $r \times m$ FFT with twiddle factor multiplication sandwiched in the middle. A two-dimensional FFT of size $n_1 \times n_2$, performs n_1 row FFTs of size n_2 followed by n_2 column FFTs of size n_1. In a naive implementation, dimensions are completely transformed in increasing or decreasing order analogous to a breadth first implementation of the mixed-radix algorithm. For a general d-dimensional FFT of size $N = n_1 \times n_2 \times n_3 \times \cdots \times n_d$, a naive implementation will compute $\prod_{i=2}^{d}(n_i)$ row FFTs of size n_1 before computing the next dimensions. On the contrary, a depth first implementation would recursively divide the d-dimensional problem into set of FFTs of fewer dimensions to the point that a single dimension needs to be transformed. Such an implementation is analogous to a depth first mixed-radix schedule, which results in better cache blocking. As discussed later in this section, even better scheduling may be achieved by alternating between multiple dimensions before completing a dimension. The UHFFT supports both depth first schedules for computing a multidimensional FFT; selection of either schedules depends on the DFTI_PLACEMENT parameter of the FFT problem DFTI_DESCRIPTOR, i.e., whether or not memory for a separate output vector is available.

6.4.1 Case 1: In-place Multidimensional FFT Computation

The in-place multidimensional FFT schedule uses a depth first recursive implementation of what is sometimes referred to as the *row-column* method. Assuming an FFT of d dimensions, Algorithm 6 computes the multidimensional FFT in-place in a recursive fashion. Each dimension n_i of the problem $N = n_1 \times n_2 \times n_3 \times \cdots \times n_d$

```
void ndfft_inplace(in,out,is,os,d,N)
begin
   m ← N/n_d;
   mis ← m * is;   mos ← m * os;
   if m=n_d then
      fft(in,out,is,os,n_d,mis,mos);
   else
      for i = 0 to n_d do
         ndfft(in+i*mis,out+i*mos,is,os,d-1,m);
      end
   fft(out,out,mos,mos,m,os,os);
end
```

Algorithm 6: Recursive implementation of multidimensional FFT

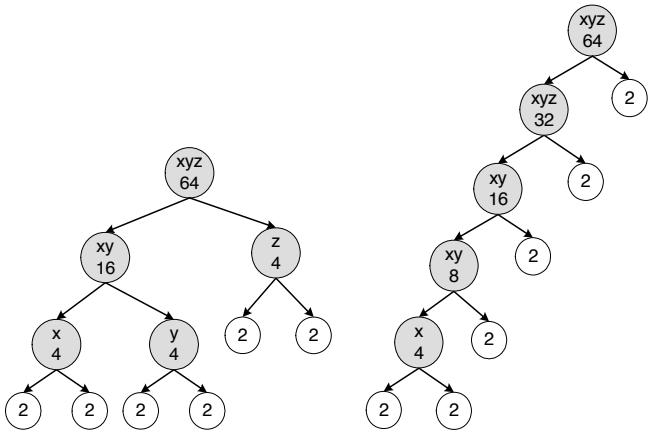
is computed using the one dimensional schedules discussed in the previous sections. However, since the computation needs to be performed in-place, the embedded schedules may use work-buffers as in the case of `outplace` and `rader` schedules. In order to minimize the workspace, the buffer is reused by the one dimensional FFTs that need a work-buffer.

6.4.2 Case 2: Out-of-place Multidimensional Computation

Algorithm 6 is a simple and efficient solution to computing a multidimensional FFT. Not only does it preserve the data locality but the indexing scheme and recursion overhead is minimal as well. Note that it can be used for out-of-place computations by passing different input and output arrays. However, when a separate output array is available, the cache locality can be enhanced by using a slightly different schedule, which alternates between dimensions. In this method, the lower dimension planes or meshes are projected onto the last dimension. The algorithm starts with the last dimension d and breaks the problem using the mixed-radix algorithm. At the leaves of the butterfly (rank 1), the algorithm switches to dimension $d-1$ and continues recursively until dimension $d = 1$. At that point, an FFT of size n_1 is computed followed by rank 1 of butterfly, i.e., for dimension $d = 2$. This algorithm results in a slightly complex indexing scheme in the multidimensional array and also contains some calling overhead due to a deeper call tree, as shown in Figure 6.11. To illustrate the two schedules, let us consider a 3-dimensional FFT of size $N = 64 = 4 \times 4 \times 4$:

```
[(outplace4,2mr2),(outplace4,2mr2),(outplace4,2mr2)]
```

As shown in Figure 6.13, there are 16 rows of size 4 consisting of contiguous data along the x axis. The columns along the y axis consists of data at stride 4 and the vectors along the third dimension consists of data at stride 16. In the first case (in-place), the algorithm transforms four rows followed by four columns. Similarly, the remaining three xy planes are transformed before transforming the third dimension (16 vectors of size 4). Note that the 16 vectors have a vector-stride of 1 (along



(a) In-place (b) Out-of-place

Figure 6.11: Multidimensional codelet call recursion tree

rows). In the second case (out-of-place), the butterfly computation starts along the z dimension by projecting the xy planes onto the z axis. At the leaves of butterfly network, the projected xy planes are transformed before computing the codelets in rank 1 of FFTs along the z axis. In the example under consideration, planes 0 and 2 are transformed followed by 16 codelets of size 2. Similarly, planes 1 and 3 are transformed followed by 16 twiddle codelets of size 2. In the final step, 32 vectors of size 2 (rank 2 butterfly along the z axis) are transformed. Figure 6.12 shows a performance comparison of the two implementations on the Xeon Clovertown CPU. We evaluated three dimensional FFTs of powers-of-two dimensions of the same size, i.e., $N = 2^i \times 2^i \times 2^i$, for $1 \leq i \leq 9$. For both implementations, identical factors and algorithms were selected for each dimension. Notice that for large sizes, the performance of the out-of-place implementation is better than the in-place implementation. As discussed above, this is due to better cache locality of the out-of-place multidimensional FFT implementation.

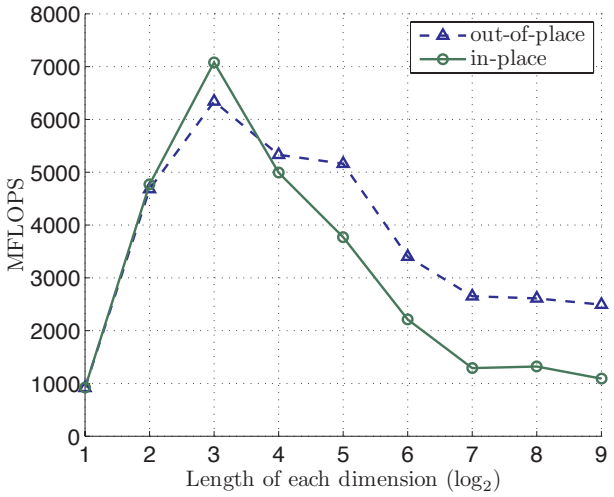


Figure 6.12: Performance comparison of 3D FFTs using the two implementations on Xeon Clovertown

6.5 Parallel FFT Schedules

The main performance issue of serial (single node) FFTs has largely been the effective memory bandwidth and vectorization; achieving super-linear speedup on SMPs is possible because of larger effective cache size. Parallel algorithms for FFT can be broadly divided into two categories, i.e., those that perform explicit reordering of data using transposes [1, 10] and those that do not perform any movement of data. Computing an FFT on distributed data is not possible without movement of data. But, parallel FFT can be computed without remapping of data on architectures with shared address space. In general, explicit scheduling may perform better depending on the cost of data communication among processors. The performance of multithreaded execution depends on that of the serial code. Once an FFT problem is divided among threads, each of them executes part of the serial schedule. In

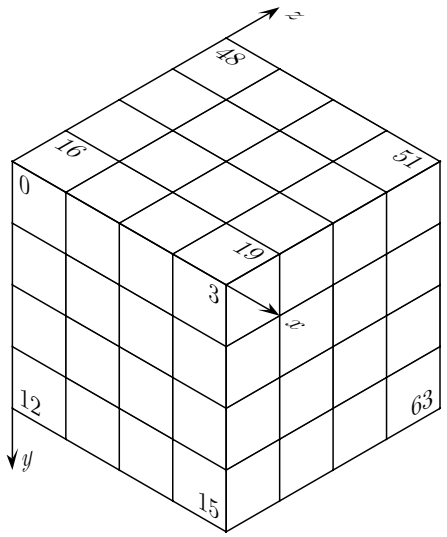


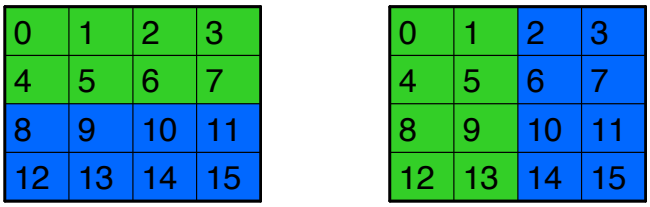
Figure 6.13: Three dimensional array of size $4 \times 4 \times 4$

the mixed-radix formulation of the FFT (Eq. 3.5), a problem of size N is divided into two factors r and m, where $N = r \times m$. It allows r row FFTs of size m, i.e, $(I_r \otimes W_m)$ to be executed independently followed by "twiddle" multiplication and m independent column FFTs of size r, i.e., $(W_r \otimes I_m)$. Both, the permutation step and the "twiddle" multiplication step can be fused inside the first FFT subproblem, while still maintaining the data parallelism. Let us consider the example shown in Figure 6.2. Notice that the problem can be easily distributed between two processors by assigning upper and lower subtrees that compute the row FFTs of size 4. Similarly, the last rank can be parallelized by distributing four column FFTs of size 2 among available processors. Before computing the last rank and after finishing the second to last rank, threads need to be synchronized at a rendezvous point called *barrier*. Synchronization of threads adds to the overhead of multithreaded execution and causes major performance degradation if proper load-balancing is not employed.

In general, even distribution of work is possible when P divides $\sqrt{N}$, i.e., $(P|\sqrt{N})$.

6.5.1 Row/Column Blocking

The communication cost on shared-memory multiprocessor systems with uniform memory access (UMA) does not play a major role in the overall performance. However, different computation distributions have an impact on the performance of memory systems on some architectures. Figure 6.14 shows the block execution distribu-



(a) Row FFT (b) Column FFT

Figure 6.14: Distribution of parallel FFT

tion of two ranks of a parallel schedule for an out-of-place in-order FFT of size 16, i.e., `(mrp2,(outplace16,4mr4))16`. Notice that execution of the rows and columns can alternatively be distributed in a block cyclic manner. However, such a distribution of columns is likely to result in poorer performance because of *false sharing* of boundary elements among processors. This was verified through an experiment on a four processor SMP machine. As shown in Figure 6.15, larger block distribution results in significant performance improvement over smaller blocks. This is mainly due to the cache coherence conflicts and false sharing [43], which are caused by the elements, at the column boundaries, falling in same cache line but different

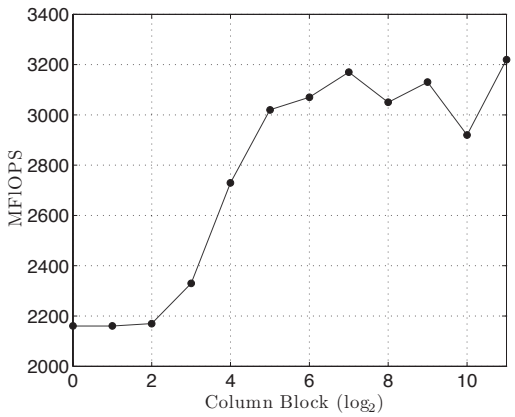


Figure 6.15: Performance impact due to data distribution on Itanium 2 (Quad)

processors. As a rule of thumb, choosing maximum blocks of columns works best. In general, the blocks should be at least $size_{cacheline}/size_{datapoint}$ elements apart.

6.5.2 Multithreading Model

Most compilers support native multithreaded programming APIs. The Posix thread (PThread) library routines are flexible but offer limited portability across different architectures. OpenMP provide a portable and scalable interface for developing multithreaded applications. Two types of multithreaded programming models are commonly used; the fork/join model and the thread pooling model. Posix thread is an example of fork/join threading model. Nevertheless, thread pooling can be built on top of fork/join threading. Most OpenMP implementations use thread pools to avoid the overhead of creating and destroying threads after each parallel region. These threads exist for the duration of the program execution. Implementation of parallel FFT using OpenMP is relatively straightforward given an efficient parallel

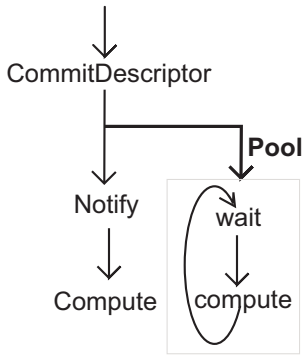


Figure 6.16: Thread pool model

plan. The two loops around the row and column FFTs are decorated with `omp` directives and they are distributed in contiguous chunks (blocks). In our PThreads implementation, we use a thread pooling technique to avoid the overhead of creating threads. The pool of peer threads, as shown in Figure 6.16, is created when the `dfti_descriptor` is submitted and after the best plan is selected. The pool is destroyed when the descriptor is freed. To reduce the cost of synchronization, we implemented a low latency barrier (with sense-reversal) using an atomic decrement instruction as listed in the code segment (Algorithm 7). In addition to that, we used *busy wait* when the plan was perfectly load balanced. Contrary to waiting on events, this technique avoids the overhead of system calls and thread scheduling through the operating system. In Figure 6.17, we give a performance comparison of a multithreaded implementation using pooling and a customized barrier with an implementation that used a fork/join model and native synchronization primitives. In general, we observed that thread pooling was a major factor in the performance improvement for relatively small sizes. But, for larger sizes, the *busy wait* resulted in slightly lower performance. In another experiment, we compared the performance of

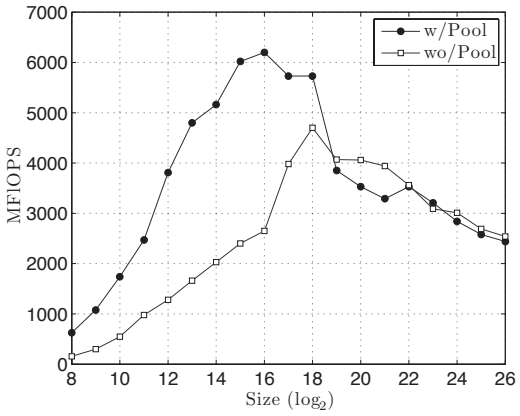


Figure 6.17: Performance improvement due to pooling and busy wait techniques on Xeon5100 (four cores), using scalar codelets

the two parallel FFT implementations, i.e., OpenMP and PThreads on a quad SMP Itanium 2 machine. Our PThreads implementation uses thread pooling and busy wait techniques, which are common in most OpenMP flavors. In both implementations, we favor allocating the execution of contiguous chunks (blocks) of rows and columns to each core/processor. Considering all these similarities, it is not surprising that the performance of both implementations is almost identical as shown in Figure 6.18. Since the UHFFT run-time scheduler performs the scheduling (load distribution) in the precomputation stage, the role of an OpenMP compiler is generally limited to providing portable thread management and a synchronization mechanism rather than providing efficient load-balanced loop distribution. In Figure 6.18, notice that for FFTs of small sizes, serial execution performs better than multithreaded execution.

In the discussion above, we focused on the parallelization of out-of-place in-order FFT computations. The implementation of multidimensional schedules is fundamentally the same, whereby the execution distribution takes place along the last (dth)

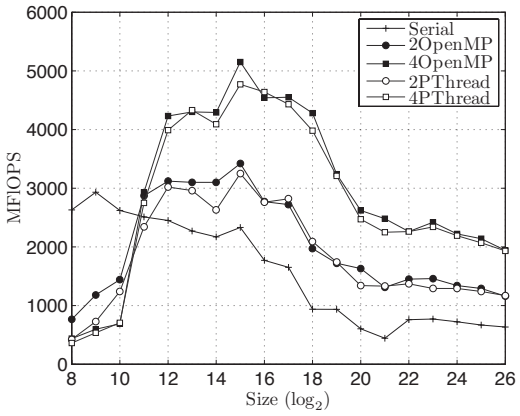


Figure 6.18: Performance Comparison of two multithreaded implementations on Itanium 2 quad SMP

dimension. This ensures that there is only a single point of synchronization. In-place in-order FFT schedules, however, require two points of synchronization, i.e., after the 1st rank and after the $(\log(n) - 1)$th rank.

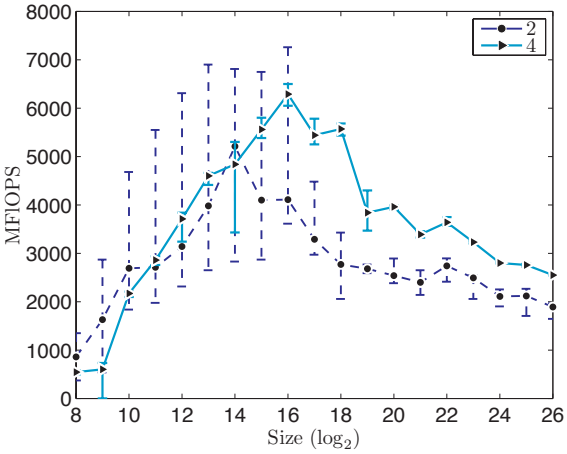
6.5.3 Thread Affinity

Both the Xeon Woodcrest and the Opteron 275 architectures have identical number of cores but their cache configuration is quite different. The two cores on the Xeon have a shared L2 cache while the cores on the Opteron have private caches similar to a conventional SMP machine. Although the shared cache configuration has its benefits in certain scenarios, it can pose some scheduling problems. On the Xeon, we observed inconsistent performance for small sizes (that fit in cache) when only two threads were spawned. This phenomenon is shown in Figure 6.19; notice the extent of performance variation on the Xeon compared to the Opteron. Although

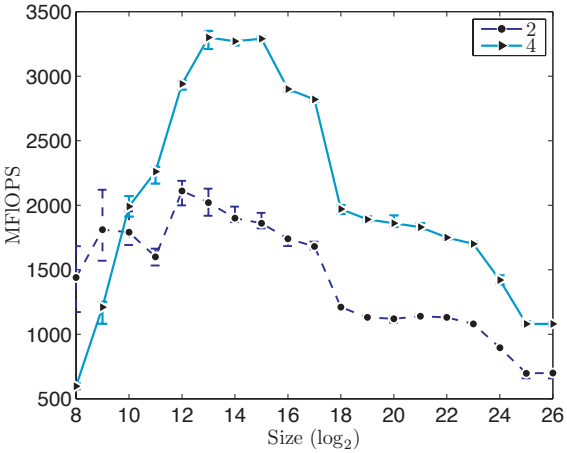
```
void barrier()
begin
   local_sense ← ¬local_sense;
   AtomicDecrement(global_count);
   if global_count=0 then
      global_count ← num_threads;
      global_sense ← local_sense;
   else
      while global_sense!=local_sense do
         if count > MAXCOUNT then count←0; SLEEP;
         count++;
      end
end
```

Algorithm 7: Customized barrier implementation using *busy wait*

the scheduler in the Linux kernel 2.6 tries to schedule tasks on different sockets when the system is lightly loaded, there is no guarantee that the scheduler will get it right the first time. The performance drops when the two threads are scheduled on different cores on the same socket, which may result in data evictions due to conflicts in the shared cache. This problem can be resolved by setting the thread affinity to CPUs such that threads are placed properly. In the UHFFT, the threads are explicitly bound to cores on different sockets if the number of threads spawned is smaller than the available cores on a node.



(a) Xeon (shared L2) using scalar codelets



(b) Opteron (private L2) using scalar codelets

Figure 6.19: Performance variation on two multicores with different cache configurations (Shared L2 Cache vs Separate L2 Cache)

Chapter 7

Run-time Search

Run-time search and analysis is one of the main ingredients of the two-layered adaptation methodology. An FFT is identified by the DFTi descriptor, which describes various characteristics of the problem including size, precision, input and output data type and number of threads. Given the parameters of a problem, the initialization routine attempts to select a plan that minimizes the execution time on the given architecture. The best schedule, thus selected, may be used repeatedly to compute the transforms of the same size. In general, the search scheme can be driven by a model or it can be empirical[65]. Empirical search generally does a better job at finding the best performing code at an additional cost in terms of the search time. Most auto-tuning libraries such as ATLAS[18], FFTW[25] and SPIRAL[44] favor an empirical search scheme, giving preference to FFT execution performance over one time plan search cost. For FFT, empirical search can take orders of magnitude more time than the actual execution of code due to an exponential space of algorithm and

factorization trees. But, the cost of empirical search schemes can be minimized by pruning the search space and reusing the cached empirical performance models. In the UHFFT, we have implemented multiple search methods with varying costs, which allow users to select the search scheme according to their needs. Our results indicate that expensive run-time empirical search can be avoided by generating domain specific models (offline) without compromising the quality of schedules generated.

7.1 Search Space

The space of FFT schedules consists of algorithms and their factorizations. The UHFFT search engine (planner) selects the best schedule in two stages. In the first stage, a pruned library of codelets and sub-schedules is constructed using heuristics. For example, given an FFT of size $N = n_1 \times n_2 \times \cdots \times n_i \times p_1 \times p_2 \times \cdots \times p_j$ that contains some co-prime factors, $p_1, p_2, \ldots, p_j$, we can pre-select a `pfa` sub-schedule of size $S_1 = p_1 \times p_2 \times \cdots \times p_j$. This results in a pruned search space consisting of fewer factors, i.e., $N = n_1 \times n_2 \times \cdots \times n_i \times S_1$. Similarly, the sub-schedules for large prime factors are also constructed in this stage. This stage produces a trimmed library of modules (codelets and sub-schedules), which we refer to as the *ad hoc* library. In the second stage, the best mixed-radix factorization is searched from the set of modules (in the *ad hoc* library) for the given FFT. The search space for mixed-radix factorizations of an FFT of size 16 is shown in Figure 7.1. Each of the eight branches (sequences) of the tree represents a possible factorization for computing the size 16 FFT. Note that the size of the search space depends on the

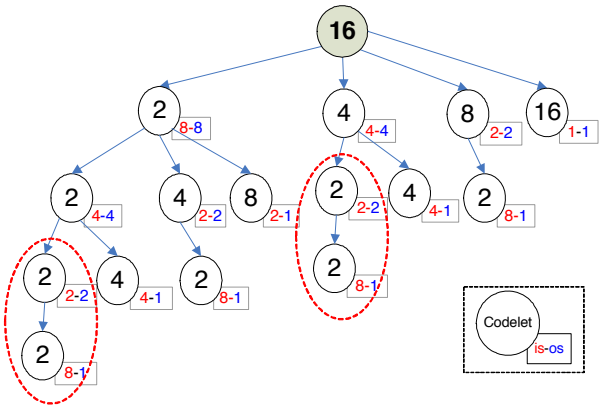


Figure 7.1: Search space for mixed-radix FFT of size 16

number of codelets present in the library. For a power of two size ($N = 2^i$) FFT, the search space can be pruned by getting rid of less efficient codelets. In Figure 7.2, we show the performance of 248 mixed-radix factorizations for a size 512 FFT. The left end of the plot starts with the smallest codelet, i.e., size 2 and recursively factorizes the subtrees in a pattern shown in Figure 7.1. Notice that the best factorizations are clustered at the right end of the graph, which tend to use larger power-of-two size codelets. Performance of the best factorization is almost an order of magnitude better than the worst performing schedule. In the UHFFT, we have implemented three search schemes of varying costs, which are discussed in the next sections.

7.2 Empirical Search

In a naive empirical search scheme, the best schedule is selected after executing all possible factorizations for the given size. The schedule thus selected, will have the smallest execution time out of all the possible factorizations. But, the time

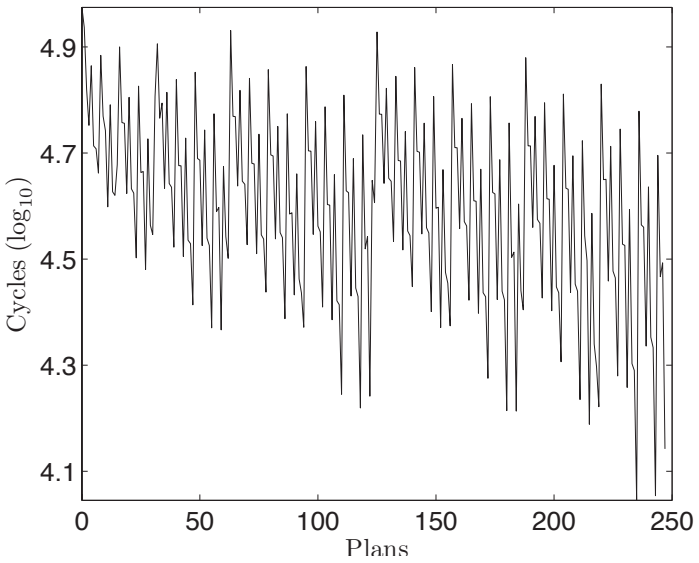


Figure 7.2: Performance of 248 different factorizations for mixed-radix FFT of size 512

required to exhaustively search an exponential space of factorizations may be quite large. Furthermore, the exhaustive search scheme is also costly in terms of the workspace requirement. To avoid the drawbacks of an exhaustive search scheme, we use the *memoization* technique. Using dynamic programming, the tree of mixed-radix factorizations is evaluated in a bottom up fashion.

7.2.1 Context Sensitive Empirical Search

The first approach called the *context sensitive* empirical search (DFTI_HIGH), empirically evaluates the sub-schedules in a bottom up fashion using dynamic programming. We take advantage of the recursive structure of the FFT to avoid re-evaluating common subsequences (branches with identical strides) in the factorization tree. As

shown in Figure 7.1, some factorizations contain identical subsequences, i.e., same codelets are used with identical I/O strides. We minimize the number of empirical evaluations by memoizing the cost of each subsequence in a bottom up fashion. To formulate the problem in a recursion, we consider an FFT of size N, which consists of small factors, i.e, $N = n_1 \times n_2 \times n_3 \times \cdots \times n_j$. Assume that the *ad-hoc library* contains modules of sizes, $r_1, r_2, \ldots, r_k$, where $r_k = \prod_{q \in \{1,2,\ldots,j\}} n_q$. When $j = 1$ i.e., $N = n_1$, the best factorization can be found by selecting the module that yields the minimum execution time $C_{min}(N)$ for computing the FFT. However, when $j > 1$, we can use the optimal substructure of the problem to characterize it in a recursion, given by:

$$C_{min}(N) = \begin{cases} 0 & if\ N = 1 \\ min\left(C_{min}(N/r_1) + C(r_1), \ldots, C_{min}(N/r_i) + C(r_i)\right) & \forall\ r_i | N \end{cases}$$

The cost of a module or codelet $C(r_i)$ is represented by its execution time with appropriate input and output strides. The costs of evaluated subsequences for specific strides are stored in a lookup table. Each time a subsequence is not found in the table, the sub-schedule must be executed to compute its cost. The sub-schedule is executed completely to get an accurate measure of the cost; it ensures that the state of the cache (context) is representative of the actual execution of the schedule. A pseudo code of the search scheme is given in Algorithm 8.

```
void DPSearch(N,is,os)

begin
    if S ← TableLookup(N, is, os) then return S;

    S ← ∅; mincost ← MAXINT;

    foreach r_i ∈ {module library} do
        if r_i = N then S ← r_i;

        else S ← DPSearch(N/r_i, is, os) + r_i;

        cost ←Evaluate(S);

        if cost=mincost then
            TableInsert(S, N, is, os, cost);

            mincost ← cost;

        end

    end

    return TableLookup(N, is, os);
end
```

Algorithm 8: Search for the best FFT schedule using dynamic programming

7.2.2 Context Free Empirical Search

One of the drawbacks of the *context sensitive* search scheme is that it requires empirical evaluation of full sub-schedules to evaluate the total cost. The *context free* search scheme (DFTI_MEDIUM) is a *hybrid* of the empirical and estimation techniques. In this scheme, the cumulative performance of a schedule is estimated by empirically evaluating the building blocks (modules) independently, hence the name *context free*. In this scheme, the cost of a subsequence is estimated by empirically evaluating only the codelet that is encountered in a bottom up traversal. The expected execution

time is derived from the type of codelets (twiddle/non-twiddle) being used in the schedule, the number of calls to each codelet, and the codelet performance table generated at run-time for various input and output strides. This approach is much faster than the first search strategy. But, the quality of the execution schedule may suffer since it relies on the assumption that codelet timings can be used to predict the execution time of complete schedule.

7.3 Model Driven Search

The *context free* empirical search uses an estimation technique to calculate the performance of a subsequence from the execution time of codelets, which makes it significantly faster. However, it is still expensive since it enumerates all the possible factorizations and relies on run-time empirical evaluation. The run-time empirical evaluation can be easily replaced with a database of codelets' performance. Instead of generating the database at installation time, we use a lazy protocol; the models are generated if and when needed and reused repeatedly. The models are cached in a persistent file, which stores the list of strides and the sequence of codelets in descending order of performance for each stride. The model driven search scheme (DFTI_LOW) employs a greedy algorithm (shown in 9) to select the best codelet (factor) in each rank, starting at rank 1 ($stride = 1$). To keep the database compact, we only store strides that are powers-of-two. Furthermore, the codelets use identical stride for input and output vectors since they are executed in-place, i.e., the input and output vectors point to the same location. In order to estimate the performance

of a codelet of size r for a non-power of two $stride$, such that $2^i < stride < 2^{i+1}$, we use the performance data represented by $stride = 2^i$.

```
void GreedySearch(N,is,os)
begin
    m ← N; stride = os;
    while m > 1 do
        logs ← log2(stride) + ϵ;
        for i ← 0 to NUM_CODELETS do
            r ← perfmodel [logs] [i];
            if m%r = 0 then break;
        end
        InsertInSchedule(GetModule(r));
        stride ← stride × r;
        m ← m/r;
    end
end
```

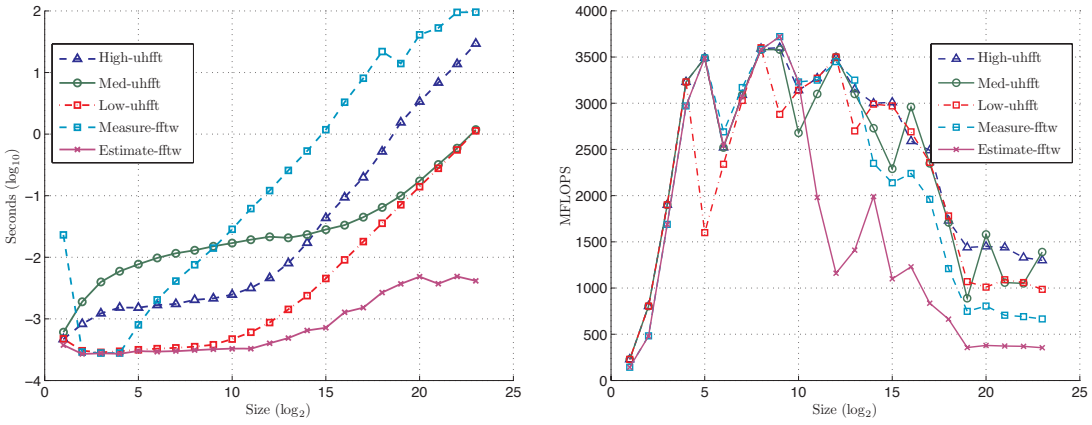
Algorithm 9: Model driven search using greedy algorithm

7.4 Performance Comparison of Search Methods

One of the main goals of this dissertation is to minimize the cost of search through development of efficient performance models and search schemes. Both the UHFFT and FFTW implement multiple run-time search schemes to select the best schedule (plan) of execution for a given DFT problem. In our analysis, we include the

three search schemes implemented in the UHFFT, i.e., DFTI_HIGH, DFTI_MEDIUM and DFTI_LOW. For the FFTW library, we have selected two search schemes, i.e., FFTW_MEASURE and FFTW_ESTIMATE. We do not include the MKL's search schemes in our analysis because the recent versions of MKL do not perform any run-time search in the `dfti_descriptor` initialization step. We use two metrics to evaluate the efficiency of a search scheme for selecting the best schedule (plan) of execution.

The cost of a search scheme is given by the amount of time (seconds) it takes to initialize the FFT computation schedule. In the DFTI interface, the initialization is performed by `DftiCommitDescriptor` function. The second metric used to rank the search schemes is the accuracy in finding the best schedule. We compare five search schemes implemented in the most recent versions of the UHFFT and FFTW for three sets of FFTs, i.e, powers-of-two, non-powers-of-two and prime sizes, given in Table 7.1.



(a) Cost of search

(b) Accuracy of search

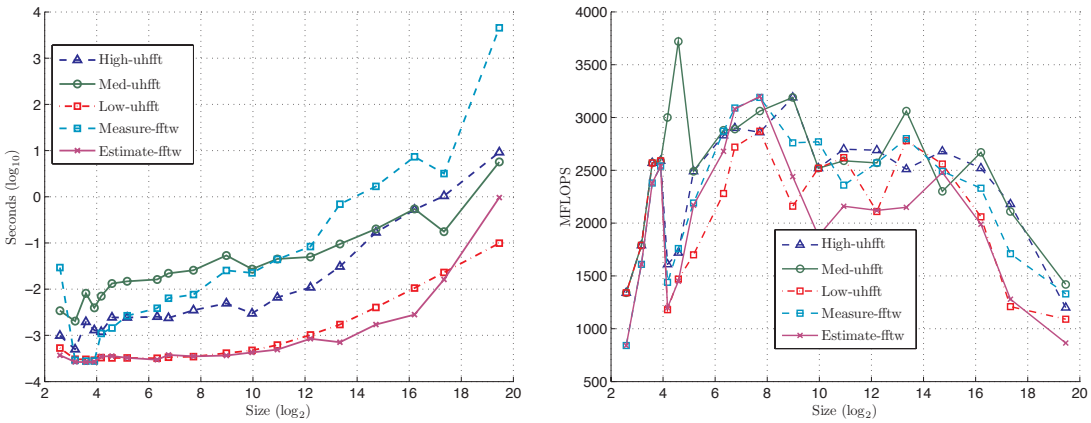
Figure 7.3: Comparison of various search schemes for powers-of-two size FFTs executed on Itanium 2

7.4.1 Powers-of-two Size FFTs

In Figure 7.3, we show a comparison of the search schemes for out-of-place complex to complex FFTs of powers-of-two sizes. Graph 7.3(a) compares the search costs for each FFT size. Graph 7.3(b) compares the performance of the best schedule generated by each search scheme. Notice that the FFTW's empirical search scheme (FFTW_MEASURE) is an order of magnitude slower than the UHFFT's most expensive search scheme (DFTI_HIGH). In general, if the schedule is not likely to be used many times (e.g., in one-dimensional FFTs), it is better to use low cost estimation schemes. The FFTW_ESTIMATE method is faster than the DFTI_LOW search scheme but also generates significantly slower schedules. In terms of effectiveness (accuracy) of scheduling, the three search schemes implemented in the UHFFT generated schedules that were very close to each other in performance, indicating higher effectiveness (accuracy) of our model driven and hybrid search schemes.

7.4.2 Non-powers-of-two Size FFTs

In Figure 7.4, we compare the search schemes for non-powers-of-two size FFTs. For non-powers-of-two sizes, the UHFFT employs heuristics so that the faster PFA algorithm may be used for co-prime factors. This results in a pruned search space since the co-prime and large prime factors are replaced by their schedules using heuristics. Therefore, the search schemes in the UHFFT are significantly faster than the FFTW's empirical search scheme. The current implementation of the DFTI_LOW



(a) Cost of search

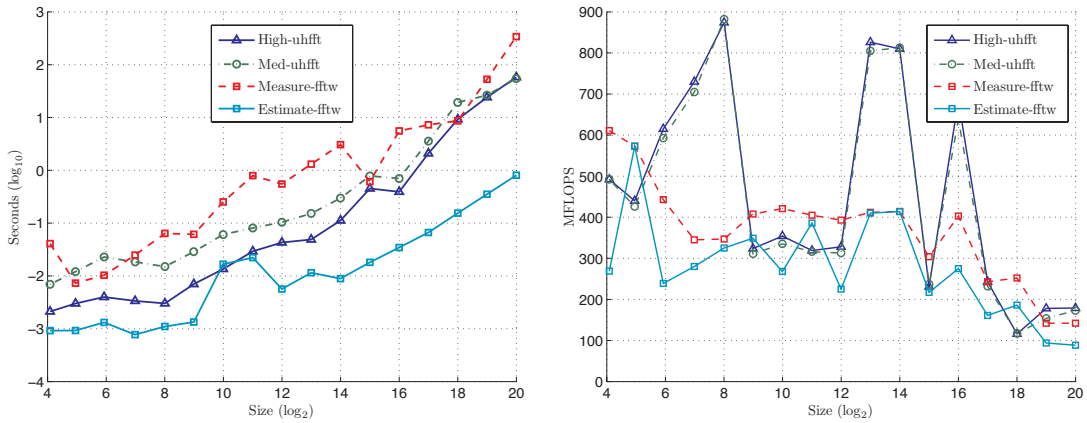
(b) Accuracy of search

Figure 7.4: Comparison of various search schemes for non-powers-of-two size FFTs executed on Itanium 2

search scheme does not use PFA schedules, which may result in generation of sub-optimal schedules. Notice that the cost of search does not increase exponentially as it did for powers-of-two sizes. As discussed previously, the cost of the empirical search depends on the number of factors it needs to evaluate. When the number of factorizations is limited, as is the case for many non-powers-of-two sizes, the search space is dramatically reduced. Moreover, as the number of factors increase, so does the size of the pre-computed twiddle table. The cost of the twiddle table set up forms a major component in the search methods implemented in the UHFFT.

7.4.3 Large Prime Size FFTs

In the UHFFT, large prime size FFTs are computed using Rader's algorithm [45], which uses a couple of $N-1$ size transforms (convolution). The cost of search in



(a) Cost of search

(b) Accuracy of search

Figure 7.5: Comparison of various search schemes for large prime size FFTs executed on Itanium 2

prime size FFTs is the amount of time spent in searching for the best schedule for an $N-1$ size FFT and in initialization of data structures for a `rader` schedule. In Figure 7.5, we show a comparison of cost and accuracy of search schemes for various prime sizes. Notice that both the search schemes implemented in the UHFFT take almost the same amount of time and produce identical schedules. For certain sizes that contain large prime factors, recursive application of Rader's algorithm results in slower schedules in the UHFFT; e.g., prime sizes 509, 1021, 2053 and 4093 (between 2^8 and 2^{12}) use large prime factors. The FFTW avoids recursive execution of Rader by implementing a zero-padded variant.

7.5 Performance Comparison of FFT Libraries

In this section, we compare the performance of FFTs of various sizes and dimensions on multiple architectures using the UHFFT and FFTW libraries. On Intel and AMD architectures, we also compare the FFT performance with Intel's Math Kernel Library (MKL). We evaluate the FFT libraries on the basis of their performance in four categories:

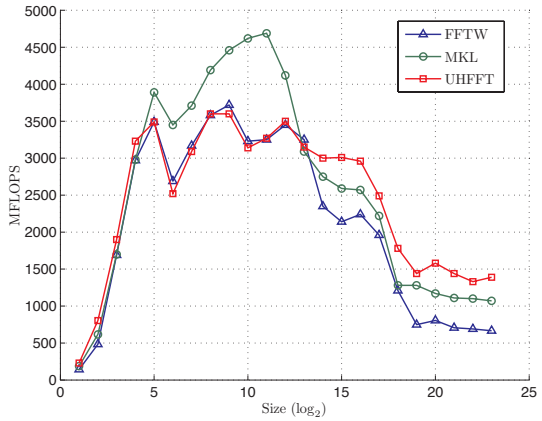
1. Efficiency of micro-kernels (codelets)
2. Performance of in-cache sizes
3. Performance of out-of-cache sizes
4. Speedup gain due to multithreaded execution

7.5.1 One-dimensional Data

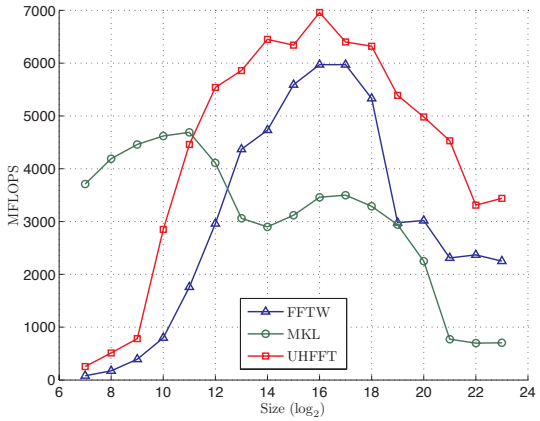
The powers-of-two size FFTs are frequently used in comparisons of different FFT implementations. We have selected a range of powers-of-two sizes ($2^1 \leq N \leq 2^{23}$) to compare out-of-place and in-place complex to complex FFT performance on Itanium 2, Xeon Clovertown and Opteron 285. For each architecture, we executed the best schedule in serial (on single core) and using all available processors.

Out-of-place In-order FFTs

Figures 7.6, 7.7 and 7.8, show the performance of out-of-place execution using the UHFFT, the FFTW and the MKL libraries. It is not surprising that Intel's vendor library (MKL) performs best on the Itanium and Xeon for small and medium size FFTs. Observe that for large (out-of-cache) sizes, the UHFFT performs better than the other two libraries except on Opteron, which has a small cache size with very low degree of associativity $d = 2$. On architectures that have a small degree of associativity d, the UHFFT search engine is likely to choose small size codelets ($n \leq d$) for higher (large stride) ranks. Recall from the analysis of performance models in Chapter 5, that size 2 codelets are not as efficient as the larger codelets. The FFTW employs transposes to reorder the data access pattern to avoid large strides on the Opteron. Both the UHFFT and FFTW generate SIMD codelets on Xeon, but, unlike FFTW, the UHFFT does not perform extensive code optimizations on SIMD codelets. That is why the performance of the UHFFT for in-cache and codelet size FFTs is lower than the other two libraries. However, the UHFFT performs better for very large sizes because it manages the cache better, especially in dealing with the twiddle multipliers as discussed in previous chapters. As far as multithreaded performance is concerned, the UHFFT consistently out-performs the other two libraries. This is owing to the design of the multithreaded run-time support of the UHFFT, which uses customized low cost synchronization mechanism and thread pooling as discussed in Section 6.5.2. The UHFFT search engine uses estimation techniques to distribute the tasks equally among the threads.

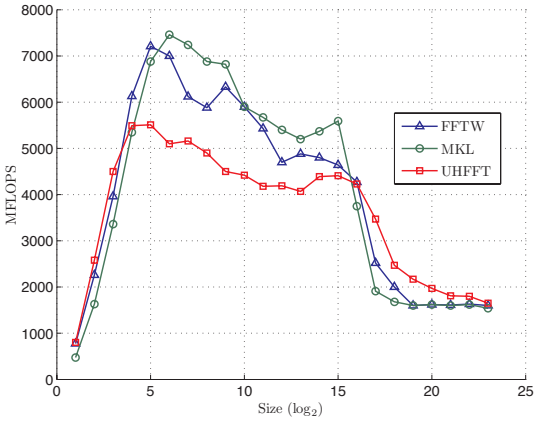


(a) Serial

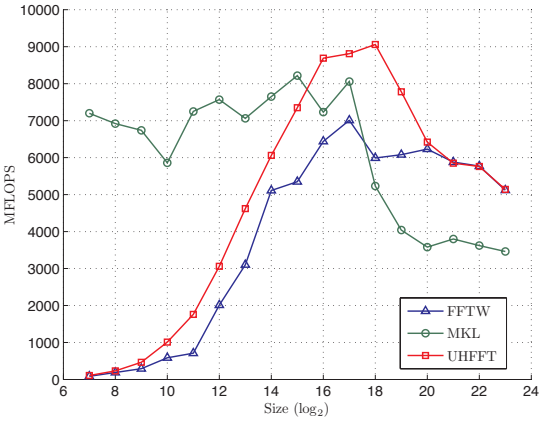


(b) Four threads

Figure 7.6: Itanium 2 - performance comparison of out-of-place complex to complex FFTs of powers-of-two sizes

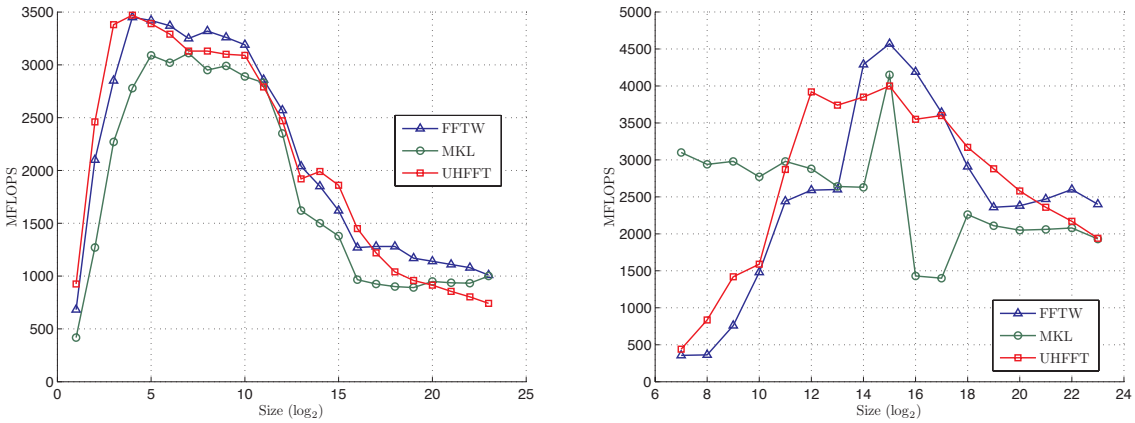


(a) Serial



(b) Eight threads

Figure 7.7: Xeon Clovertown - performance comparison of out-of-place complex to complex FFTs of powers-of-two sizes



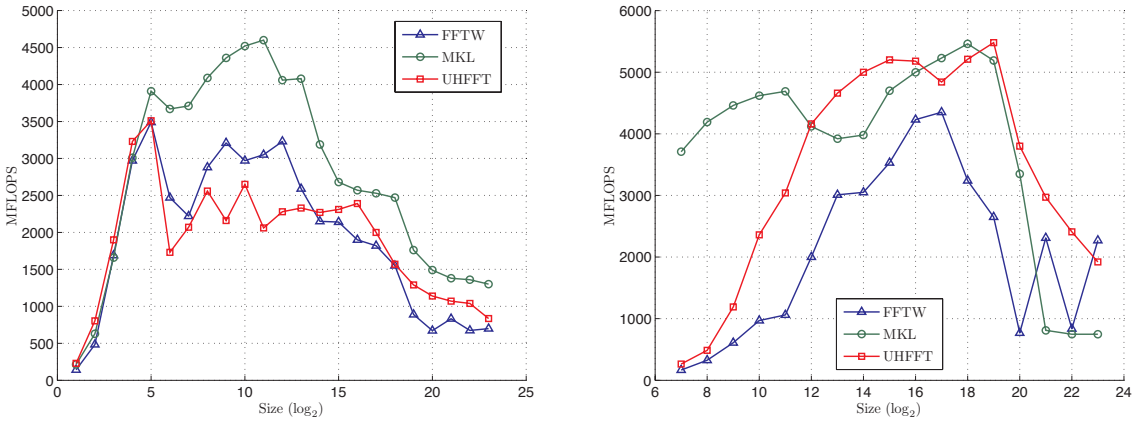
(a) Serial

(b) Four Threads

Figure 7.8: Opteron 285 - performance comparison of out-of-place complex to complex FFTs of powers-of-two sizes

In-place In-order FFTs

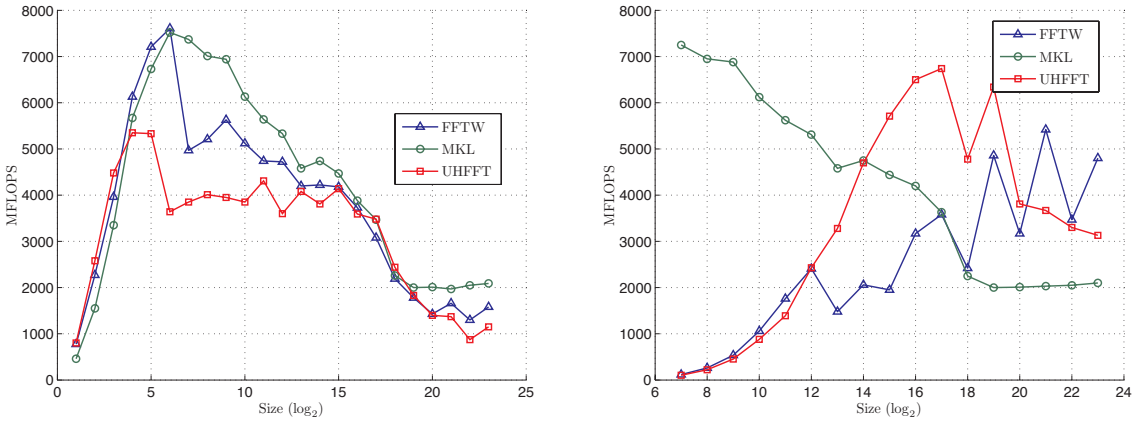
In-place FFTs replace the input vector with the results of the transform. The in-place FFT can be computed without using any temporary buffers by employing an algorithm that performs partial transposes. In order to achieve that, the size N must be factorized in a palindrome of factors. The UHFFT does not use any buffers in computing the in-place in-order FFT. On the contrary, the FFTW allocates and uses additional buffers to achieve better performance, especially for small sizes. As shown in Figures 7.9(a), 7.10(a) and 7.11(a), the FFTW performs better than UHFFT for medium size in-place FFTs.



(a) Serial

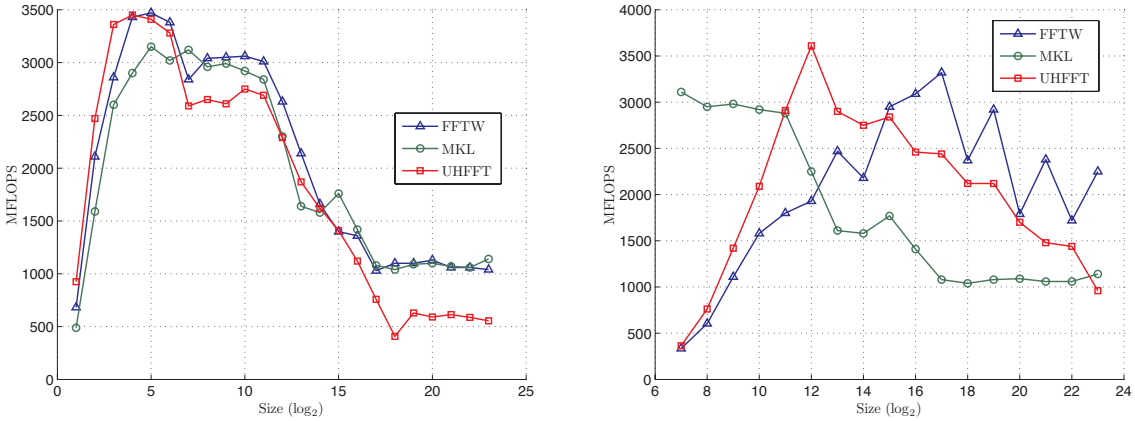
(b) Four threads

Figure 7.9: Itanium 2 - performance comparison of in-place complex to complex FFTs of powers-of-two sizes



(a) Serial (b) Eight threads

Figure 7.10: Xeon Clovertown - performance comparison of in-place complex to complex FFTs of powers-of-two sizes

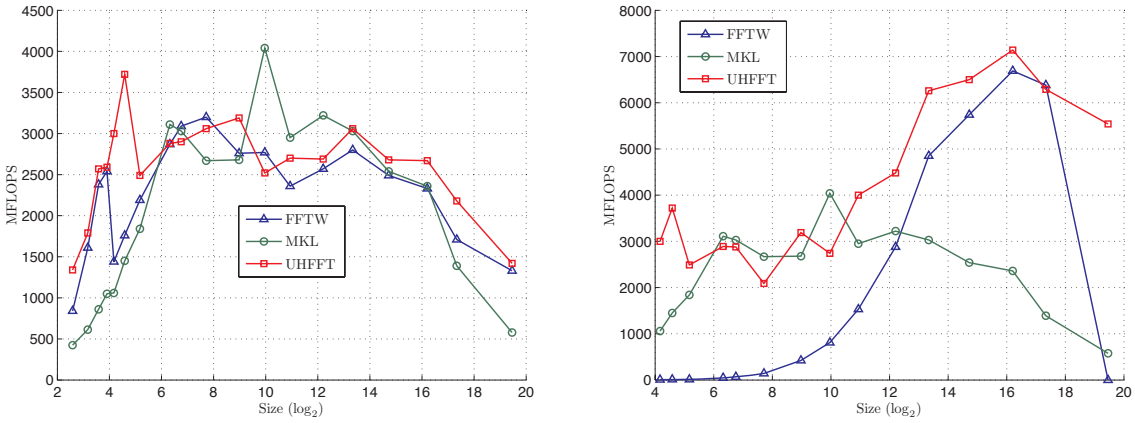


(a) Serial (b) Four threads

Figure 7.11: Opteron 285 - performance comparison of in-place complex to complex FFTs of powers-of-two sizes

Non-powers-of-two Size FFTs

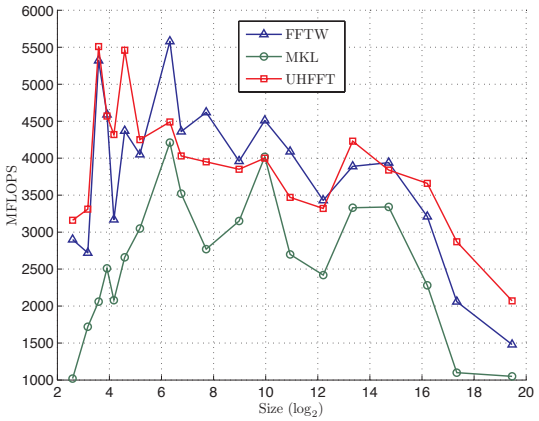
The UHFFT implements PFA driver routines for computing the transforms where the size N can be factorized in co-prime factors. On the other hand, the FFTW is best at handling sizes of the form $2^a \times 3^b \times 5^c \times 7^d \times 11^e \times 13^f$. In fact, the sizes included in our analysis are of the form that is suited to the FFTW. Nevertheless, as shown in Figures 7.12(a), 7.13(a) and 7.14(a), the performance of UHFFT is very competitive with that of FFTW. As far as multithreaded performance is concerned, the UHFFT performs better than the other two libraries in most cases.



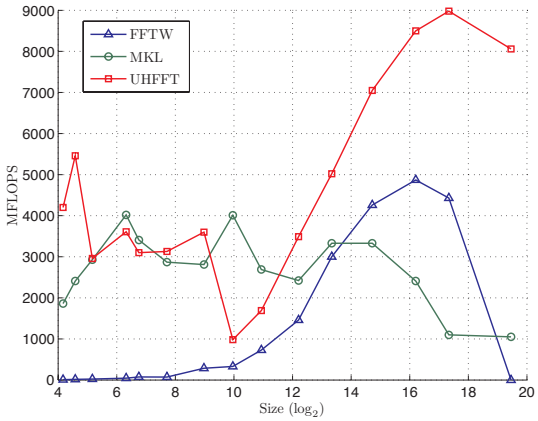
(a) Serial

(b) Four threads

Figure 7.12: Itanium 2 - performance comparison of out-of-place complex to complex FFTs of non-powers-of-two sizes

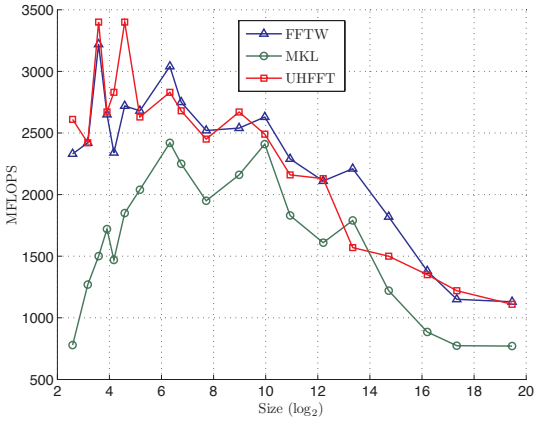


(a) Serial

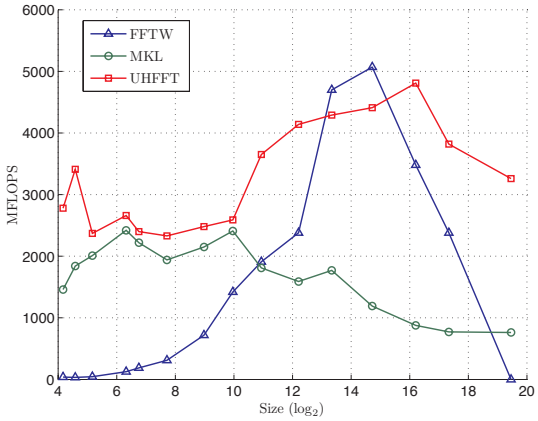


(b) Eight threads

Figure 7.13: Xeon Clovertown - performance comparison of out-of-place complex to complex FFTs of non-powers-of-two sizes



(a) Serial



(b) Four threads

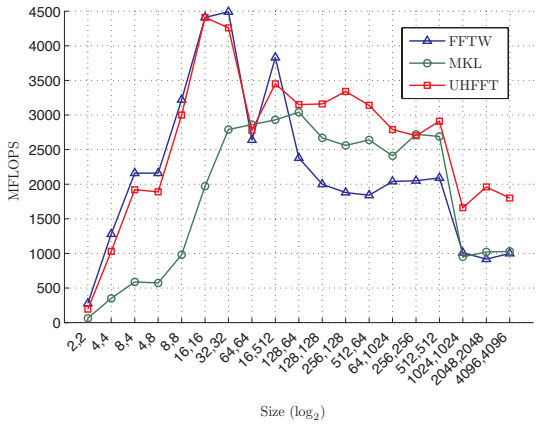
Figure 7.14: Opteron 285 - performance comparison of out-of-place complex to complex FFTs of non-powers-of-two sizes

7.5.2 Multidimensional Data

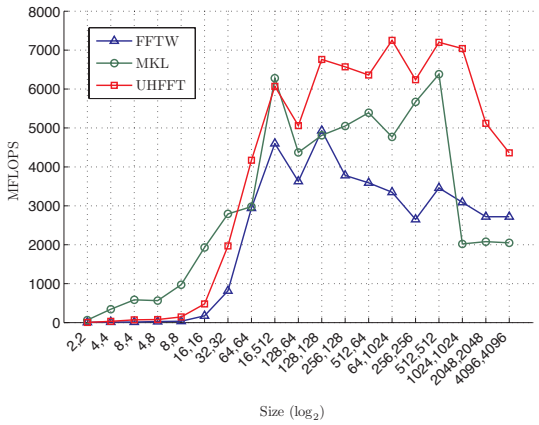
In this section, we compare the performance of the libraries for two-dimensional and three-dimensional out-of-place FFTs. Although, the UHFFT is quite efficient at computing arbitrary size, multidimensional FFTs, we discuss only powers-of-two size multidimensional FFTs. For each size, we compare the performance of both serial and parallel (multithreaded) execution. In the UHFFT, the search engine selects the schedule for each dimension separately (with corresponding strides) before combining them together in a multidimensional schedule. Even when the dimensions are of equal size, a separate schedule is searched to take into account the effects of different strides along each dimension. The multithreaded schedules are estimated in a straightforward fashion by dividing the problem along the last dimension.

Two-dimensional FFTs

Figures 7.15, 7.16 and 7.17, show the performance of two-dimensional out-of-place FFTs on the Itanium 2, Xeon Clovertown and Opteron 285 architectures. For most sizes, the UHFFT compares favorably with the other two libraries for both serial and parallel execution.

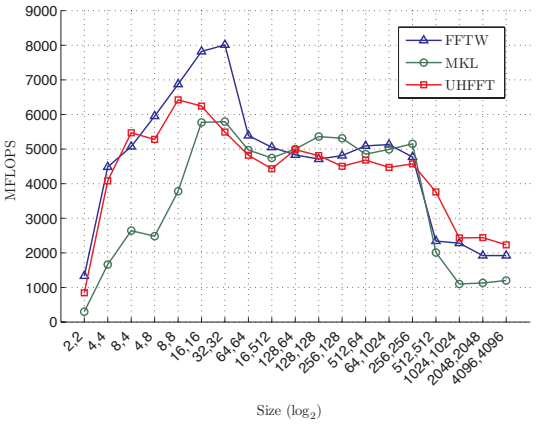


(a) Serial

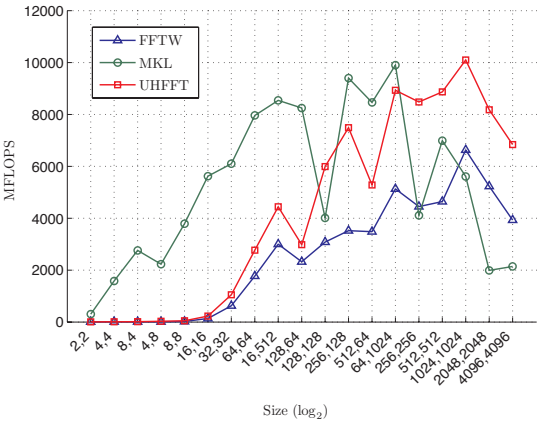


(b) Four threads

Figure 7.15: Itanium 2 - performance comparison of out-of-place complex to complex two-dimensional FFTs

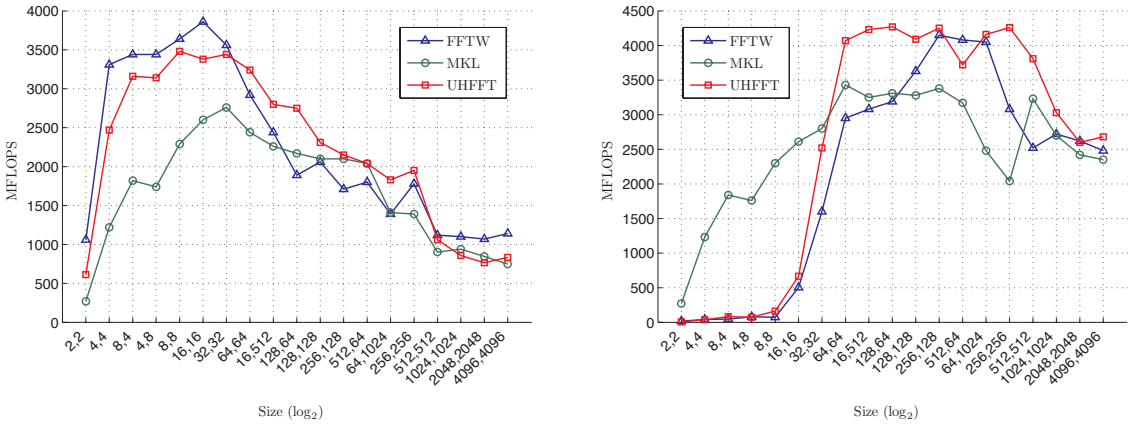


(a) Serial



(b) Eight threads

Figure 7.16: Xeon Clovertown - performance comparison of out-of-place complex to complex two-dimensional FFTs



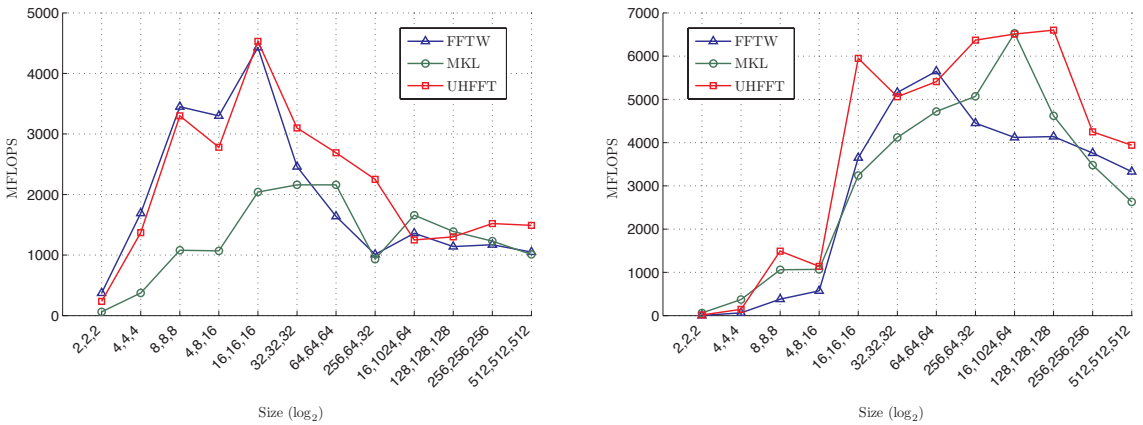
(a) Serial

(b) Four threads

Figure 7.17: Opteron 285 - performance comparison of out-of-place complex to complex two-dimensional FFTs

Three-dimensional FFTs

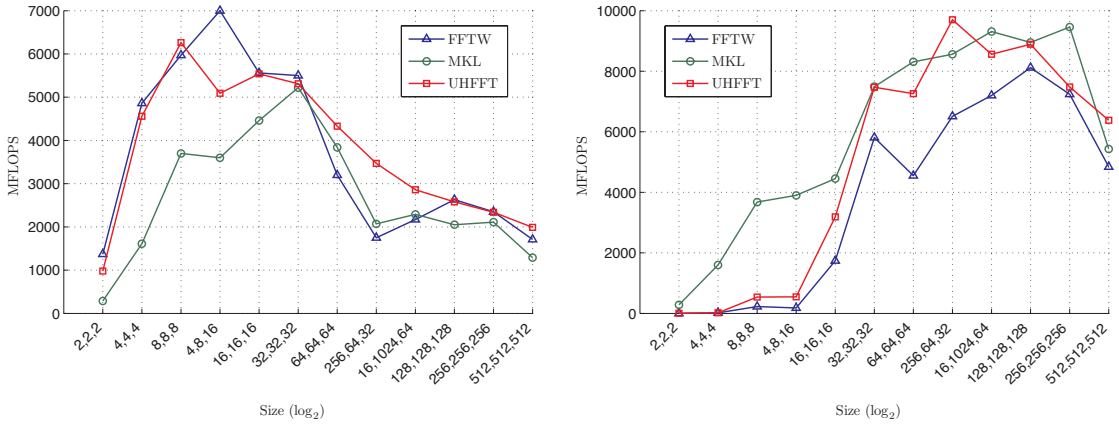
Figures 7.18, 7.19 and 7.20, show the performance of three-dimensional FFTs on the Itanium 2, Xeon Clovertown and Opteron 285 architectures. Like two-dimensional FFT, the UHFFT compares favorably with the other two libraries for both serial and parallel execution.



(a) Serial

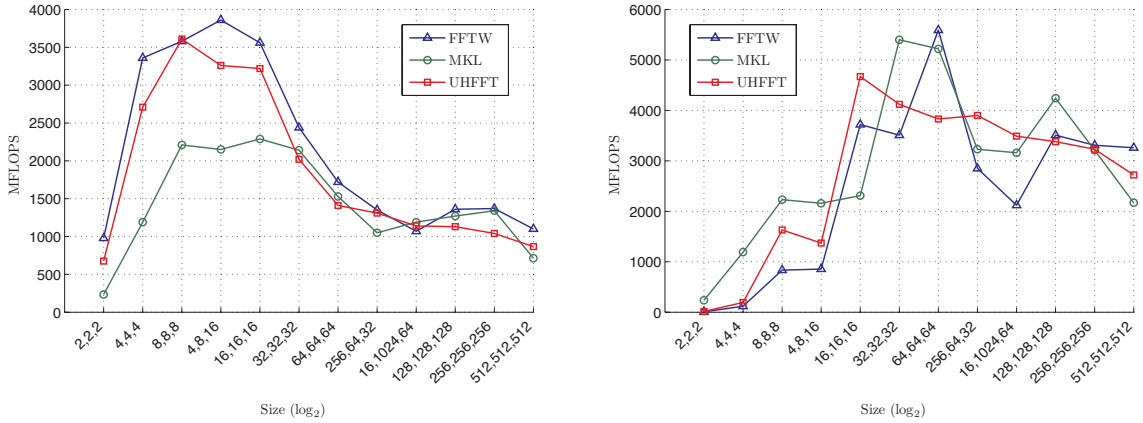
(b) Four threads

Figure 7.18: Itanium 2 - performance comparison of out-of-place complex to complex three-dimensional FFTs



(a) Serial (b) Eight threads

Figure 7.19: Xeon Clovertown - performance comparison of out-of-place complex to complex three-dimensional FFTs



(a) Serial (b) Four threads

Figure 7.20: Opteron 285 - performance comparison of out-of-place complex to complex three-dimensional FFTs

Table 7.1: Three representative sets of FFT sizes included in the benchmarking

No	Powers-of-two	Non-powers-of-two	Prime
1	2	6	17
2	4	9	31
3	8	12	61
4	16	15	127
5	32	18	257
6	64	24	509
7	128	36	1021
8	256	80	2053
9	512	108	4093
10	1024	210	8191
11	2048	504	16381
12	4096	1000	32771
13	8192	1960	65537
14	16384	4725	131071
15	32768	10368	262147
16	65536	27000	524287
17	131072	75600	1048573
18	262144	165375	
19	524288	720720	
20	1048576		
21	2097152		
22	4194304		
23	8388608		

Chapter 8

Conclusions

In this dissertation, we developed and evaluated a framework for constructing and executing dynamic schedules for FFT computation on hierarchical and shared-memory multiprocessor architectures. We presented the design of the UHFFT library that consists of four main components; a code generator, a formal language to express and construct the FFT schedules, an executor containing driver routines for serial and parallel computation of the FFT and a scheduler that implements the run-time search and estimation mechanism to find the best schedule of execution. To validate our methodology, we also presented performance comparisons of the UHFFT with FFTW and Intel's Math Kernel Library (MKL) on the Itanium 2, the Xeon Clovertown and a second generation Opteron. The results show that our implementations of various driver routines compare favorably against the FFTW and MKL libraries. We have shown that the UHFFT outperforms FFTW and MKL for most architectures on problems too large to fit in cache. Moreover, our low-overhead

multithreaded driver routines deliver better performance on multi-core and shared-memory multiprocessor architectures. We also presented a comparison of the various run-time search schemes implemented in the UHFFT and FFTW. Our results indicate that the model driven search can be accurate in predicting the best performing FFT schedule for a minimum cost of initialization. In the following, we list the main contributions of this dissertation.

8.1 Contributions

a. Code Generator

As part of this dissertation, we implemented an automatic empirical optimization mechanism in the FFT code generator, which has been integrated in the UHFFT library. The code generator was enhanced to automatically generate so called *twiddle* codelets, rotated PFA codelets, coupled in-place codelets and loop-vectorized codelets. Furthermore, the code generator also generates short vector (SIMD) variants of all the codelet types.

b. The FFT Schedule Specification Language

The code generated at installation time is assembled together at run-time to solve large FFT problems. We implemented a concise language based on simple context free grammar rules for specifying a variety of FFT schedules and algorithms. The FFT schedule specification language (FSSL) is particularly suited to hierarchical

memory and multi-core architectures since it specifies both schedule and blocking of the computation. The language also helps in better understanding of the correlation between an FFT schedule (algorithm and factorization) and its performance on the given architecture, which helps in the development of intuitive models (heuristics) that can be implemented in run-time search schemes to avoid expensive empirical analysis.

c. Serial and Parallel FFT Driver Routines

In the UHFFT, we have implemented efficient serial and parallel FFT driver routines for each of the FSSL rules. Our implementations of various driver routines compare favorably against the FFTW and Intel's MKL library. Our experiments show that the UHFFT outperforms FFTW and MKL for most architectures on problems too large to fit in cache. Moreover, our low-overhead multithreaded driver routines deliver better performance on multi-core architectures.

d. Multiple Run-time Search Methods

We have implemented three different run-time search methods in the scheduler, including a model-driven search that avoids run-time empirical analysis. Our preliminary results show that the model driven search can be accurate in predicting the best performing FFT schedule for a minimum cost of initialization.

e. Methodology for the Development of Domain Specific Adaptive Libraries

In general, this dissertation provides a complete recipe for developing adaptive domain specific libraries. Based on our experiences in the development of the UHFFT, following are the nine important ingredients of the recipe:

1. Automatic generation of compute intensive kernels that maximize the utilization of functional pipelines and available registers.

2. A specification of parameterized interfaces for the microkernels including the variants that may be optimized for special scenarios.

3. Optionally, a compiler feedback mechanism may be included that evaluates the versions of the same code type and selects the best version based on the empirical evaluation.

4. High resolution and accurate timers (or performance counters). The timers form an integral part of the adaptive libraries because the accuracy of the performance models and code scheduling decisions depends on the accuracy of high resolution timers.

5. A flexible and robust user interface to access the functionality of the library. The functionality should be accessible through a consistent user interface.

6. A simple language to specify the mathematical algorithms (driver routines) implemented in the library. The language not only provides an abstraction

layer for complex algorithms, it also ensures that the library can be easily extended for future generations of architectures. The language may also be useful in generating performance models, as it describes the data and control flow of the computation on a given architecture.

7. Various driver routines for serial and parallel execution of algorithms. The driver routines must have consistent interfaces so that they can be integrated seamlessly within the framework.

8. Multiple search schemes of varying costs. Ideally, the search methods should include a constant time search scheme and an empirical search scheme to provide the options of accuracy and speed.

9. Auxiliary tools that help in benchmarking and comparison of alternative implementations.

8.2 Limitations and Future Work

The UHFFT is a robust library for computing the DFT while maintaining performance portability across different architectures. However, it is an ongoing research work and is far from complete. Our future work will involve work on the current limitations of the library, which are given below:

- At the code generator level, the scalar optimizations should be applied to the DAG. For SIMD codelets in particular, the arithmetic simplifications may result in better overall performance.

- The code generator is fully capable of generating real and trigonometric transforms. In the future, we plan to integrate the generation of real, sine and cosine codelets in the UHFFT library.

- At run-time, data reordering may improve performance for certain architectures that have a small cache with a low degree of associativity, e.g., second generation Opteron processor. For those architectures, fast transposes may improve performance for very large sizes. Moreover, contiguous temporary buffers may be used as workspaces so that very large strides can be avoided.

- The model driven search does not include the prime factor algorithm (PFA) in its search space, which may result in lower performance for some non-powers-of-two sizes. In the future, we plan to extend the `dfti_low` search method to include all the algorithms.

- Large prime sizes are computed using Rader's algorithm, which may not be the best option for relatively small prime sizes. Furthermore, some large prime sizes may need recursive application of Rader's algorithm, which should be avoided using the zero-padding technique employed by the FFTW.

- Current implementation of the multidimensional in-place FFT is not as efficient as the out-of-place FFT because it uses a different schedule due to the absence of the output array. Using temporary buffers may improve the performance of in-place FFTs of single or multiple dimensions.

Bibliography

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-d fft. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 34–40, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[2] Ayaz Ali. Uhfft (prime size support). Technical report, University of Houston, 2004.

[3] Ayaz Ali. Impact of instruction scheduling and compiler flags on codelets' performance. Technical report, University of Houston, 2005.

[4] Ayaz Ali. Uhfft2 homepage. `www.cs.uh.edu/~ayaz/uhfft`, 2007.

[5] Ayaz Ali, Lennart Johnsson, and Dragan Mirkovic. Empirical Auto-tuning Code Generator for FFT and Trignometric Transforms. In *ODES: 5th Workshop on Optimizations for DSP and Embedded Systems, in conjunction with International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, March 2007.

[6] Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. Adaptive computation of self sorting in-place ffts on hierarchical memory architectures. In *High Performance Computation Conference (HPCC)*, pages 372–383, Houston, TX, September 2007.

[7] Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. Scheduling FFT Computation on SMP and Multicore Systems. In *International Conference on Supercomputing*, Seattle, WA, June 2007.

[8] AMD. *Software Optimization Guide for AMD64 Processors*, 25112-3.06 edition, September 2005.

[9] D. H. Bailey. A high-performance fast Fourier transform algorithm for the Cray-2. *The Journal of Supercomputing*, 1(1):43–60, ???? 1987.

[10] D. H. Bailey. Ffts in external or hierarchical memory. *J. Supercomput.*, 4(1):23–35, 1990.

[11] David H. Bailey. A high-performance FFT algorithm for vector supercomputers. *International Journal of Supercomputer Applications*, 2(1):82–87, 1988.

[12] David H. Bailey. Unfavorable strides in cache memory systems. *Sci. Program.*, 4(2):53–58, 1995.

[13] C. Sidney Burrus and Peter W. Eschenbacher. An in-place, in-order prime factor FFT algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29:806–817, 1981.

[14] C. Sidney Burrus and H. W. Johnson. An in-order, in-place radix-2 FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 9:473 – 476, 1984.

[15] Jacqueline Chame, Chun Chen, Pedro Diniz, Mary Hall, Yoon-Ju Lee, and Robert Lucas. An overview of the eco project. In *Parallel and Distributed Processing Symposium*, pages 25–29, 2006.

[16] S. C. Chan and K. L. Ho. On indexing the prime-factor fast fourier transform algorithm. *IEEE Transactions on Circuits and Systems*, 38(8):951–953, 1991.

[17] J.C. Cooley and J.W. Tukey. An algorithm for the machine computation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[18] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93:293–312, February 2005.

[19] Jack Dongarra and Francis Sullivan. Guest editors' introduction: the top 10 algorithms. *Computing in Science & Engineering*, 2:22–23, 2000.

[20] P. Duhamel and H. Hollmann. Split radix fft algorithm. *Electronics Letters*, 20(1):14–16, 1984.

[21] J.O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, 21:801–803, 1972.

[22] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W. Ueberhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[23] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. FFT program generation for shared memory: SMP and multicore. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 115, New York, NY, USA, 2006. ACM Press.

[24] Matteo Frigo. A fast Fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 169–180, New York, NY, USA, 1999. ACM Press.

[25] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[26] Matteo Frigo and Steven G. Johnson. Fftw homepage. `www.fftw.org`, 2007.

[27] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[28] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *LCPC '02: Proc. Languages and Compilers for Parallel Computers.*, pages 305–315, College Park, MD, USA, 2002.

[29] W. Morven Gentleman and G. Sande. Fast fourier transforms – for fun and profit. In *AFIPS, Fall Joint Computer Conference*, pages 563–578, Spartan, Washington, 1966.

[30] I.J. Good. The interaction algorithm and practical fourier analysis. *Journal of the Royal Statistical Society, Series B*, 20(2):361–372, 1958.

[31] Markus Hegland. A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing. *Numerische Mathematik*, 68(4):507–547, 1994.

[32] Intel Corporation. *Intel 64 and IA-32 Architectures: Optimization Reference Manual*, 248966-015 edition, May 2007.

[33] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, New York, NY, USA, 1981. ACM Press.

[34] Alan H. Karp. Bit reversal on uniprocessors. *SIAM Review*, 38(1):1–26, 1996.

[35] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

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

[37] Charles Van Loan. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[38] Qingda Lu, Sriram Krishnamoorthy, and P. Sadayappan. Combining analytical and empirical approaches in tuning matrix transposition. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 233–242, New York, NY, USA, 2006. ACM.

[39] Dragan Mirkovic and S. Lennart Johnsson. Automatic Performance Tuning in the UHFFT Library. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 71–80, London, UK, 2001. Springer-Verlag.

[40] Dragan Mirkovic, Rishad Mahasoom, and S. Lennart Johnsson. An adaptive software library for fast Fourier transforms. In *International Conference on Supercomputing*, pages 215–224, 2000.

[41] Phil Mucci. Papi homepage. `http://icl.cs.utk.edu/papi`, 2007.

[42] Kenji Nakayama. An improved fast fourier transform algorithm using mixed frequency and time decimations. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(2):290–292, 1988.

[43] David A. Patterson and John L. Hennesy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2007.

[44] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[45] C.M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proc. of the IEEE*, 56:1107–1108, 1968.

[46] M. Frigo S. G. Johnson. A modified split-radix fft with fewer arithmetic operations. *IEEE Trans. Signal Processing*, 55(1):111–119, 2007.

[47] A. Saidi. Decimation-in-time-frequency fft algorithm. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 453–456, 1994.

[48] Richard C. Singleton. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 17:93–103, 1969.

[49] H. V. Sorenson, M. Heidemen, and C. Sidney Burrus. On computing the split radix fft. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34(1):152–156, 1986.

[50] Paul N. Swarztrauber. Vectorizing the ffts. In *Parallel Computations*, pages 51–83. 1982.

[51] Daisuke Takahashi. A blocking algorithm for fft on cache-based processors. In *Proc. 9th International Conference on High Performance Computing and Networking Europe (HPCN Europe)*, pages 551–554. Springer-Verlag, 2001.

[52] Daisuke Takahashi. A blocking algorithm for parallel 1-d fft on shared-memory parallel computers. In *Proc. 6th International Conference on Applied Parallel Computing (PARA)*, pages 380–389. Springer-Verlag, 2002.

[53] Ping Tak Peter Tang. DFTI – A New Interface for Fast Fourier Transform Libraries. *ACM Transactions on Mathematical Software*, 31(4):475–507, December 2005.

[54] Clive Temperton. Implementation of a Self-Sorting In-Place Prime Factor FFT Algorithm. *Journal of Computational Physics*, 54:283–299, 1985.

[55] Clive Temperton. A new set of minimum-add small-n rotated DFT modules. *J. Comput. Phys.*, 75(1):190–198, 1988.

[56] Clive Temperton. Self-Sorting In-Place Fast Fourier Transforms. *SIAM Journal on Scientific and Statistical Computing*, 12(4):808–823, 1991.

[57] Clive Temperton. A generalized prime factor fft algorithm for any n=2p3q5r. *SIAM J. Sci. Stat. Comput.*, 13(3):676–686, 1992.

[58] L. H. Thomas. Using a computer to solve problems in physics. *Applications of Digital Computers*, 1963.

[59] Richard Tolimieri, Myoung An, and Chao Lu. *Algorithms for discrete fourier transform and convolution*. Springer-Verlag, 1997.

[60] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[61] R. Clint Whaley and David B. Whaley. Tuning High Performance Kernels through Empirical Compilation. In *ICPP '05: In Proceedings of the 2005 International Conference on Parallel Processing*, pages 89–98, Oslo, Norway, 2005. EEE Computer Society.

[62] S. Winograd. On computing the discrete fourier transform. *Mathematics of Computation*, 32:175–199, 1978.

[63] R. Yavne. An economical method for calculating the discrete fourier transform. In *Proc. AFIPS Fall Joint Computer Conf.*, volume 33, pages 115–125, 1968.

[64] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas. *Proceedings of the IEEE*, 93:358–386, February 2005.

[65] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. *SIGPLAN Not.*, 38(5):63–76, 2003.

[66] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 93–104, New York, NY, USA, 2007. ACM.

[67] Zhao Zhang and Xiaodong Zhang. Fast bit-reversals on uniprocessors and shared-memory multiprocessors. *SIAM Journal on Scientific Computing*, 22(6):2113–2134, 2001.