Technische Universität Wien
Vienna University of Technology

Dissertation

FFT Specific Compilation on IBM Blue Gene

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften
unter der Leitung von

Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Christoph W. Überhuber
E101 – Institut für Analysis und Scientific Computing

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Stefan Kral
Matrikelnummer 9625239
Anzengrubergasse 61/1/6
2380 Perchtoldsdorf

Perchtoldsdorf, am 22. Mai 2006

Vorwort

Algorithmen zur digitalen Transformation von Signalen sind im Scientific Computing von größter Bedeutung und werden in zahlreichen Gebieten angewendet – von der Echtzeit-Signalverarbeitung in eingebetteten Systemen bis zur numerischen Lösung partieller Differentialgleichungen im Rahmen komplexer Simulationen auf Supercomputern.

Die Entwicklung und Publikation der schnellen Fourier-Transformation (FFT) durch Cooley und Tukey leitete die Entwicklung einer neuen Klasse schneller Signalverarbeitungsalgorithmen ein, die – im Gegensatz zur direkten Auswertung des entsprechenden Matrix-Vektor-Produktes – nicht eine Berechnungskomplexität von $O(N^2)$, sondern nur von $O(N \log N)$ haben. Für eine bestimmte Transformation gibt es aber nicht nur *einen* eindeutig bestimmten schnellen Algorithmus, sondern eine ganze Vielzahl äquivalenter Algorithmen. Diese Algorithmen unterscheiden sich nur unwesentlich im Hinblick auf ihren Rechenaufwand, umso mehr aber in ihrem Speicherzugriffsverhalten, was auf modernen Computersystemen mit ihren mehrstufigen Speicherhierarchien zu enormen Laufzeitunterschieden führen kann.

Automatische Performance-Tuning-Systeme, wie zum Beispiel die den State-of-the-art verkörpernden Signalverarbeitungs-Programmbibliotheken Fftw und Spiral, führen auf einem gegebenen Computersystem eine Suche nach dem optimalen Algorithmus im Raum aller äquivalenten Algorithmen und Implementierungen durch. Da der Output von Performance-Tuning-Systemen aber in Form von C-Code erfolgt, ist deren Leistung durch die Qualität der verfügbaren Compiler beschränkt.

Die vorliegende Arbeit stellt einen neu entwickelten Special-Purpose-Compiler vor, der für die Übersetzung laufzeitkritischer Codes existierende C Compiler ersetzt. Dieser Compiler, die *Vienna MAP compiler tool-chain*, besteht aus den folgenden Komponenten, die speziell den Bedürfnissen von Signalverarbeitungscodes angepasst sind: (*i*) Der *MAP Vectorizer* extrahiert 2-weg SIMD-Parallelismus in numerischen Straight-Line-Codes. (*ii*) Der *MAP Optimizer* führt lokale Codeverbesserungen durch, wie sie von versierten Assembler-Programmierern manuell ausgeführt werden. (*iii*) Zuletzt erzeugt das *MAP Backend* Assemblercode für die Zielarchitektur.

Die wichtigste Ziel-Architektur des MAP-Compilers ist der IBM PowerPC 440 FP2 Prozessor, der in IBM Blue-Gene-Systemen – den derzeit schnellsten Supercomputern der Welt – eingesetzt wird. Fftw-Grundroutinen, die mit Hilfe der Blue-Gene-Version des MAP-Compilers übersetzt werden, erreichen eine Effizienz von bis zu 80% und damit die dreifache Leistung jener Objekt-Codes, die von der aktuellsten Version des optimierenden IBM XL C Compilers erzeugt werden.

Preface

Digital signal transforms are core algorithms in computational science and engineering, ranging from real-time signal processing in small-scale problems with stringent time constraints up to large-scale simulations based on partial differential equation solvers running on the world's largest supercomputers.

Starting with Cooley and Tukey's work on the *fast Fourier transform* (FFT), a vast class of fast signal transform algorithms has been developed, pushing the computational complexity down from $O(N^2)$ to $O(N \log N)$. In practice, however, there is not just one unique algorithm, but a large number of fast algorithms for computing one specific transform. These diverse algorithms are equivalent, but may differ significantly with regard to their memory access behavior, which causes tremendous runtime differences on all common-place machines with deep memory hierarchies.

Automatic performance tuning systems—like the state-of-the-art signal transform libraries FFTW and SPIRAL—search the space of suitable algorithms and implementations, automatically generating a large number of promising codes. To obtain the best performing code on a given target hardware, the search process is guided by empirical runtime measurements. However, as the program generators used in automatic performance tuning systems produce high-level C code, the performance of these systems is clearly limited by the quality of available compilers.

The thesis at hand presents a newly developed special-purpose compiler—the Vienna MAP compiler tool chain—as a replacement for general purpose high-level compilers in the context of automatic performance tuning systems. The MAP tool chain is composed of several generic components, consecutively focusing on specific properties of signal transform codes. (*i*) The MAP vectorizer extracts 2-way SIMD-style parallelism out of numerical straight line code. (*ii*) The MAP optimizer performs local code improvements similar to the ones that experienced programmers would achieve by hand. (*iii*) Finally, the MAP backend produces assembly code for the specific target architecture.

The primary target architecture of the MAP compiler is IBM's PowerPC 440 FP2 processor used in all Blue Gene systems, currently being the fastest supercomputers worldwide. FFTW's core routines were compiled by the Blue Gene version of the MAP compiler. The resulting assembly code boasts an unprecedented performance—reaching a level of 80% efficiency. That way, MAP compiled codes are up to three times as fast as object codes obtained with the latest version of IBM's optimizing XL C compiler.

Acknowledgements

I want to thank everybody who supported me in writing this thesis.

First, and most importantly, I want to thank my family—my mother Anna, my father Alfred, and my sister Maria. You probably don't know how much strength I get out of our family ties.

Next, I want to thank my advisor, Christoph Ueberhuber, for his encouragement, his patience, and for the enormous amount of feedback and support.

Also, I want to thank all members of Aurora group 5, in particular Franz Franchetti for motivating me to work on Blue Gene and for his ideas regarding code optimization on Blue Gene, Juergen Lorenz for his experiments and hints on compilers, Andreas Bonelli for his patience in explaining tensor products to me, and the late Herbert Karner for guiding my first steps in the field of applied mathematics.

I want to thank the authors of Fftw, Matteo Frigo and Steven Johnson, for their cooperation and their encouragement.

Also, I want to thank the following people at IBM—for their advice, organisational skills, and for helping me carry out experiments on pre-release Blue Gene hardware: Gheoghe Almasi, Jose Brunheroto, John Gunnels, Manish Gupta, Jose Moreira, and Ramendra Sahoo.

Finally, I want to thank the US Department of Energy at Lawrence Livermore National Laboratory (LLNL) and the Austrian Science Fund (FWF, SFB Aurora) for their financial support.

Stefan Kral

Contents

Chapter 1

Introduction

Digital signal transforms are core algorithms in computational science and engineering, ranging from real-time signal processing in small-scale problems with stringent time constraints up to large-scale simulations based on partial differential equation solvers running on the world's largest supercomputers.

This wide range of applications—each having its particular requirements with regard to type of transformation (linear vs. non-linear), data types (fixed-point vs. floating-point arithmetic), data formats (real, interleaved complex, split complex), one or multi-dimensional data—is covered by a large class of digital signal transform algorithms, packaged into libraries providing application programmers with well-tested, well-documented tools fitting a vast area of problems.

In the field of compute-intensive scientific numerical software, speed is a highly relevant demand on digital signal transform codes. Minimizing the runtime required for transforming data allows dealing with ever more complex problems and ever larger data sets. Thus, time is a major driving force in the development of software for computational science and engineering applications.

By transforming the power equation of physics, i. e., power = work/time, it follows that

$$\text{Time} = \frac{\text{Work}}{\text{Peak-Performance} \times \text{Efficiency}}. \tag{1.1}$$

This equation shows that there are basically three ways for minimizing the runtime required for solving a particular problem. Firstly, the work that needs to be done could be reduced, i. e., the algorithms' complexity could be minimized. Secondly, the peak performance of the computer system might be increased. Finally, the efficiency is to be maximized, i. e., the percentage of the peak performance actually achieved by the algorithm implemented on a given computer system has to be made as large as possible.

The three factors of runtime equation (1.1) are certainly not independent of each other. For example, new hardware features that help to increase the peak performance of a computer system may decrease the efficiency of certain programs and may often require new types of algorithms to achieve a proportional runtime reduction.

1.1 Parts of a Computer System

In the context of minimizing the time required for transforming digital signal data, the components of a general purpose computer system take different roles. Fig. 1.1 shows a hierarchy of the relevant parts of a computer system—comprising hardware and software—and symbolizes their impact on actual performance.

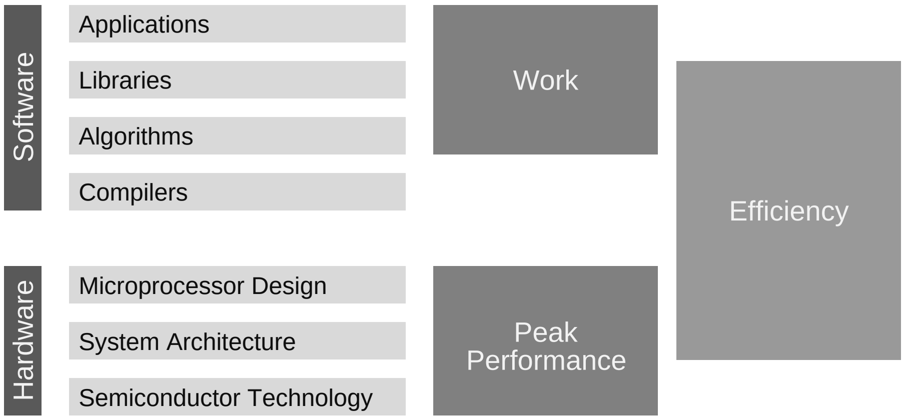

Figure 1.1: Hardware and Software Components of a Computer System. Any component of a computer system (depicted on the left) has an individual influence on the three factors of equation (1.1), i. e., work, efficiency, and peak performance. The peak performance of a system is determined by the hardware alone, work (complexity) is solely a software issue, while efficiency depends on both hardware and software.

1.1.1 Hardware Components

Hardware development—among other goals—aims at (*i*) pushing the aggregated peak performance of a computer system and at (*ii*) promoting high efficiency, i. e., making available a satisfactory percentage of the theoretical peak performance to many applications.

Semiconductor Technology provides the engineering processes required for producing highly integrated microprocessors as well as memory and logic chips.

In 1965 Intel co-founder Gordon E. Moore predicted that it would be technically feasible to regularly double the number of circuit elements integrated into a single chip, provided the market demand grew constantly to finance the required research and development.

Moore's prediction—commonly referred to as "Moore's Law" (see Fig. 1.2)—has held true for the past 40 years. It turned out to be an important factor in achieving the unbroken exponential growth of performance, as miniaturization progress allows increasing clock frequencies and to integrate more and more complex features.

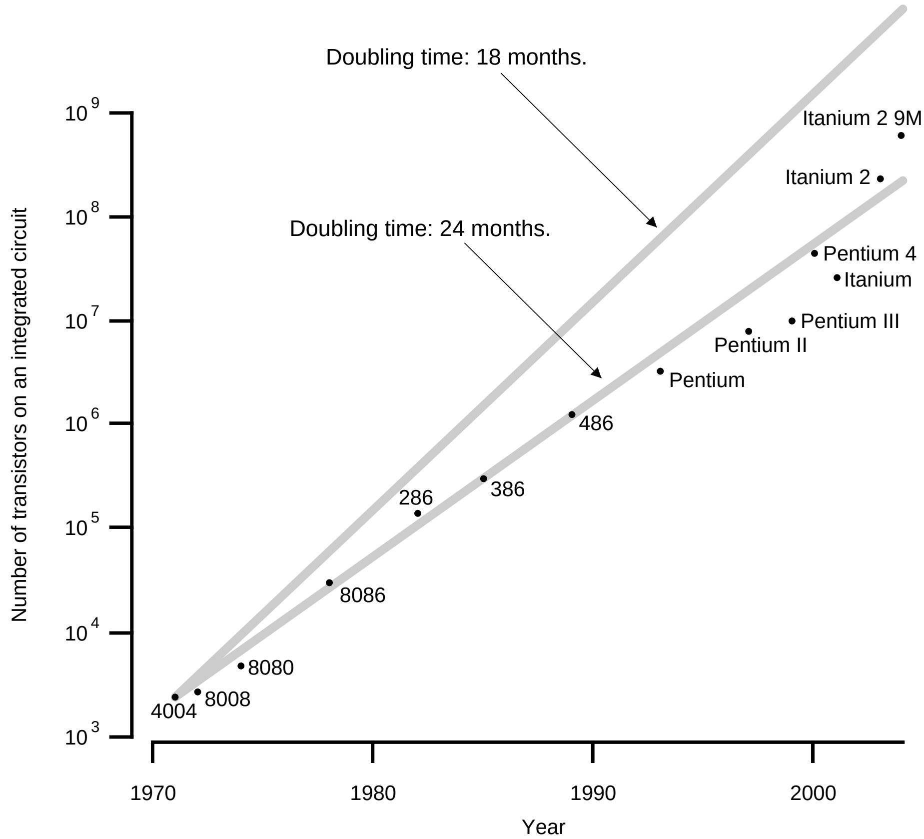

Figure 1.2: Moore's Law. The illustration shows the number of feature elements integrated into a selection of Intel microprocessors, ranging from the 4 bit 740 kHz "4004" to the 64 bit 1.6 GHz "Itanium 2 9M". The discussion of hardware development will focus on the widely used IA-32 (x86) processor family (8086, 286, 386, 486, and Pentium).

The semiconductor industry expects Moore's law to remain valid for the next 10 to 15 years, as reflected in the *international technology roadmap for semiconductors* (ITRS) [44].

System Architecture defines the large-scale structure and configuration of a computer system. A single system usually comprises several microprocessors, variable amounts of shared or distributed memory organized in a memory hierarchy, as well as peripherals like permanent storage devices, networking infrastructure, and audio/video hardware—typically linked by a high-speed interconnect optimized for high throughput and low access time.

Microprocessor Design produces processor models that can be mapped onto silicon utilizing the available process technology, defining both details directly visible to an assembly programmer—like the *instruction set architecture* (ISA) and the *application binary interface* (ABI)—as well as micro-architectural details about *how* instruction sequences are processed internally.

System architecture and *microprocessor design* turn the exponential growth of the number of circuits on a chip—made possible by advances in *semiconductor technology*—into a tantamount performance growth of cost-efficient computer systems, suitable for scientific, commercial, and particularly for personal computing, and, to an increasing extent, for the consumer electronics mass-markets, represented, for instance, by video game consoles.

While microprocessor design directly focuses on maximizing the performance of individual processing components, system architecture tries to aggregate the respective components in a way that leads to an optimal overall performance.

1.1.2 Software Components

Software development, as opposed to hardware development, aims at (*i*) minimizing the workload and at (*ii*) maximizing efficiency.

Algorithms are responsible for carrying out the workload of applications. For many problems in science and engineering, there are large sets of algorithms available to choose from for solving well-defined parts of a given problem or for combining solutions of subproblems. Individual algorithms may differ in many respects, including memory footprint, arithmetic complexity, best-case vs. worst-case number of operations, memory access patterns, size, steadfastness, generality, maintainability, and, quite likely, many more.

On a high-level, any particular algorithm requires a specific amount of work to solve a given problem, which is quite often characterized by abstract measures like asymptotic complexity classes.

On the actual implementation level, algorithms expressed by means of a programming language often have a profound impact on the efficiency of the code output by the compiler.

Libraries comprise a collection of algorithms in form of computer code, needed for solving a class of problems in some specific domain, conveniently offered through a documented library-application interface. Low-level libraries shield applications from specific target processor details. For instance, PAPI [66] provides an abstract interface to performance counters for enabling accurate runtime measurements.

Applications in computational science and engineering often utilize optimized libraries to solve specific subproblems efficiently.

Compilers transform a wide range of codes written in some high-level language into assembly code, aiming at good utilization of the target hardware and its specific features.

Optimizing compilers for general purpose microprocessors usually focus on maximizing speed or on minimizing code size. Special purpose compilers like optimizing compilers for *digital signal processors* (DSPs) may additionally focus on special issues like minimizing energy consumption [58].

For at least two reasons, compilers have significantly pushed back assembly hand coding efforts in almost any domain. Firstly, high-level languages allow programmers to focus on the software and to ignore hardware-specific features, which improves portability and maintainability of the codes and raises human productivity. Secondly, programmers gradually gained well-earned trust in the ability of the compilers to produce high-quality assembly code—one of the main reasons for the success of one of the earliest wide-spread high-level programming languages, Fortran [6].

System Software. The operating system has been deliberately left out of the following considerations, because its influence on the focus of this thesis—single-threaded codes working on data present in memory—is of minor importance.

1.2 Maximizing Peak Performance

Apart from raising the clock frequency, several techniques have proven successful for increasing the peak performance of a single chip and have found their way into the mainstream of general purpose computing during the past years. All these techniques directly aim at increasing parallelism, for instance, by raising the number of functional units available.

Dedicated Hardware Support for basic scalar data types is a necessary prerequisite for achieving competitive performance. For instance, starting with the Intel 486, the IA-32 family includes dedicated on-chip hardware for the support of floating-point arithmetic, which is more than ten times faster than emulating floating-point operations in software.

The amount of hardware features supplying basic floating-point operations differs considerably between various processor families, a fact reflected in the size of the instruction set. For instance, PowerPC processors offer a *fused multiply-add* (FMA) instruction, i. e., a floating-point instruction that combines a multiplication with a dependent addition, thus providing a means for effectively doubling the theoretical peak performance.

Instruction Pipelining can improve the instruction throughput by subdividing instructions into different stages, overlapping the execution of different instructions at different stages.

Because of the cost of mispredicted branches and because of dependencies between instructions, pipeline lengths cannot be arbitrarily increased without degrading the performance of the majority of codes. Such efficiency related issues are the reason why various hardware techniques like branch optimization and out-of-order execution, presented in the next section, have found their way into modern processor design.

Pipelining was a novel feature of the RISC design of the SPARC and the MIPS processor families [67], introduced in 1985. These processors have been highly successful on the workstation market, and set the stage for all modern processors featuring a RISC-style core. Eventually, this technology entered the desktop sector, with the Intel 486 in 1989.

Over the past twenty years, the number of instruction pipeline stages in general purpose processors has risen more than previously considered [67]. While the classic RISC pipeline had only five stages—instruction fetch, instruction decode, execute, memory access, and writeback—pipelines in contemporary processors have around 15 stages, the Intel Pentium 4 being a noticable exception with more than 30 pipeline stages. With their new architecture introduced in 2006, Intel made a considerable step back and reduced the pipeline length to 14.

Instruction Level Parallelism. Executing multiple instructions in parallel can be accomplished by either super-scalar execution or by *explicitly parallel instruction computing* (EPIC).

With super-scalar execution, the processor hardware is responsible for extracting parallelism out of a sequence of instructions. With EPIC, this responsibility is transferred to the compiler, thus allowing for a simpler design of the hardware.

All current mass-market processors are based on super-scalar execution, usually allowing to execute up to a handful instructions of different types (memory, integer, and floating-point) in parallel.

SIMD Style Parallelism. Starting with the mid-1990s, many processors included *single-instruction multiple-data* (SIMD) style instruction set extensions for efficiently dealing with relatively short vectors of fixed-lengths, which consist of scalar integer, floating-point, or pixel data.

Hardware support for SIMD style parallelism is still gaining momentum, with the latest example being Intel's Core Duo processor, introduced in 2006. Core Duo signifantly improves upon previous designs with regard to the throughput of SIMD instructions.

Multi-core Computing. Since the end of the 1990s, the continual combined increase of clock frequencies and pipeline depths has slowed down significantly. To further increase the peak performance, chip manufacturers started integrating multiple independent processor cores onto a single chip.

The first multi-core processors had two cores and targeted servers and workstations: IBM Power 4 (2000), HP PA-8800 (2003), Sun UltraSPARC IV (2004), AMD Opteron (2005), and Intel Xeon DP (2005).

In 2005, AMD and Intel introduced two new dual-core processors for desktop computers: the Athlon 64 X2 and the Intel Pentium 4D.

Multi-core technology entered the video game console market in late 2005, when Microsoft's Xbox 360—featuring the triple-core PowerPC-based IBM Xenon processor—became available.

As of mid-2006, another two important milestones were reached. First, Intel introduced Core Duo, the first dual-core processor for notebook computers. Then, Sun released the eight-core UltraSPARC T1 processor, which specifically targets multi-threaded web-servers, offering both multiple cores and chip-level multi-threading with up to four logical threads per core.

Ongoing and Future Development. Future processor development of general purpose processors will most likely see only marginal increases in pipeline depths and in the amount of super-scalar execution supported.

SIMD support is very likely to be improved (*i*) by adding useful new instructions that are currently still missing in most SIMD instruction set extensions and (*ii*) by adding more functional units available for use by SIMD instructions.

The number of cores integrated into one chip will steadily rise. The IBM cell processor, which comprises one main PowerPC core and eight *synergistic processor units* (SPU), will become available in late 2006 with the release of Sony's Playstation 3 game console. General purpose quadruple-core processors for both servers and desktop computers are anticipated for 2007. Also for 2007, Sun plans to release the UltraSPARC T2 that executes up to eight logical threads per core.

1.3 Maximizing Efficiency

In the past thirty years, development in process technology, general purpose computer hardware, and computer software has been influenced more and more by the demands of the mass market, particularly by personal computers.

Multi-level Memory Hierarchies. The need for cost-effective, highly-integrated *random access memory* (RAM) with moderate power consumption, put *dynamic RAM* (DRAM) ahead of *static RAM* (SRAM). As DRAM is significantly slower than SRAM, with the performance gap widening year by year, multiple levels of caches, varying in speed and size, were added as a buffer between the processor(s) and main memory.

While caches are usually fully transparent to the programmer, they may become a severe performance obstacle if an application does not exhibit high temporal and spatial locality of reference (see Fig. 1.3).

Modern processor hardware automatically prefetches data that are likely to be needed in the near future, and additionally supports software-directed prefetching. Both prefetching methods may contribute to increased performance, but may also deteriorate performance, if the data being prefetched are not used in time.

To reduce latencies, caches are often integrated into the processor leading to a mixed Harvard/von-Neumann architecture with unified level 2 and level 3 caches, a unified address space, but with the level 1 caches being split for instructions and data.

Processor Micro-Architecture Improvements aim at a high execution efficiency of processor cores.

Branch optimization—static or dynamic branch prediction and speculative execution—aims at minimizing branch related pipeline stalls.

Out-of-order execution enables the processor core to execute instructions in an order that is different from the one present in the machine code, i. e., instructions may be delayed until all required operands are available. Out-of-order execution has proven useful for increasing efficiency in the presence of long instruction pipelines and high memory latencies.

Chip-Level Multithreading. Based on the assumption that most threads are not utilizing all available multiple execution units at the same time, simultaneous multithreading allows the issuing of multiple instructions from multiple threads in one cycle, with multiple "logical" threads of execution competing for the very same computational resources.

While this technique may also be used for error detection and recovery, the main focus lies on increasing the throughput of a system running multiple threads.

Progress in Compiler Development. To produce high quality output for a wide range of input codes, modern compilers include sophisticated optimization mechanisms, addressing scalar, vector and parallel codes. Some optimizing compilers allow their decisions to be guided by data gained from dynamic profiling. The level of efficiency reached by a compiler depends on how well an input code in high-level language can be mapped to assembly code that exploits hardware specific features to a satisfactory degree.

1.4 Optimizing Signal Transform Performance

State-of-the-art signal transform software to be run on general purpose computers achieves high performance by explicitly addressing two issues that are not handled by existing general purpose compilers. Firstly, they aim at a minimization of the work required for any particular transform. And secondly, they focus on optimizing the order of memory access operations.

1.4.1 Minimizing Work

An important and often very successful way of minimizing the time required to solve a particular problem is to minimize the algorithm's complexity, i. e., to reduce the algorithmic workload that needs to be done. In some areas, like solving special linear systems, complexity reduction of algorithms was as successful as progress in hardware development reflected by Moore's law.

Fast Signal Transforms. In the case of linear digital signal transforms, the development of a revolutionary new class of algorithms with reduced arithmetic complexity has been initiated by the (re-)discovery of the *fast Fourier transform* (FFT) by Cooley and Tukey in 1965. The FFT is based on a factorization of the *discrete Fourier transform* (DFT) matrix into a product of sparse matrices, done in a recursive fashion.

Similar decompositions were derived for many other linear signal transforms, like the *Walsh-Hadamard transform* (WHT), various *discrete cosine transforms* (DCT), or certain wavelet transforms—and all are based on the same principles as Cooley and Tukey's approach.

Because of its great influence on "the development and practice of science and engineering", the FFT has been named one of the "Top Ten Algorithms of the 20th Century", the result of an international voting compiled by Dongarra and Sullivan [18].

Reduced Arithmetic Complexity. The divide-and-conquer style decomposition of the transform matrix pushes the problem complexity of the respective transform from $O(N^2)$, i. e., the complexity of general matrix-vector products, down to $O(N \log N)$. This kind of complexity reduction is particularly valuable because its speed-up effect gets bigger and bigger with increasing problem size.

For example, on general purpose sequential computers, well-written FFT codes always outperform straight-forward DFT codes implementing a full matrix-vector product, even for the smallest problem sizes. This performance advantage of the FFT over the DFT even increases as the problem size gets larger.

1.4.2 Efficiency Issues

The advent of deep memory hierarchies made clear that "making the common case fast in an economical way" may work well for many codes, but had a severe impact on the efficiency of digital signal transform codes.

Locality of Reference. While the formulation of many fast algorithms in Cooley-Tukey style naturally leads to a recursive implementation, considerable effort has been put into the design of FFT codes in iterative style, assuming that iterative constructs like loops have significantly lower overhead than recursive subroutine calls. Loop nested signal transformation algorithms, however, do not exhibit high locality of reference, which inevitably leads to a performance deterioration, in particular if the access speed of main memory is low.

To compare the locality of reference of iterative vs. recursive algorithms, Fig. 1.3 illustrates two in-place 16-point radix-2 Cooley-Tukey style algorithms calculating the Walsh-Hadamard transform (WHT). The illustration on the left shows the behavior of an iterative algorithm, the one on the right the behavior of an equivalent recursive version.

The different memory access patterns illustrated by Fig. 1.3 are responsible for several problems. Firstly, the iterative access pattern is more prone to cache capacity misses, as the algorithm's storage access operations linearly progress

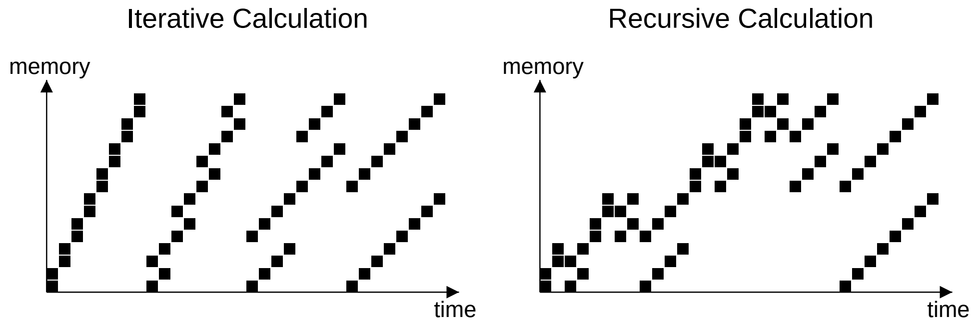

Figure 1.3: Comparison of Memory Access Patterns of Two $\mathbf{WHT_{16}}$ Implementations. In both address-space/time diagrams, a consecutive part of memory holding the signal data being transformed is presented on the y-axis, while the x-axis marks advances in time. A dot at position (i, j) tells that the algorithm accesses the i^{th} element of some data vector at the j^{th} time-step of the calculation.

over the entire data several times, while the recursive algorithm splits the original transform into two, solves them separately and combines the respective results. Secondly, common-place systems with 2^k-way set associative caches, may have a lot more cache conflict misses running an iterative algorithm than by running an equivalent recursive version, especially if the vector size is a power of two, like in the chosen example.

Problems Arising from Cache Associativity. As Cooley-Tukey style algorithms have a lower arithmetic operation count for power-of-two than for non-power-of-two vector lengths, a bizarre situation occurs on systems with 2^k-way set associative caches: Algorithms with the lowest operation count are the ones that are hit the hardest by cache conflict misses due to the order of memory access operations.

Choice of an Efficient Algorithm. Iterative signal transform algorithms have low control-flow related overhead. Recursive algorithms have better locality of reference. To get the best of both worlds, iterative and recursive algorithms may be combined.

For a given transform type, there is, in general, an exponentially growing number of different iterative/recursive Cooley-Tukey style algorithms. All these different algorithm versions perform a comparable number of arithmetic operations—often within a very narrow range of a few percent—and still, their performance varies significantly, mostly because of different orders of memory accesses.

1.4.3 Automatic Performance Tuning

The step from programming one fixed algorithm for solving a given problem towards searching the space of possible algorithmic implementations marks the appearance of a new paradigm in high-performance computing, automatic performance tuning.

Based upon empiric runtime measurement instead of explicit performance modeling, automatic performance tuning allows creating portable software that covers *all* performance critical aspects of a target machine, including those that cannot be modeled easily, like memory access behavior. Taking runtime as the objective function of the planning process does not only focus on hardware related issues, but also on the software side, i. e., algorithms and compilers.

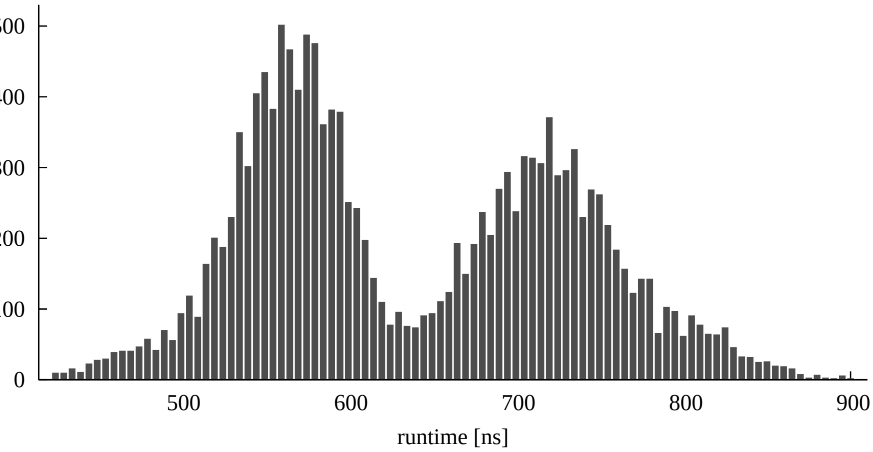

Figure 1.4: Runtime Comparison of 15,778 Implementations of a $\mathbf{DCT}_{16}^{\mathbf{IV}}$. This diagram from [71] shows a histogram of runtimes for 15,778 straight-line codes computing the type IV discrete cosine transform for vectors of length 16, $\mathrm{DCT}_{16}^{\mathrm{IV}}$. All codes were automatically generated by SPIRAL, compiled using *GCC* 2.95, and measured on an Intel Pentium 4 with 1.8 GHz. While the number of arithmetic operations performed by these codes differs only by a few percent—the number of additions ranges from 96 to 104 and the number of multiplications from 48 to 56—their runtime spans from 430 to 900 nanoseconds, more than a factor of 2. Only 1.5% of the codes are within a 10% range of the fastest code found.

One of the motivations for automatic performance tuning is to overcome the limitations of existing general purpose compilers. Fig. 1.4 shows that general purpose compilers may exhibit paradox behavior when applied to comparatively long sequences of straight line code. To obtain satisfactory performance, the following options are available, each having its own advantages and shortcomings.

(*i*) SPIRAL puts the compiler into the optimization loop, which guards against compiler idiosyncrasies, as a large number of codes are actually generated and evaluated. However, the size of this optimization loop significantly contributes to long adaptation times.

(*ii*) FFTW has a somewhat smaller optimization loop, which speeds up the automatic adaptation process, but also necessitates an adaptation of the domain-specific code generator to the peculiarities of the compilers used [28].

(*iii*) The work presented in this thesis aims at overcoming all the compiler-related problems by replacing the general purpose compiler by a domain-specific compiler, the MAP tool chain. By focusing on straight line code, the design and implementation of the MAP compiler is kept as simple as possible. Although this approach is considerably less portable than the other ones, it is a successful way to higher performance.

Automatic performance tuning has been introduced as an algorithmic method for achieving high efficiency of code operating on large data sets, relieving compilers of the responsibility to optimize the order of data access operations acting on deep memory hierarchies. This paradigm has been particularly successful for libraries in the field of digital signal transforms [69, 30] and in linear algebra [16].

FFTW – The Fastest Fourier Transform in the West

In 1997, Matteo Frigo and Steven G. Johnson presented FFTW [29, 30], the "Fastest Fourier Transform in the West", a portable, high-performance library for computing complex and real discrete Fourier transforms in one or many dimensions. In the meantime, FFTW has become the de-facto standard in FFT software.

Planning. Unlike other FFT libraries, FFTW does not rely on one fixed algorithm to perform a requested transform, but uses a separate stage of planning to determine how to efficiently solve a specific problem on some given target machine using a fixed set of DFT kernel routines called codelets. As planning is guided by empirical measurement, it is quite likely that—for non-trivial problem sizes—FFTW actually executes two different algorithms when run on two sufficiently different machines. The knowledge gained in the planning stage, called "wisdom", may be stored for later reuse. When performing a transformation, FFTW traverses plans, calling the specified kernel routines as requested.

Automatic FFT Code Generation. FFTW's kernel routines are generated automatically by a special purpose FFT code generator called `genfft`. Apart from many others, `genfft` is a remarkable piece of software because of two reasons. Firstly, it was able to automatically derive new algorithms previously unknown in the signal processing community. And secondly, `genfft` demonstrated how high-level code can profit both from domain specific optimization techniques like FFT-graph based high-level scheduling and compiler specific tricks like providing tight life-spans for temporary variables to the C compiler.

Acceptance and Recognition. FFTW is the well-established de-facto standard for computing the DFT, ranging from desktop computers—where it has been integrated into the MATLAB system—up to the largest supercomputers like IBM's Blue Gene systems. It offers portable high performance on a wide range of machines, often beating other available free and commercial libraries, especially in cases that require the processing of large amounts of data. In recognition of their efforts, the authors of FFTW were awarded the prestigious James H. Wilkinson Prize for Numerical Software in 1999. Current development of FFTW is presented in [30].

SPIRAL – Signal Processing Algorithms Implementation Research for Adaptive Libraries

Spiral [61] is a generator system for DSP transform software and hardware, primarily developed by a group of researchers at Carnegie-Mellon University. Spiral operates on its own special-purpose high-level language—the *signal processing language* (SPL)—that allows describing linear signal transformations concisely in a special mathematical notation and to explicitly express relevant semantic information.

Similar to Fftw, Spiral performs a search for the best-suited implementation of a given transform. Unlike Fftw, Spiral is not packaged as a stand-alone library including a number of widely used codes, but rather as a library generator. The current status and ongoing development in Spiral is presented in [69].

1.4.4 Utilizing SIMD Extensions

High-level compilers have long shielded the application programmer from the complexity of many internal features of orthogonal processor architectures.

This situation changed considerably when special purpose SIMD instruction set extensions were added to general purpose processors to push the performance of multimedia applications.

Compiler Support for SIMD Instructions may be provided by general purpose compilers in two different ways.

Firstly, a compiler may completely hide the complexity of these features, by offering automatic SIMD vectorization of loops or basic blocks, possibly aided by user-supplied directives ("pragmas").

Secondly, the compiler may take the easy way out and expose the complexity of the extensions, supplying special vector types and ISA specific intrinsic functions operating on values of these types. This does not only push the responsibility for the utilization from the compiler to its users, but also results in non-portable code because of the inherent differences of existing SIMD instruction set extensions.

Compiler Inadequacies. Several experiments have shown disappointing results of general purpose compilers when handling digital signal transform codes (see Fig. 1.4). This has the following reasons.

Firstly, most codes to be compiled in the context of performance tuning systems are automatically generated by a program generator and can be very large, pushing some compilers to their limit.

Secondly, the code may not profit from loop-level vectorization, which tries to identify multiple independent iterations of a loop that can be executed simulatenously using vector instructions, simply because it does have any.

Thirdly, there may be more than one way to map a scalar code to short vector SIMD hardware. Unless a vectorizer performs a tremendous amount of search, it is very likely to produce SIMD code with poor performance.

Finally, linear transform code has a rich internal structure that cannot be seen directly in the code. Knowledge about this particular structure can be utilized to guide the vectorization process, and thus allows producing satisfactory output in reasonable time.

High-Level Vectorization. Due to the limitations of existing compilers, existing successful approaches for automatically creating high-performance SIMD enabled digital signal transform code address vectorization on the algorithmic (or the library) level, performing algebraic manipulations of high-level domain specific language constructs, utilizing rich semantic information. The resulting code is then transformed to non-portable high-level code with intrinsics.

1.5 The Vienna MAP Compiler Tool Chain

SIMD extensions led to a revival of hand coding and hand tuning of non-portable code, especially in a number of performance critical applications.

This thesis presents special compilation techniques for creating single-threaded high-performance digital signal transforms codes, closing the performance gap between code generated by general purpose compilers and assembly code written by skilled hand-coders.

The Vienna MAP compiler has been developed to focus on compute intensive straight line codes as generated automatically by special purpose program generators like FFTW's `genfft`.

Structure of the Compiler. The Vienna MAP compiler comprises a set of relatively generic components, arranged in the form of a flexible tool chain, that allows for easy experimentation with different arrangements of compilation phases, including feedback driven optimization loops.

Unlike other program generators, MAP does not produce C code, but covers the entire range of compilation, from domain-specific high-level code to target specific assembly code.

Human-readable Representations. All input, output, and intermediate program representations are fully human-readable. The compiler does not only allow introspection at any given point, but also the injection of code into the compiler.

The injection of code helps with the development and maintenance of the compiler and aids the understanding of details of the compilation process, which can get quite complex, due to interactions between tool chain components.

The intermediate representation supports the annotation of instructions to preserve high-level information (e. g., about variable aliasing, variable types, etc.) throughout the compilation.

Optimizations performed by the MAP compiler include newly developed techniques for utilizing advanced hardware features like 2-way SIMD-style functional

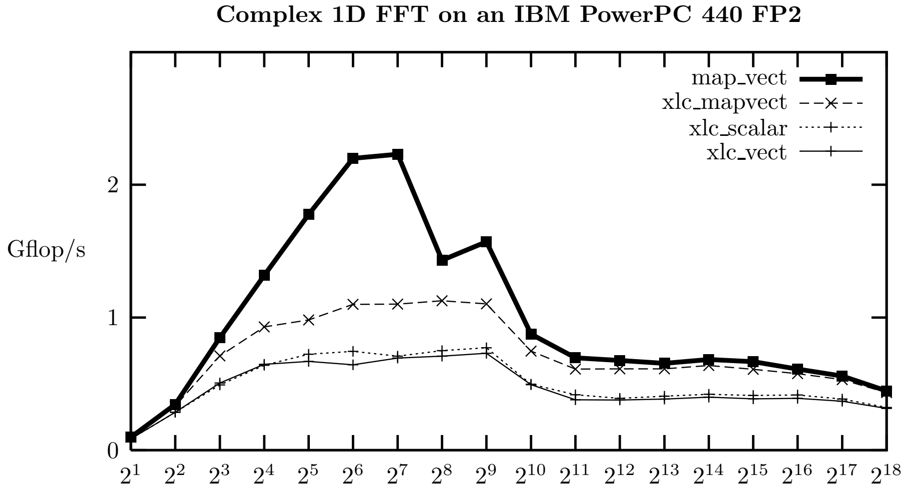

Figure 1.5: Performance Comparison for Power-of-two 1D FFTs. This plot shows the performance of FFT routines compiled by the following compilers and settings. *xlc_scalar* uses the IBM XL C compiler without automatic vectorization. *xlc_vect* uses IBM XL C with automatic vectorization. *xlc_mapvect* uses the MAP vectorizer and optimizer, producing C code with SIMD intrinsics compiled by IBM XL C. *map_vect* uses the MAP vectorizer, optimizer, and backend.

units and FMA operations, as well as established compiler backend techniques, like optimal register allocation and list-based instruction scheduling for superscalar architectures.

Maintainability and Portability. Many components of the MAP tool chain (e. g., vectorizer, register allocator, and instruction scheduler) can easily be adapted to any current target architecture by using parameter files that describe relevant parts of the target architecture.

Implementation on IBM's Blue Gene Systems. The newly developed techniques have been integrated into a special version of the MAP tool chain aiming at the Blue Gene systems' IBM PowerPC 440 FP2 processors.

To assess the performance gain obtained with the presented techniques, code produced by the MAP compiler was integrated into the automatic performance tuning library Fftw to create a state-of-the-art high-performance PowerPC specific version of Fftw targeting at IBM's Blue Gene supercomputer series, i. e., Fftw-Gel for Blue Gene.

Fig. 1.5 compares the performance of FFT routines on a single processor of an IBM Blue Gene system, clearly demonstrating that all parts of the MAP compiler significantly improve the floating-point performance. In the best case, MAP reaches as much as 80% of the optimum performance, thus beating the latest version of IBM's optimizing XL C compiler by up to a factor of three.

Fftw-Gel for Blue Gene has been used in a number of applications, including the runner-up of the 2005 Gordon Bell Award, the large-scale molecular dynamics simulation code Qbox [39], which achieved a sustained floating-point performance of 60 Tflop/s. Recent developments of application codes utilizing Fftw-Gel reach an even more outstanding performance level.

Synopsis

This thesis consists of three main parts: (*i*) Foundations, (*ii*) the Vienna MAP compiler, and (*iii*) experimental results.

Foundations

Chapter 2 provides the mathematical foundations of discrete linear transforms and in particular of the fast Fourier transform.

Chapter 3 presents the software architecture of automatic performance tuning software and the SIMD instruction set extensions available on modern microprocessors, particularly focusing on the peculiarities of the Blue Gene processor.

The Vienna MAP Compiler

The main part of thesis presents the newly developed special-purpose Vienna MAP compiler for IBM Blue Gene in detail.

Chapter 4 describes a new method for the automatic 2-way SIMD vectorization of numerical straight line code, which produces globally optimal code in a large number of cases, including complex FFT kernels of arbitrary lengths and real FFT kernels of even lengths.

Chapter 5 illustrates domain-specific peephole optimization techniques aiming at a good utilization of target specific hardware features like SIMD FMAs.

Chapter 6 describes domain-specific backend optimization techniques and their implementation in a backend specifically designed for the IBM PowerPC 440 FP2 processor.

Chapter 7 illustrates the functioning of the Vienna MAP compiler as a whole, describing its input, its output, its ways of intermediate program representation, as well as its compilation process by tracking a selection of example codes through all relevant stages.

Experimental Results

Chapter 8 presents and discusses performance data of code generated by the Vienna MAP compiler, which have been measured recently on state-of-the-art Blue Gene systems that are currently the fastest computers worldwide.

Chapter 2

Fast Algorithms for Linear Transforms

Discrete linear transforms are important tools used in a wide range of real-life applications, ranging from spectal analysis of signals, to the lossy compression of video and audio data, to voice recognition, to the multiplication of large integers or polynomials, to solving partial differential equations.

This chapter defines and discusses discrete linear transforms and fast algorithms for such transforms, following the methodology introduced by the Spiral team [61]. The approach is based on Kronecker product factorizations of transform matrices and on recursive factorization rules.

2.1 Discrete Linear Transforms

This section defines discrete linear transforms as a foundation for the specific discussion of fast Fourier transform algorithms in the next section. In this thesis, the main focus is on the discrete Fourier transform and its fast algorithms based on the Cooley-Tukey recursion.

Discrete linear transforms are represented by real or complex valued matrices and their application means to calculate a matrix-vector product. Thus, they express a base change in the vector space of sampled data.

Definition 2.1.1 (Real Discrete Linear Transform) *Let $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, and $M \in \mathbb{R}^{m\times n}$. The real linear transform of x is obtained by the matrix-vector multiplication*

$$y = M\,x.$$

Examples of real discrete linear transforms include the Walsh-Hadamard transform as well as all sine and cosine transforms.

Definition 2.1.2 (Complex Discrete Linear Transform) *Let $x \in \mathbb{C}^n$, $M \in \mathbb{C}^{m\times n}$, and $y \in \mathbb{C}^m$. The complex linear transform of x is given by the matrix-vector multiplication*

$$y = M\,x.$$

A particularly important example of complex discrete linear transforms and the main focus in this thesis is the discrete Fourier transform (DFT), which, for size N, is given by the following two definitions.

Definition 2.1.3 (Discrete Fourier Transform Matrix) *The matrix* DFT_N *is defined for any* $N \in \mathbb{N}$ *with* $i = \sqrt{-1}$ *by*

$$\mathrm{DFT}_N = \left(e^{2\pi ik\ell/N} \mid k, \ell = 0, 1, \ldots, N-1\right).$$

The values $\omega_N^{k\ell} = e^{2\pi ik\ell/N}$ *are called* twiddle factors.

Example (DFT Matrices) The first five DFT matrices are

$$\mathrm{DFT}_1 = (1), \quad \mathrm{DFT}_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \mathrm{DFT}_3 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & e^{-2\pi i/3} & e^{-4\pi i/3} \\ 1 & e^{-4\pi i/3} & e^{-2\pi i/3} \end{pmatrix}$$

$$\mathrm{DFT}_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}, \quad \mathrm{DFT}_5 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & e^{-2\pi i/5} & e^{-4\pi i/5} & e^{-6\pi i/5} & e^{-8\pi i/5} \\ 1 & e^{-4\pi i/5} & e^{-8\pi i/5} & e^{-2\pi i/5} & e^{-6\pi i/5} \\ 1 & e^{-6\pi i/5} & e^{-2\pi i/5} & e^{-8\pi i/5} & e^{-4\pi i/5} \\ 1 & e^{-8\pi i/5} & e^{-6\pi i/5} & e^{-4\pi i/5} & e^{-2\pi i/5} \end{pmatrix}.$$

DFT_4 is the largest DFT matrix having only *trivial* twiddle-factors, i. e., 1,i,-1,and $-i$.

Definition 2.1.4 (Discrete Fourier Transform) *The discrete Fourier transform* $y \in \mathbb{C}^N$ *of a data vector* $x \in \mathbb{C}^N$ *is given by the matrix-vector product*

$$y = \mathrm{DFT}_N\, x.$$

2.1.1 Fast Algorithms

Many discrete linear transforms can be calculated by using fast algorithms, which reduce the algorithmic complexity from $O(N^2)$—as required by the direct evaluation of a matrix-vector product—down to $O(N \log N)$. This complexity reduction, and the resulting runtime reduction, makes these transforms suitable for the processing for large amounts of data.

Example (FFT) The complexity reduction factor obtained by using an FFT algorithm instead of plain matrix-vector multiplication is approximately $2N^2/(5N \log_2 N)$. For $N = 2^{20}$ this complexity reduction is equivalent to the progress of 20 years in hardware development.

Mathematically, any fast algorithm for a discrete linear transform can be viewed as a factorization of the transform matrix into a product of sparse matrices. It is a specific property of discrete linear transforms that these factorizations are highly structured and can be written in a very concise way using the formalism of Kronecker (tensor) products [82].

Two particularly important classes of sparse matrices arising in FFT factorizations are stride permutation matrices and twiddle factor matrices. The product of an input vector with these sparse matrices can be performed with $O(N)$ cost.

The permutation operator L_n^{mn} sorts the components of x according to their index modulo n. Thus, components with indices equal to $0 \bmod n$ come first, followed by the components with indices equal to $1 \bmod n$, and so on.

Definition 2.1.5 (Stride Permutation) *For a vector $x \in \mathbb{C}^{mn}$ with*

$$x = \sum_{k=0}^{mn-1} x_k e_k^{mn} \quad \textit{with} \quad e_k^{mn} = e_i^n \otimes e_j^m, \quad \textit{and} \quad x_k \in \mathbb{C},$$

the stride permutation L_n^{mn} is defined by its action on the tensor basis of $\mathbb{C}^{mn}$:

$$\mathrm{L}_n^{mn}(e_i^n \otimes e_j^m) = e_j^m \otimes e_i^n.$$

Twiddle factor matrices are certain diagonal matrices, whose elements are roots of unity.

Definition 2.1.6 (Twiddle Factor Matrix) *Let $\omega_N = e^{2\pi i/N}$ denote the Nth root of unity. The twiddle factor matrix, denoted by T_m^{mn}, is a diagonal matrix defined by*

$$\mathrm{T}_m^{mn}(e_i^m \otimes e_j^n) = \omega_{mn}^{ij}(e_i^m \otimes e_j^n), \quad i = 0, 1, \ldots, m-1,\ j = 0, 1, \ldots, n-1,$$

$$\mathrm{T}_m^{mn} = \bigoplus_{i=0}^{m-1} \bigoplus_{j=0}^{n-1} \omega_{mn}^{ij} = \bigoplus_{i=0}^{m-1} \Omega_{n,i}(\omega_{mn}),$$

where $\Omega_{n,k}(\alpha) = \mathrm{diag}(1, \alpha, \ldots, \alpha^{n-1})^k$.

Example ($\mathbf{DFT_4}$) Consider a sparse factorization, i. e., a fast algorithm, for DFT_4. Using the mathematical notation from [82] it follows that

$$\mathrm{DFT}_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

$$= \left(\begin{array}{cc|cc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ \hline 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{array}\right) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix} \cdot \left(\begin{array}{cc|cc} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{array}\right) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

$$= (\mathrm{DFT}_2 \otimes \mathrm{I}_2) \quad \cdot \quad \mathrm{T}_2^4 \quad \cdot \quad (\mathrm{I}_2 \otimes \mathrm{DFT}_2) \quad \cdot \quad \mathrm{L}_2^4.$$

T_2^4 denotes the twiddle matrix $\mathrm{diag}(1, 1, 1, i)$. L_2^4 denotes a stride permutation that swaps the two middle elements x_1 and x_2 of four-dimensional vectors, i. e.,

$$\begin{pmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{pmatrix} = \mathrm{L}_2^4 \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}.$$

2.1.2 Automatic Derivation of Fast Algorithms

In [22] a method has been introduced that *automatically* derives fast algorithms for a given transform and size. This method is based on algebraic symmetries of the transformation matrices utilized by the software package AREP [21], a library for the computer algebra system GAP [37] used in SPIRAL. AREP is able to factorize transform matrices and to find fast algorithms automatically. In [68] an algebraic derivation of fast sine and cosine transform algorithms is described.

2.1.3 Recursive Rules

One key element in factorizing a discrete linear transform matrix into sparse factor matrices is the application of *breakdown rules*.

A breakdown rule describes the factorization of a given transform matrix into into several transforms of smaller sizes. These smaller transforms, which may be of a different type, can be further expanded. Thus, rules can be applied recursively to reduce a large linear transform to a number of smaller discrete linear transforms.

The applicability of breakdown rules depends on the transform size, implicitly encoded into the left-hand side of a rule. For instance, in the following definitions, the rule for breaking down a WHT may only be used for transform lengths being a power-of-two. Variables occuring on the left-hand side of a rule are used as parameters of the expression on the right-hand side.

In the following examples P_n, P'_n, and P''_n denote permutation matrices, S_n denotes a bidiagonal matrix and D_n a diagonal matrix [85].

Example (Walsh-Hadamard Transform) The WHT_N for $N = 2^k$ is given by

$$\mathrm{WHT}_{2^k} = \overbrace{\mathrm{DFT}_2 \otimes \ldots \otimes \mathrm{DFT}_2}^{k \text{ times}}.$$

A particular example of a breakdown rule for this transform is

$$\mathrm{WHT}_{2^k} = \prod_{i=1}^{k} \left(\mathrm{I}_{2^{k_1+\cdots+k_{i-1}}} \otimes \mathrm{WHT}_{2^{k_i}} \otimes \mathrm{I}_{2^{k_{i+1}+\cdots+k_t}}\right), \quad k = k_1 + \cdots + k_t. \tag{2.2}$$

Example (Discrete Cosine Transform) The matrix DCT_N for arbitrary N is given by

$$\mathrm{DCT}_N = \big(\cos\left((\ell + 1/2)k\pi/N\right) \mid k, \ell = 0, 1, \ldots, N-1\big).$$

A corresponding breakdown rule is

$$\mathrm{DCT}_{2n} = P_{2n}\left(\mathrm{DCT}_n \oplus S_{2n}\ \mathrm{DCT}_n\ D_{2n}\right) P'_{2n} \left(\mathrm{I}_n \otimes \mathrm{DFT}_2\right) P''_{2n}.$$

Example (Discrete Fourier Transform) A rule for the DFT_N matrix is given by

$$\mathrm{DFT}_{mn} = \left(\mathrm{DFT}_m \otimes \mathrm{I}_n\right)\ \mathrm{T}^{mn}_n \left(\mathrm{I}_m \otimes \mathrm{DFT}_n\right)\ \mathrm{L}^{mn}_m\,. \tag{2.3}$$

(2.3) is the Cooley-Tukey FFT written in Kronecker product notation [45]. This particular rule will be discussed in more detail in Section 2.2.

Transforms of higher dimension are captured in the framework of breakdown rules in a natural way. For example, if M is the matrix of an $N \times N$ transform, then the corresponding two-dimensional transform is given by $M \otimes M$. Using the respective property of the tensor product, the breakdown rule

$$M \otimes M = (M \otimes \mathrm{I}_N)\,(\mathrm{I}_N \otimes M) \tag{2.4}$$

is obtained.

2.1.4 Formulas and Base Cases

Eventually a mathematical *formula* is obtained when all transforms are expanded into base cases.

Example (Fully Expanded Formula for $\mathbf{WHT_8}$) According to rule (2.2), WHT_8 can fully be expanded into

$$(\mathrm{DFT}_2 \otimes \mathrm{I}_4)(\mathrm{I}_2 \otimes \mathrm{DFT}_2 \otimes \mathrm{I}_2)(\mathrm{I}_4 \otimes \mathrm{DFT}_2)$$

with DFT_2 being the base case.

2.1.5 Trees and Recursion

The recursive decomposition of a discrete linear transform into smaller ones using recursion rules can be expressed by trees. The authors of Fftw call these trees *plans* while the Spiral team calls them *rule trees*. In these trees the essence of the recursion—the type and sizes of the child transforms—is specified.

As an illustrative example, rule trees for a recursion rule that breaks down a transform of size N into two smaller transforms is discussed. Fig. 2.1 shows a tree of a discrete linear transform of size $N = mn$ that is decomposed into two smaller transforms of the same type of size m and n. The node marked with mn is the *parent node* of the *child nodes* lying directly below, which indicate transforms of size m and n.

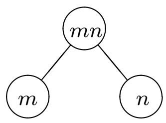

Figure 2.1: Tree representation of a discrete linear transform of size $N = mn$ with one recursion step applied.

Analogously, Fig. 2.2 shows a tree of a discrete linear transform of size $N = kmn$ where in a first step the transform is decomposed into discrete linear transforms of size k and mn. In a second step the transform of size mn is further decomposed into transforms of size m and n.

In general, the splitting rules are *not* commutative with respect to m and n. Thus, the trees are generally *not* symmetric. *Left* and *right child nodes* have to

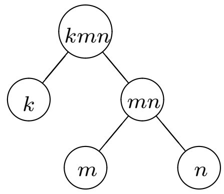

Figure 2.2: Right-expanded tree, two recursive steps.

be distinguished, which is done simply by left and right branches. Every tree has one *root node*, i. e., a node having no parent nodes. Nodes without children are called *leaf nodes*. All remaining nodes are intermediate nodes.

The upmost recursive decomposition in a tree, the one of the root node, is called the *top level decomposition*. If its two branches are equivalent the tree is called *balanced*, if they are nearly equivalent it is said to be "somewhat balanced." But there also exist trees that are not balanced at all. They may be even extremely unsymmetrical. A tree with just leafs as left children is formed strictly to the right. Such a tree is called *right-expanded*, its contrary *left-expanded*.

2.1.6 The Search Space

By selecting different breakdown rules, a given discrete linear transform expands to a large number of formulas that correspond to different fast algorithms. For example, for $N = 2^k$, there are $k-1$ ways to apply rule (2.3) to DFT_N. A similar degree of freedom recursively applies to the smaller DFTs obtained, which leads to $O(5^k/k^{3/2})$ different formulas for DFT_{2^k}. In the case of the DFT, allowing breakdown rules other than (2.3) further extends the formula space.

The problem of finding an efficient formula for a given transform translates into a search problem in the space of formulas for that specific transform. The size of the search space depends on the rules and transforms actually used.

The conventional approach to solving the search problem is to make an educated guess (using some machine characteristics as hints) which formula might lead to an efficient implementation and then to continue by optimizing this formula.

The automatic performance tuning systems Spiral and Fftw use a radically different approach. Instead of explicitly modeling machine characteristics and their impact on performance, both systems use intelligent search, guided by empirical runtime measurements, to find implementations well-adapted to some particular target. Spiral uses various search strategies and fully expands the formulas. Fftw uses *dynamic programming* and restricts its search to the coarse grain structure of the algorithm [30]. The rules are hardcoded into the executor while the fine grain structure is fixed by the codelet generator `genfft` at compile time.

2.2 The Fast Fourier Transform

In the last section, discrete linear transforms and the discrete Fourier transform were briefly introduced using a special mathematical notation, which made clear that such transforms feature an intrinsic recursive structure.

In this section, different types of the Cooley-Tukey recursion are presented. The difference between conventional iterative algorithms and the recursive approach used by SPIRAL and FFTW is discussed. Conventional iterative algorithms and vector computer algorithms are summarized.

2.2.1 The Cooley-Tukey Recursion

In 1965 Cooley and Tukey [13] published the fast Fourier transform for problem sizes being powers of two. Other authors extended the idea from powers of two to arbitrarily composed numbers. A summary of the historic development of the FFT, whose origins date back to Carl Friedrich Gauss, can be found in [82].

In the decades following Cooley and Tukey's publication, FFT algorithms were obtained by applying breakdown rules recursively and then manipulating the resulting formulas to obtain the respective iterative algorithms. However, in the context of this thesis, the recursive rules are more important than the iterative algorithms. Section 2.2.2 discusses the difference between the iterative and the recursive approach.

There are four equivalent formulations of the Cooley-Tukey recursion rule, shown in Theorem 2.2.1. These versions can be transformed into each other by exploiting algebraic identities of tensor products, effectively choosing whether the identity matrix is the left or the right factor of the first and the second tensor product occuring in the rule.

Theorem 2.2.1 (Cooley-Tukey Breakdown Rules) *For $mn \geq 2$*

$$\begin{aligned}
\mathrm{DFT}_{mn} &= (\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}_n^{mn}\, (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}_m^{mn} && \textbf{(DIT)}\\
&= \mathrm{L}_n^{mn}\, (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{T}_n^{mn}\, (\mathrm{DFT}_m \otimes \mathrm{I}_n) && \textbf{(DIF)}\\
&= (\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}_n^{mn}\, \mathrm{L}_m^{mn}\, (\mathrm{DFT}_n \otimes \mathrm{I}_m) && \textbf{(Vector)}\\
&= \mathrm{L}_m^{mn}\, (\mathrm{I}_n \otimes \mathrm{DFT}_m)\, \mathrm{L}_n^{mn}\, \mathrm{T}_n^{mn}\, (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}_m^{mn}\,. && \textbf{(Parallel)}
\end{aligned}$$

With *decimation in time* (DIT), the initial step is a permutation, which allows to perform the transform out-of-place without any additional copying overhead. The *vector* (or *four-step*) and the *parallel* (or *six-step*) FFT rules aim at a formulation exhibiting terms that are inherently well-suited for expressing vector or parallel operations. Vector and parallel rules are usually applied only once on top-level.

The parallel rule of Theorem 2.2.1 was developed by Bailey [7]. A second recursive application of this rule is the prerequisite for parallel one-dimensional FFT algorithms which overlap communication and computation [25, 47].

2.2.2 Iterative vs. Recursive FFT Algorithms

This section outlines the two basic strategies in organizing FFT programs. For the implementation of an FFT algorithm there are two radically different strategies for successively applying one of Cooley-Tukey breakdown rules.

In recursive FFT algorithms, the multiplication by the DFT matrix is performed by calling program modules that compute the subproblems according to the chosen rule. The child problems are further decomposed by recursively calling the same program again and again until the DFTs are small enough to be executed directly by optimized leaf routines. This approach is used by SPIRAL and FFTW.

Iterative FFT code contains the entire matrix decomposition explicitly and manages all tasks directly. Thus, all recursive decomposition steps are flatted and the computation is done stagewise leading to conventional triple loop FFT implementations [82] that perform the DFT computation stagewise. Each stage requires an additional pass through the data vector.

Recursive FFTs

While divide-and-conquer descriptions of the FFT are standard in introductory texts, almost all non-adaptive high performance FFTs use an iterative implementation. This is due to the widespread opinion that recursive implementations of divide-and-conquer algorithms are too expensive. This belief is based on the fact that the required function calls were among the computationally most expensive instructions for a long time.

However, an intriguing feature of divide-and-conquer algorithms is that they should run well on computers with deep memory hierarchies without the need for blocking or tiling. Each successive divide step in the divide-and-conquer process generates subproblems that touch successively smaller portions of data, thus increasing locality of reference. For any level of the memory hierarchy, there is a level of division below which all the data touched by the subproblem will fit into that level of the memory hierarchy. Therefore, a divide-and-conquer algorithm can be viewed as an algorithm that is blocked for all levels of the memory hierarchy.

This idea of achieving optimal use of caches on all levels of a memory hierarchy without knowledge of their sizes is refered to as *cache-obliviousness* [32].

In the recent development of computer systems, memory access operations became more and more expensive when compared to function calls. Several years ago FFTW [29] broke with the tradition of iterative FFTs and implemented a recursive hardware adaptive FFT computation.

SPIRAL [61, 69] extended this recursive approach to all kinds of discrete linear transforms. For each transform and problem size SPIRAL generates intrinsically recursive code, that is partially unrolled for even higher performance.

Iterative FFTs

All conventional non-adaptive FFT algorithms and their implementation have an iterative structure. The first FFT algorithm published by Cooley and Tukey was a right-expanded radix-2 factorized FFT, whose decomposition strategy was clear and explicitly implemented. For decades the execution of an FFT was seen as a sequence of computational stages; each stage corresponding to one factor of the products that define algorithms in the modern notation.

Typically, Cooley-Tukey type FFT algorithms for dealing with data vectors of length $N = 2^n$ are implemented in form of a *triple-loop* [82].

While the design and programming of such implementations is rather easy [82], their performance is often not optimal because vector lengths vary from stage to stage and therefore the cache usage far from being optimal. Yet, as long as the input vector fits into the cache memory entirely the iterative strategy was superior because there is no overhead due to additional program organization, which was expensive on earlier computer generations. On current computer systems, however, such a clear statement is not possible any more.

Vector Computer FFT Algorithms and Short Vector Extensions

The two Stockham FFT algorithms [80, 82] and the 4-step FFT algorithm have been designed specifically for conventional vector computers. In principle, these algorithms could be used also on current processors featuring short vector SIMD extensions, but they have several drawbacks there.

Complex Arithmetic. Vector computer FFT algorithms are formulated using complex matrices. Thus, it is necessary to reformulate complex transforms using real matrices and formulas to capture the level of details required for implementations successfully utilizing short vector SIMD extensions.

Vector Length and Stride. All three algorithms mentioned above are optimized for *long vectors*. The Stockham algorithms are optimized for *fixed stride* but not for *unit stride* memory access. Accordingly, these algorithms do not produce good performance when running on short vector SIMD extensions.

Iterative Algorithms. The very nature of the two Stockham FFT algorithms and the 4-step FFT algorithm is an iterative one, which conflicts with the requirements of Spiral and Fftw to support adaptivity.

Thus, algorithms specifically designed for conventional vector computers are not suitable for the short vector SIMD extensions of modern processors. To harness the performance potential offered by SIMD extensions, methods specifically adressing the specific features and limitations of the respective SIMD instruction set extensions are required.

Chapter 3

Software and Hardware Architecture

In the context of producing efficient digital signal transform codes, a number of software and hardware features need to be taken in account.

Section 3.1 is devoted to current software architecture in scientific computing and presents the state-of-the-art in architecture adaptive numerical software used in the field of linear algebra and digital signal transforms [62]. The following three sections are devoted to modern hardware architecture. Section 3.2 describes short vector SIMD ISA extensions, focusing on their basic properties, their utilization, as well as similarities to vector supercomputers. A basic discussion of 2-way SIMD instructions follows in Section 3.3. Finally, Section 3.4 introduces the IBM PowerPC 440 FP2, the processor used in IBM's Blue Gene supercomputers, and describes its most important features, characteristics, and limitations.

3.1 Automatic Performance Tuning Software

Automatic performance tuning goes a step beyond standard compiler optimization. It is a problem specific approach and thus is able to achieve much more than general purpose compilers are capable of. For instance, Atlas' search for the correct loop tiling for carrying out a matrix-matrix product is a loop transformation a compiler could do in principle (and some compilers actually try to do), if the compiler had an accurate machine model to deduce the correct tiling. But compilers do not reach Atlas' performance by far. The same phenomenon occurs with the source code scheduling done by Spiral and Fftw for their automatically generated straight line code, which should be done satisfactorily by the target compiler. Again, available general purpose compilers fail to produce well-performing object code.

3.1.1 Compiler Optimization Methodology

Modern compilers make extensive use of optimization techniques to improve the object code's performance. Most advanced optimization techniques depend on static program analyses based on simplified machine models. These optimization techniques include high-level loop transformations, such as loop unrolling and tiling. These methods have been extensively studied for over 30 years and have produced, in many cases, satisfactory results. However, the underlying machine models are inherently inaccurate, and transformations are not independent from each other in their effect on performance. Thus, the compiler's task of deciding the best sequence of transformations is a very difficult one [2].

Typically, compilers use heuristics that are based on averaging observed behavior for a small set of benchmarks. Furthermore, while processors and memory hierarchies are typically modeled by static analysis, this does not account for the behavior of the entire system. For instance, the register allocation policy and strategy for introducing spill code in the backend of the compiler may have a significant impact on the resulting performance. Thus, static analysis is a means to improving program performance, which is limited by compile-time decidability.

3.1.2 The Program Generator Approach

One of the foundations of source code adaptation at compile-time is automatic code generation. In this approach, a *code generator*, i. e., a program that produces other programs, is used. The code generator takes as parameters certain information concerning the source code adaptations to be made, e. g., instruction cache size, choice of combined or separate multiply and add instructions, lengths of floating-point and fetch pipelines, and so on. Depending on these parameters, the code generator produces source code hopefully having the desired characteristics.

3.1.3 Compile Time Adaptive Algorithms Using Feedback Information

Not any important architectural variable can be handled by *parameterized* compile-time adaptation. Adaptation to some architectural variables would require a modification of the underlying source code. This brings in the need for the second method of software adaptation, compile-time adaptation by *feedback directed* code generation, which involves actually generating different implementations of the very same algorithm and selecting the best performing one.

There are at least two different ways to proceed:

(*i*) The simplest approach is to get the programmer to supply various hand-tuned implementations, and then to choose a suitable one.

(*ii*) The second method is based on automatic code generation. In this approach, parameterized code generators are used. Performance optimization with respect to a particular hardware platform is achieved by searching, i. e., varying the generator's parameters, benchmarking the resulting routines, and selecting the fastest implementation. This approach is also known as *automated empirical optimization of software* (AEOS) [88].

In the remainder of this section, existing performance tuning software is described briefly.

PHiPAC

Portable high-performance ANSI C (PHiPAC) was the first system that implemented the "generate and search" methodology [9]. PHiPAC's code generator produces matrix-matrix multiply implementations with various loop unrolling depths, varying register and L1- and L2-cache tile sizes, different software pipelining strategies, and enables other options. The output of the generator is C code, both to make the system portable and to shift responsibility to the compiler for performing the final register allocation and instruction scheduling. The search phase benchmarks code produced by the generator under various options to select the best performing implementation.

ATLAS

The *automatically tuned linear algebra software* (ATLAS) project is an ongoing research effort of Jack Dongarra's group at the University of Tennessee, Knoxville, focusing on empirical techniques in order to produce software having portable performance. Initially, the goal of the ATLAS project was to provide a portably efficient implementation of the *basic linear algebra subroutines* (BLAS). Now ATLAS provides at least some level of support for all of the BLAS, and first tentative extensions beyond this level have been taken.

While originally the ATLAS project's principle objective was to develop an efficient library, today the field of investigation has been extended. Within a couple of years new methodologies to develop self-adapting programs have become established, the AEOS approach has been established, thus forming a new sector in software evolution. ATLAS' adaptation approaches are typical AEOS methods; even the concept of AEOS—automated empirical optimization of software—was coined by ATLAS' developers [88]. In this manner, the second main goal of the ATLAS project is a general investigation in program adaptation using AEOS methodology.

ATLAS uses automatic code generators in order to provide different implementations of a given linear algebra operation, and involves sophisticated search scripts and robust timing mechanisms in order to find the best performing way of carrying out this operation on a given architecture. More information on ATLAS can be found in [16].

FFTW

The *fastest Fourier transform in the west* (FFTW) was the first effort to automatically generate FFT code using a special purpose compiler and using measured runtime as optimization criterion [29, 30]. On many architectures, FFTW performs better than publicly available FFT codes and as least as good as hand optimized vendor-supplied libraries across different machines. Several extensions to FFTW exist, including the AC FFTW [33] package and the UHFFT library [59].

Currently, FFTW is the most widely used portable high performance FFT library that is publicly available.

FFTW provides a recursive implementation of the Cooley-Tukey FFT algorithm. The actual computation is done by automatically generated routines called *codelets* which form the base case of the Cooley-Tukey recursion. For a given problem size there are many different ways of solving the problem with strongly differing runtimes. FFTW uses dynamic programming which takes the actual runtime as cost function to find a performance optimized code for calculating a given DFT_N on a particular machine. FFTW consists of the following fundamental parts.

The Planner. At runtime but as a one time operation during the initialization phase, the *planner* uses dynamic programming to find a good decomposition of the problem size into a tree of computations according to the Cooley-Tukey recursion called *plan*.

The Executor. When solving a problem, the *executor* interprets the *plan* as generated by the planner and calls the appropriate codelets with the respective parameters as required by the plan. This leads to data access patterns which respect memory access locality.

The Codelets. The actual computation of the FFT subproblems is done within the *codelets*. These small routines come in two flavors, (*i*) *twiddle codelets* which are used for the left subproblems and additionally handle the twiddle matrix, and (*ii*) *no-twiddle codelets* which are used in the leaf of the recursion and which additionally handle the stride permutations. Within a larger variety of FFT algorithms is used, including the Cooley-Tukey algorithm, the split-radix algorithm, the prime factor algorithm, and the Rader algorithm [82].

The Codelet Generator genfft. At install time, all codelets are generated by a special purpose compiler called the *codelet generator* `genfft` [31, 28]. As an alternative the preponderated codelet library can be downloaded as well. In the standard distribution, codelets of sizes up to 64 (not restricted to powers of two) are included. But if special transform sizes are required, the required codelets can be generated.

Details about FFTW can be found in [29].

SPIRAL

SPIRAL (*signal processing algorithms implementation research for adaptive libraries*) is a generator for high performance code for discrete linear transforms [61] being developed by a group of researchers at Carnegie-Mellon University (CMU). SPIRAL uses a mathematical approach that translates the implementation problem of discrete linear transforms, like the DFT, the discrete cosine transforms (DCTs), and many others, into a search in the space of structurally different algorithms and their possible implementations, to generate code that is adapted to a given computing platform. SPIRAL's approach is to represent the many different algorithms for a transform as formulas in a concise mathematical language, the signal processing language SPL. These formulas are automatically generated and automatically translated into code, thus enabling an automated search. More specifically, SPIRAL is based on the following observations.

- For every discrete linear transform transform there exists a *very large* number of different *fast* algorithms. These algorithms differ in dataflow but are essentially equal in the number of arithmetic operations.
- A fast algorithm for a discrete linear transform can be represented as a *formula* in a concise mathematical notation using a small number of mathematical constructs and primitives.
- It is easy to *automatically generate* alternative formulas, i. e., algorithms, for a given discrete linear transform.
- A formula representing a fast discrete linear transform algorithm can be mapped *automatically* onto a program in a high-level language like C or Fortran.

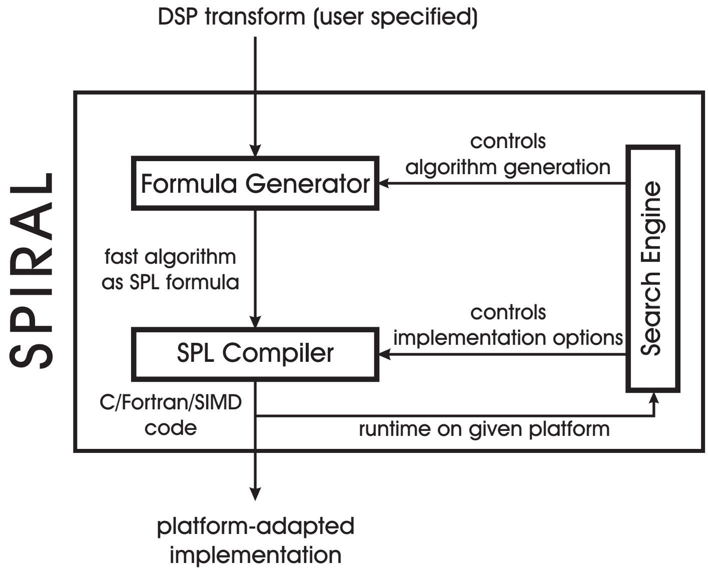

Figure 3.1: General view of SPIRAL's architecture.

The architecture of SPIRAL is schematically depicted in Fig. 3.1. The user specifies a digital signal processing (DSP) transform to be implemented, e. g., a discrete Fourier transform of size 1024. The *formula generator* expands the transform into one or several formulas, i.e., algorithms, represented in the SPIRAL proprietary signal processing language SPL. The *formula translator*—also called SPL compiler—translates a formula into C or Fortran code. The runtime of the generated code is fed back into a *search engine* that controls the generation of the next formula and makes possible implementation choices, such as the degree of loop unrolling. Iteration of this process finally yields a platform-adapted implementation of the desired transform. Search methods in SPIRAL include dynamic programming and evolutionary algorithms. By including the mathematics of signal transforms into the system, SPIRAL can optimize, akin to a human expert programmer, on the implementation level *and* the algorithmic level to find the best match to the given platform. Details about SPIRAL can be found in [70, 74, 90].

3.2 Short Vector SIMD Extensions

One of the most important hardware features with respect to performance optimization in scientific computing *single-instruction multiple-data* (SIMD) instructions. Major vendors of general purpose microprocessors have included SIMD extensions to their *instruction set architectures* (ISA) to improve the performance of multi-media applications by exploiting the subword level parallelism available in most multi-media kernels.

All current SIMD extensions are based on the packing of large registers with smaller data-types (usually of 8, 16, 32, or 64 bits). Once packed into larger registers, operations are performed in parallel on the separate data items within the vector register.

While the first SIMD extensions only operated on vectors of integers, newer extensions provide floating-point support, making them useful in the context of scientific computing. Table 3.1 gives an overview over the SIMD floating-point capabilities found in current microprocessors. Section 3.3 provides an elaborate presentation of the basics of 2-way SIMD ISA extensions.

Table 3.1: Floating-Point SIMD Instruction Set Extensions. For each SIMD ISA, the vector length, the calculation precision, and the list of supporting processors is provided.

SIMD ISA Name	Type	Processor(s)
3DNow!	$2 \times$ single	AMD K6-II+
Ext. 3DNow!	$2 \times$ single	AMD K6-II+, Athlon, Opteron
SSE	$4 \times$ single	Intel Pentium III, AMD Opteron
SSE2	$2 \times$ double	Intel Pentium 4, AMD Opteron
SSE3	$2 \times$ double	Intel Pentium 4e, AMD Opteron
Altivec (VMX)	$4 \times$ single	Motorola PowerPC G4
IA64	$2 \times$ single	Intel Itanium
FP2	$2 \times$ single/double	IBM PowerPC 440 FP2

Common short vector SIMD extensions increase the arithmetic operation throughput by a factor two, as shown in Table 3.2.

Table 3.2: Scalar and 2-way SIMD Arithmetic Operation Throughput. The PowerPC 440 FP2 can execute 2 additions/subtractions per cycle or alternatively 2 multiplications per cycle, or 2 multiplications with 2 dependent additions/subtractions in an FMA.

Processor(s)	ISA	Add/Sub	Mul	Total
Intel Pentium 4	scalar	1	1	1
IBM PowerPC 440	scalar	1	1	1 (2)
AMD Athlon/Opteron	scalar	1	1	2
Intel Pentium 4	SSE2/SSE3	1	1	2
AMD Opteron	SSE2/SSE3	1	1	2
AMD Athlon	3DNow!	2	2	4
IBM PowerPC 440	FP2	2	2	2 (4)

Because of the increases in operation throughput, the addition of a SIMD extension to a target processor renders existing legacy scalar codes obsolete, as such codes utilize only a small fraction of the potential peak performance.

Data Streaming Support. One of the key features needed in fast multi-media applications is the efficient streaming of data into and out of the processor. Multimedia programs such as video decompression codes stress the memory system in ways that multi-level cache hierarchies cannot handle efficiently, operating on volumes of data much larger than first-level caches. Streaming memory systems and the respective compiler optimizations aim at reducing memory latency (for example, via prefetching) and have the potential to improve these applications' performance, by (*i*) prefetching data before it is actually used and by (*ii*) controling the cacheability of data, thereby minimizing cache pollution.

3.2.1 SIMD Restrictions

Utilizing SIMD extensions efficiently is by no means an easy or straightforward task, as the SIMD extensions impose strong restrictions on algorithms.

Data Alignment. Common-place SIMD extensions can only access *naturally aligned vectors* efficiently. Although some extensions (e. g., AMD 3DNow!) support loading sub-vectors or accessing unaligned vectors, these operations are more costly than aligned vector accesses. With SSE2 and FP2, accessing non-naturally aligned data causes an internal processor exception, requiring costly operating system intervention to align the data and to restart the application.

Mixing Scalar and SIMD Code. SIMD extensions found in general purpose microprocessors usually do not allow the efficient mixing of SIMD and scalar instructions operating on the same data set. AMD 3DNow! does not support scalar operations on vector data at all, while Intel SSE2 and IBM FP2 allow scalar computation on the lower half of 2-way SIMD vector registers. Still, even on SSE2 and FP2, mixing scalar and SIMD requires expensive auxiliary data reordering operations.

Restricted Instruction Level Parallelism. Short vector SIMD extensions specify parallelism on the level of single instructions, which makes them similar to multiple instruction issue techniques like super-scalar execution or explicit parallel instruction computing. However, as the definition of the particular SIMD instruction set usually imposes severe restrictions on the supported data layouts and SIMD operations, their utilization is not only a scheduling problem, but also an instruction selection problem.

3.2.2 SIMD Software Support

Currently, application developers have three common methods for accessing the SIMD instructions of a general-purpose microprocessor: (*i*) They can invoke vendor-supplied libraries that utilize the new instructions, (*ii*) rewrite key portions of the application in assembly language using SIMD instructions, or (*iii*) code in a high-level language and use vendor-supplied macros that make available the extended functionality through a simple function-call-like interface.

System Libraries. The simplest approach to improving application performance is to rewrite the system libraries to employ the SIMD hardware features. The clear advantage of this approach is that existing applications can immediately take advantage of the new hardware without recompilation. However, the restriction of SIMD instruction utilization to the system libraries also limits potential performance benefits. An application's performance will not improve unless it invokes the appropriate system libraries, and the overheads inherent in the general interfaces associated with system functions will limit application performance improvements. Even so, this is the easiest approach for a system vendor, and vendors have announced or plan to provide such enhanced libraries.

Assembly Language. At the other end of the programming spectrum, an application developer can benefit from SIMD instructions by rewriting key portions of an application in assembly language. Though this approach gives a developer great flexibility, it is generally tedious and error prone. In addition, it does not guarantee a performance improvement over code produced by optimizing compilers, given the complexity of today's microarchitectures.

Programming Language Abstractions. Recognizing the tedious and difficult nature of assembly coding, most hardware vendors, who have introduced multimedia extensions, provide programming-language abstractions. These give an application developer access to the newly introduced hardware features without having to actually write assembly language code. Typically, this approach results in a function-call-like abstraction that represents one-to-one mapping between a function call and a SIMD instruction.

There are several benefits of this approach. First, the compiler—and not the developer—is responsible for machine-specific optimizations such as register allocation and instruction scheduling. Second, this method integrates SIMD operations directly into the surrounding high-level code without an expensive procedure call to a separate assembly language routine. Third, it provides a high degree of portability by isolating the application from the specifics of the underlying hardware implementation. If the SIMD primitives do not exist in hardware on the particular target machine, the compiler can replace the SIMD macro by a set of equivalent operations.

The most common language extension supplying SIMD primitives is the enhancement of the C programming language by function-call like intrinsics (or built-in) functions and new data types to mirror the instructions and vector registers. For most SIMD extensions, at least one compiler featuring such language extensions exists. Examples include C compilers for HP's MAX-2, Intel's MMX, SSE, and SSE 2, Motorola's AltiVec, and Sun's VIS architecture as well as the Gnu C compiler that supports a broad range of short vector SIMD extensions.

Each intrinsic directly translates into a single SIMD instruction, and the compiler allocates registers and schedules instructions. This approach would be even more attractive to application developers if the industry agreed upon a common set of macros, rather than having a different set from each vendor.

Vectorizing Compilers. While macros may be an acceptably efficient solution for invoking SIMD instructions within a high-level language, subword parallelism could be further exploited with automatic compilation from high-level languages to object code utilizing these instructions. Some vectorizing compilers for short vector SIMD extensions exist, including the Intel C++ compiler, the PGI Fortran compiler and the CodePlay Vector C compiler.

3.2.3 Vector Computers vs. Short Vector SIMD

Vector computers are supercomputers used for large problems in computational science and engineering, as many numerical algorithms allow those parts which consume the majority of computation time to be expressed as vector operations. This holds especially for almost all linear algebra algorithms [35, 19]. It is therefore a straightforward strategy to improve the performance of processors used for numerical data processing by providing an instruction set tailor-made for vector operations as well as suitable hardware.

This idea materialized in vector architectures comprising specific *vector instructions*, which allow for componentwise addition, multiplication and/or division of vectors as well as the multiplication of a vector's components by a scalar. Moreover, there are specific load and store instructions enabling the processor to move all components of a vector to or from main memory.

The hardware counterparts of vector instructions are the respective *vector registers* and *vector units*. Vector registers are memory elements which are able to hold vectors of a given maximum length. Vector units performing vector operations, as mentioned above, usually require the operands to be stored in vector registers.

Vector computers are specialized machines not comparable to general purpose processors featuring short vector SIMD extensions. The most obvious differences on the vector extension level are the larger vector lengths, the support for operating on smaller portions of vectors, and non-unit stride memory access. In vector computers actually multiple processing elements are processing vector data, while in short vector SIMD extensions only a very short fixed vector length is supported. Also, an increasing number of SIMD extensions—like Intel SSE3 and AMD 3DNow!—do not only offer the inter-vector parallelism style instructions found in vector supercomputers, but also intra-vector parallelism style instructions, which are particularly useful for reducing the overhead associated with data reordering within vector registers.

Example (Vector Computers) The Cray T90 multiprocessor uses Cray Research Inc. custom silicon CPUs with a clock speed of 440 MHz, where each processor has a peak performance of 1.7 Gflop/s. On any processor there are 8 vector registers with 128 words (vector elements) of eight bytes (64 bits) each.

Current vector computers provided by NEC range from deskside systems (the NEC SX-6i featuring one CPU and a peak performance of 8 Gflop/s) up to one of the most powerful computers in the world: the *Earth Simulator* featuring 5120 vector CPUs running at 500 MHz, thus reaching a theoretical peak performance of 41 Tflop/s.

The high performance of floating-point operations in vector units is mainly due to the concurrent execution of operations (as in a very deep pipeline).

There are further advantages of vector processors as compared with other processors capable of executing overlayed floating-point operations.

- As vector components are usually stored contiguously in memory, the access pattern to the data storage is known to be linear. Vector processors exploit this fact using a very fast vector data fetch from a massively interleaved main memory space.

- There are no memory delays for a vector operand that fits completely into a vector register.

- There are no delays due to branch conditions as they might occur if the vector operation were implemented in a loop.

In addition, vector processors may utilize the super-scalar principle by executing several vector operations per time unit [20].

Parallel Vector Computers

Most of the vector supercomputer manufacturers produce multiprocessor systems based on their vector processors. Since a single node is so expensive and so finely tuned to memory bandwidth and other architectural parameters, the multiprocessor configurations have only a few vector processing nodes.

Example (Parallel Vector Computers) A NEC SX-5 multi node configuration can include up to 32 SX-5 single node systems for the SX-6A configuration.

However, the latest vector processors fit onto single chips. For instance, NEC's SX-6 nodes can be combined to form much larger systems in multiframe configuration where up to 1024 CPUs are combined. In the Earth Simulator, even 5120 CPUs are integrated into a single system, comprising 640 shared memory processor nodes, connected by a 640-by-640 single-stage non-blocking crossbar switch.

Vectorizing Compilers

Vectorizing compilers were developed for the vector computers described above [91, 89]. Using vectorizing compilers to produce *short vector SIMD* code for discrete linear transforms in the context of adaptive algorithms is not straightforward. As the vectorizing compiler technology originates from completely different machines and in the short vector SIMD extensions other and new restrictions are found, the capabilities of these compilers are limited. Especially automatic performance tuning poses additional challenges to vectorizing compilers as the codes are generated automatically and intelligent search is used which conflicts with some compiler optimization. Thus compiler vectorization and automatic performance tuning cannot be combined easily. The two leading adaptive software systems for discrete linear transforms cannot directly use compiler vectorization in their code generation and adaptation process.

FFTW. Due to the recursive structure of FFTW and the fact that memory access patterns are not known in advance, vectorizing compilers cannot prove alignment and unit stride properties required for vectorization. Thus FFTW cannot be vectorized automatically using compiler vectorization.

SPIRAL. The structure of code generated by SPIRAL implies that such code cannot be vectorized directly by using vectorizing compilers without some hints and changes in the generated code. A further difficulty is introduced by optimizations carried out by SPIRAL. Vectorizing compilers only vectorize rather large loops, as in the general case the additional cost for prologue and epilogue has to be amortized by the vectorized loop. Vectorizing compilers require hints about which loop to vectorize and to prove loop carried data dependencies. It is required to guarantee the proper alignment. The requirement of a large number of loop iterations conflicts with the optimal code structure, as in discrete linear transforms a small number—sometimes as small as the vector length of the SIMD extension—turns out to be most efficient. In addition, straight line codes cannot be vectorized satisfactorily by available compilers.

Vector Computer Libraries

Traditional vector processors have typically vector lengths of 64 and more elements. They are able to load vectors at non-unit stride but feature a rather high startup cost for vector operations [45]. Codes developed for such machines do not match the requirements of modern short vector SIMD extensions. Highly efficient implementations for DFT computation that are portable across different conventional vector computers are not available. For instance, high-performance implementations for Cray machines were optimized using assembly language [46]. An example for such a library is Cray's proprietary SCILIB [53].

3.3 2-Way SIMD Basics

This section presents 2-way SIMD arithmetic and data reordering operations.

3.3.1 Arithmetic SIMD Operations

Traditionally, SIMD ISAs only supported the classical "vertical" SIMD style, defining vector operations as a number of parallel point-wise scalar operations.

However, with short vector SIMD, it also makes sense to support other styles, like "horizontal" style SIMD, which allows reducing (parts of) vectors to scalars packed into a vector. Horizontal SIMD style code can be optimized more easily, as reordering operations like "Swap" can often be folded easily into horizontal SIMD instructions, particularly if the basic scalar operation is commutative.

Fig. 3.2 depicts the two SIMD styles. While all SIMD ISAs offer vertical SIMD, only a few of them (AMD 3DNow! and Intel SSE3) also support a horizontal SIMD style.

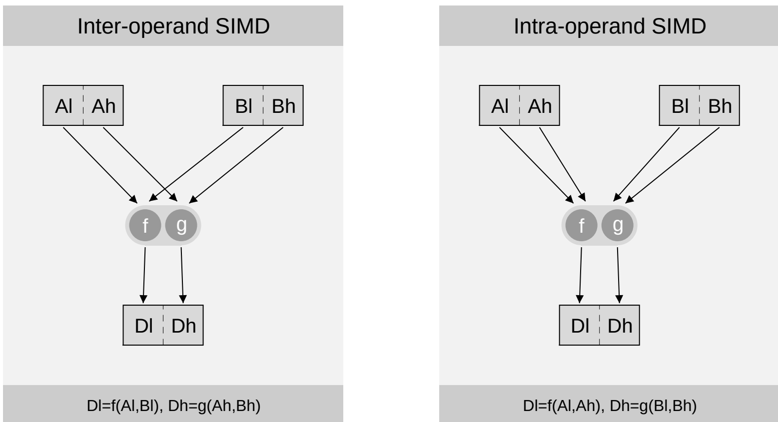

Figure 3.2: 2-way Binary SIMD Operation Layouts. Both common layouts for binary 2-way SIMD operations are shown. On the left, there is a classical inter-operand ("vertical") SIMD style operation that performs two parallel point-wise scalar operations. On the right, there is an intra-operand ("horizontal") SIMD style operation that reduces two vectors to two scalar values, packed into a SIMD vector.

SIMD sign change operations (shown in Fig. 3.3) toggle the sign of one or both parts of a vector. These operations are often needed as auxiliary instructions, particularly for "mixed" SIMD arithmetic instructions performing scalar operations with different signs, combining, for instance, one addition and one subtraction.

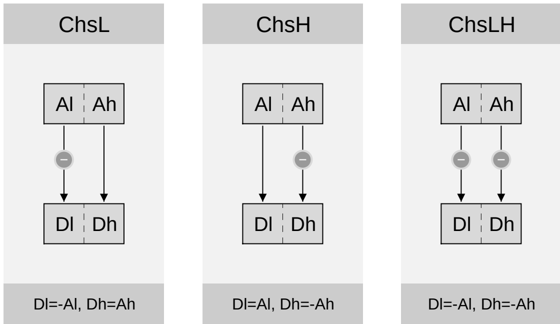

Figure 3.3: 2-way SIMD Sign Change Operations.

Sign change support greatly varies among different SIMD instruction sets. AMD 3DNow! allows performing sign change operations either by doing a multiplication or by using the bitwise integer exclusive-or operator to toggle the sign bit. Intel SSE3 offers mixed parallel SIMD add/sub instructions, which implicitly perform a sign change on one half of the input vector. IBM FP2 offers special SIMD FMA instructions that toggle some particular part of an input vector.

3.3.2 Data Reordering SIMD Operations

Depending on the concrete data formats supported by the actual SIMD arithmetic instructions available on some target architecture, the layout of data residing in vectors may need to be rearranged. Data may be shuffled in memory—using a sequence of partial vector stores followed by vector loads—or in registers. Generally, the in-register method is to be preferred whenever possible.

Fig. 3.4 shows some basic 2-way SIMD reordering operations.

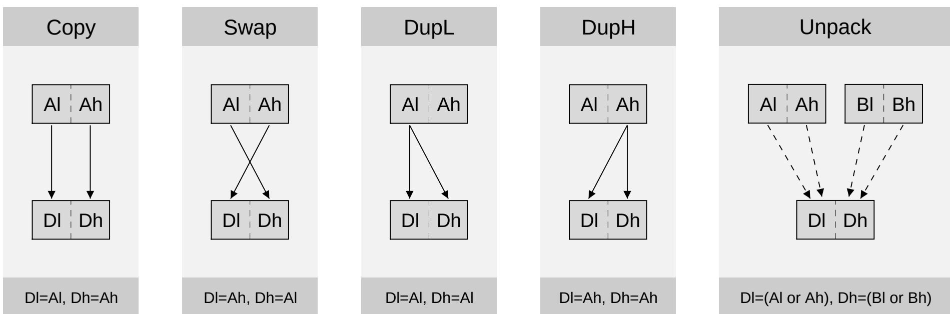

Figure 3.4: Basic 2-way SIMD Reordering Operations. On the left, four unary operations (Copy, Swap, DupL, DupH) are shown. On the right, the binary reordering operation "Unpack" is depicted, which has four different forms. "UnpackLL" packs the lower parts of two vectors, "UnpackHH" the higher part of two vectors, "UnpackLH" and "UnpackHL" the lower part of one vector and the higher part of the other one.

Arbitrary reordering two SIMD vectors of length 2 can be done by using the binary 2-way SIMD instruction “Unpack”. Common SIMD ISA extensions like SSE2/SSE3 and 3DNow! directly support “UnpackLL” and “UnpackHH”, easing the efficient in-register transposition of auxiliary 2×2 matrices. IBM FP2, on the other hand, offers extensive support for *implicitly performed* SIMD reordering operations by supplying a large variety of SIMD FMA instructions, but lacks efficient support for these operations in the general case.

“UnpackLH” and “UnpackHL” mix the lower part of one vector register with the higher part of another, combining a “Swap” operation with a dependent “UnpackLL” or “UnpackHH” instruction. Currently, only the Itanium SIMD ISA offers a reordering instruction of this kind.

3.4 The Blue Gene Processor

Blue Gene Servers. Top-performing supercomputers are usually based on the fastest processors available. However, IBM went a radically different way in their latest hardware development, building Blue Gene servers [60] on a relatively slow embedded-systems processor with low-power consumption, the IBM PowerPC 440.

To efficiently support scientific computing applications, IBM added a functional unit for double-precision scalar and 2-way SIMD floating-point (FP) arithmetic, extending the existing processor design by an auxiliary processor unit, yielding the PowerPC 440 FP2 [10].

One node of a Blue Gene server comprises two PowerPC 440 FP2 processors on one chip (one processor dedicated to computation, the other one to communication), shared memory, and high-speed network interconnect hardware. The biggest installation built to date—BlueGene/L—is made up of the unprecedented number of 65,536 nodes integrated into a single distributed memory system, connected by a 3D torus network. The BlueGene/L system has a theoretical peak performance of 360 Tflop/s, one order of magnitude larger than the Earth Simulator’s performance, the previous leader of the the Top 500 list, which provided 40 Tflop/s peak performance. As of November 2005, Blue Gene servers take three out of the ten top positions on the Top500 supercomputing list, including the number one and two.

This section presents the IBM PowerPC 440 FP2 processor, focusing on the differences between this particular processor and common-place microprocessors used in desktop computers like the Intel Pentium 4 and the AMD Athlon64.

Short Profile. IBM’s Blue Gene processor, the PowerPC 440 FP2, is a low-frequency (700 MHz) 32 bit processor with 32 integer registers, 32 SIMD FP registers, a short (7-stage) pipeline, large split L1 caches (32 kB for instructions, 32 kB for data), a fast non-pipelined multiplier, and support for 2-way super-scalar out-of-order execution.

Table 3.3: Integer Instruction Support. As Intel's x86 is a 2-operand architecture, most integer instructions are destructive, mandating that some particular source register must also be used for storing the output. Thus, instructions like subtraction or shift require additional copying of one of their source operands if the respective operand is to be referenced again.

	x86/AMD64	PowerPC
General Type	2-operand	3-operand
Shift costs	2	1
Sub costs	2	1
Negate costs	2	1
Shift and add instruction	√ (`lea`)	–

Integer Support. In digital signal transform codes that operate on floating-point data, integer instructions are solely used for performing auxiliary tasks like fulfilling the calling convention or calculating effective addresses.

Regarding the integer instruction set, the PowerPC ISA is generally more flexible than the one of the Intel x86, as shown in Table 3.3. On the other hand, the PowerPC ISA does not offer a combined *shift by a constant and add* instruction, which is particularly useful for calculating effective addresses of variably-strided array elements.

Table 3.4: PowerPC 440 FP2 Super-scalar Execution. On the PowerPC 440 FP2, both arithmetic and SIMD data-reordering instructions are assigned to the same functional unit, and cannot be executed in parallel. At most one instruction per cycle may access naturally aligned memory.

Op/Op	Load/Store	Int Arith.	SIMD Arith.	SIMD Reorder
Load/Store	–	√	√	√
Int Arith.	√	√	√	√
SIMD Arith.	√	√	–	–
SIMD Reorder	√	√	–	–

Execution Limitations. Although the PowerPC 440 FP2 is a dual-issue design, not all conceivable pairs of instructions may be executed in parallel (Table 3.4). Accordingly, this processor is significantly less powerful than full-blown processors available in desktop machines, all of which allow the parallel execution of SIMD arithmetic and SIMD reordering instructions.

Arithmetic Support. The PowerPC 440 FP2 supports scalar [73] and 2-way SIMD [17] floating point arithmetic, both operating on the same 2-way SIMD register file, with scalar instructions working on the lower half of SIMD registers. Support for single-precision floating-point data is offered only for data loads/stores and explicit rounding operations, but not for most pipelined arithmetic operations. Both for the scalar and the SIMD case, FMAs are available,

Table 3.5: Comparison of SIMD ISA Extensions. Among different SIMD ISAs there are considerable differences with regard to (*i*) the number of available logical vector registers, (*ii*) partial vector memory access support, (*iii*) addressing mode support, (*iv*) vector swap costs, and (*v*) FMA support. SSE2 and SSE3 offer 8 logical registers when used in legacy 32 bit mode, and 16 registers when operated in 64 bit mode. Explicit copying, i. e., non-destructive, SIMD swap instructions are only available with AMD 3DNow! and IBM FP2, but not with Intel SSE2/SSE3. IBM FP2 additionally offers a large number of SIMD FMAs that implicitly swap one particular input operand.

	3DNow!	Ext. 3DNow!	SSE2	SSE3	**FP2**
Number of registers	8	8	8 (16)	8 (16)	32
LoadLo/StoreLo	√	√	√	√	√
LoadHi/StoreHi	–	–	√	√	√
Load0	√	√	–	–	–
Full Addr. Modes	√	√	√	√	–
Copying swap costs	2	1	2	2	0 (1)
FMA support	–	–	–	–	√

which doubles the theoretical peak performance and improves the accuracy of the results by avoiding intermediate rounding. Table 3.5 compares a selection of SIMD instruction sets with the FP2 SIMD ISA.

Table 3.6: Comparison of Addressing Modes. Intel x86 and AMD64 (used with 3DNow! and SSE2/SSE3) are CISC-style instruction sets offering many different addressing modes. The PowerPC is an enhanced RISC-style architecture and offers "update" addressing modes, which are particularly useful for accessing continuous memory locations. It is notable that the PowerPC 440 FP2 offers a lot less addressing modes operating on SIMD vectors than on scalars.

Mode	3DNow!	SSE2/SSE3	PowerPC	**FP2**
reg	√	√	√	√
reg+reg	√	√	√	√
reg+reg update	–	–	√	√
reg+reg*k, $k \in \{2, 4, 8\}$	√	√	–	–
reg+const	√	√	√	–
reg+const update	–	–	√	–

Addressing Modes. The PowerPC 440 FP2 offers quite different addressing modes than other general purpose microprocessors, particularly when operating on SIMD vectors. Table 3.6 compares the addressing modes offered by Intel x86 compatible machines with the IBM FP2.

The IBM PowerPC 440 FP2 processor has DSP-like addressing mode limitations for SIMD loads/stores, and may thus require considerable amount of additional integer instructions for address calculation. Minimizing the number of these extra instructions poses a challenge, not to be encountered when generating code for general purpose processors. These limitations particularly affect both constant accesses and stack accesses, which would normally use the "reg+const" addressing mode.

Experiments have shown that the code needed for effective addresses calculation often has a significant negative performance impact, in particular in cases with a high ratio of the memory access count compared to the number of arithmetic operations. All fast linear signal transform algorithms have this property.

Procedure Calling Convention. The application binary interface (ABI) used in the Blue Gene environment [40] defines approximately half the registers as callee-saved, which can be a considerable disadvantage for small leaf procedures.

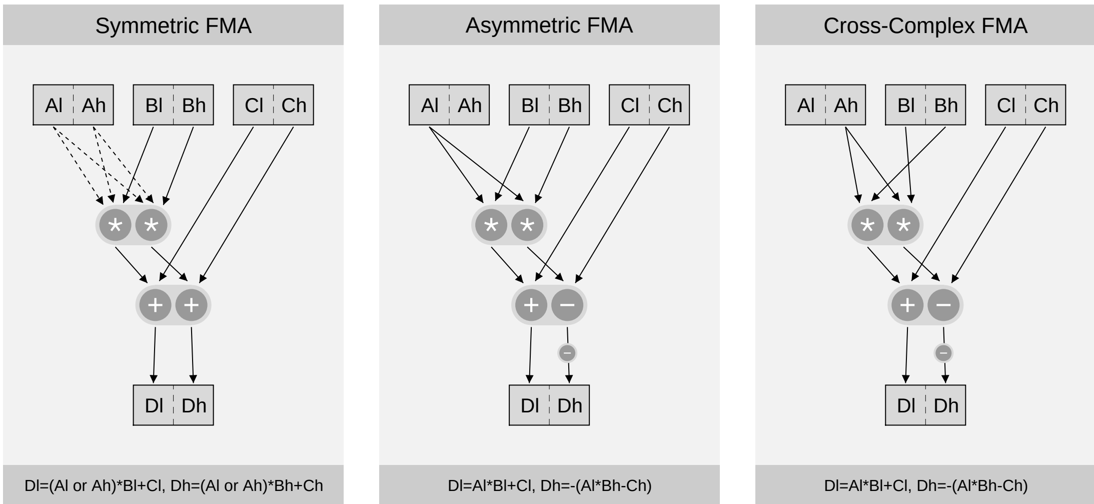

Figure 3.5: 2-way SIMD FMA Instructions on Blue Gene Systems.

Vertical SIMD. Unlike other SIMD ISAs, FP2 offers a huge collection of vertical SIMD FMAs (Fig. 3.5). IBM has coined the term SIMOMD (single-instruction multiple-operations multiple-data) to describe the FP2 instruction set, stressing that some of these instructions allow performing different operations, e. g., one addition and one subtraction, on different parts of the SIMD vector registers.

In total, the PowerPC 440 FP2 offers 24 different SIMD FMA instructions. 16 symmetric FMAs are strictly parallel for operands B and C, but allow any of the unary SIMD reordering operations (“Copy”, “Swap”, “DupL”, or “DupH”) of Fig. 3.4 to be performed on input A. Symmetric FMAs offer all four FMA variants, i. e., multiply-add, multiply-sub, negative multiply-add, and negative multiply-sub. All symmetric FMAs perform two scalar multiplications and two additions, or two multiplications and two subtractions. Four asymmetric FMAs allow performing one addition and one subtraction within the FMA. Four cross-complex FMAs additionally swap the input operand B.

Other SIMD ISAs, like Intel's SSE2/SSE3 or AMD's 3DNow!, do not support SIMD FMAs at all. The Intel Itanium supports four symmetric FMAs.

Horizontal SIMD. Native support for horizontal SIMD is completely missing in IBM FP2. Emulating horizontal SIMD operations with a sequence of vertical operations and data reordering operations is considerably more expensive than on other SIMD ISAs (Tables 3.7 and 3.8).

Table 3.7: Instruction Count for Horizontal (H) and Vertical (V) SIMD Addition and Subtraction Operations. Uniform instructions perform two additions or two subtractions, while mixed instructions perform an addition and a subtraction.

op	3DNow!	Ext. 3DNow!	SSE2	SSE3	**FP2**
H / uniform	1	1	3	1	5
H / mixed	2	1	4	2	5
V / uniform	1	1	1	1	1
V / mixed	2	2	2	1	1

Arithmetic Costs of SIMD Instructions. The costs for horizontal and vertical SIMD instructions differs considerably among different SIMD instruction sets. While all SIMD ISAs satisfactorily support the classical vertical SIMD style, horizontal SIMD style support greatly varies. FP2 completely lacks horizontal style SIMD instructions. AMD 3DNow!, AMD extended 3DNow!, and Intel SSE3 offer different sets of horizontal SIMD instructions. Table 3.7 lists the SIMD operation costs for different SIMD instruction sets.

Table 3.8: Instruction Count for SIMD Reordering Operations. Uniform unpacks ("UnpackLL" and "UnpackHH") combine the lower parts of two registers, while mixed unpacks ("UnpackLH" and "UnpackHL") combine the lower part of one register with the upper part of another one.

op	3DNow!	Ext. 3DNow!	SSE2	SSE3	**FP2**
UnpackLL, UnpackHH	1	1	1	1	2
UnpackLH, UnpackHL	2	2	2	2	2

SIMD Reordering Support. With all its feature wealth, FP2 has a startling lack of support for common SIMD reordering instructions (Table 3.8). Also, as the actually supported reordering instructions offered by FP2 cannot be executed in parallel with any computation, the costs of SIMD reordering are a main challenge with FP2.

Chapter 4

The MAP 2-way SIMD Vectorizer

In the past twenty years, hardware development produced many different techniques for improving the peak performance of general purpose processors. Because of the multifaceted relations between hardware and software with regard to achieving a satisfactory degree of efficiency, hardware development has had increasingly strong implications for (*i*) the development of application software, (*ii*) the evolution of compilation tools and libraries, and (*iii*) the maintenance and adaptation of existing legacy software.

Many processor hardware techniques introduced in the 1980s and the 1990s—like instruction pipelining, *reduced instruction set computing* (RISC), and *out-of-order* (OoO) execution—reasonably compromise between raising peak performance and holding high efficiency, which is especially important for existing software. These hardware features are handled satisfactorily by all modern high-level compilers, by applying techniques like list-based instruction scheduling and optimal instruction selection for expression trees.

However, there are other performance-related hardware features, like multi-level caches, which are not handled well by high-level general purpose compilers. As a result dramatic efficiency losses are to be observed with many existing codes. Thus, in the development of performance critical software, the responsibility of tuning programs to fit the peculiar properties of the memory hierarchy was left to application and library programmers. While producing good results in some cases, this manual tuning process was both tedious and error-prone, which—in the field of numerical libraries—eventually led to the rising of the new paradigm of automatic performance tuning.

Efficiency issues got even more critical than they already were since the introduction of multi-level caches in the late 1990s when *single-instruction multiple-data* (SIMD) extensions entered the mass market. The first SIMD extensions exclusively worked with integer vectors, but soon newer extensions emerged that also supported SIMD floating-point operations—finally making short vector SIMD suitable for numerical computation. Nowadays, SIMD extensions are commonplace on general purpose processors, used from embedded computing, to desktop machines, to supercomputers.

SIMD extensions feature instructions operating on vectors instead of scalar values, offering a much more restricted form of instruction-level parallelism than *explicitly parallel instruction computing* (EPIC) processors. Unlike the original vector instructions provided by vector supercomputers, short vector SIMD extensions in contemporary general purpose processors operate on relatively small fixed-length vectors. SIMD extensions allow for a larger number of SIMD layouts, not just the "classical" vertical (parallel point-wise) SIMD style, but also a horizontal (reduction) SIMD style.

SIMD extensions have the potential for significant performance improvements. However, when left unused, efficiency drops by 50% with 2-way SIMD, and even by 75% with 4-way SIMD. Therefore, SIMD vectorization is an important prerequisite for achieving high efficiency on modern processors (see Section 3.2).

Currently available methods for producing programs that are able to utilize short vector SIMD instructions can be categorized according to the following criteria.

Interfaces for Using SIMD Extensions. Three kinds of interfaces providing programmers with an access to SIMD features have become wide-spread. (*i*) Portable programs written in some high-level language in tandem with vectorizing compilers, possibly requiring hints (pragmas) on how to vectorize, (*ii*) proprietary, non-portable language extensions providing explicit access to short vector SIMD extensions on source-code level, and (*iii*) inherently non-portable programs written in assembly language.

Responsibility for Vectorization. The actual vectorization process can be done explicitly by application or library programmers. Contrary to this approach is the use of vectorizing compilers (which may or may not be hinted by programmers) to extract parallelism from portable programs. As a third option, program generators may generate innately vectorized codes.

Focus of Vectorization. Vectorization methods extract a restricted form of parallelism—suitable for the respective SIMD instruction set extension—either from independent loop iterations (*loop level vectorization*), or straight line code (*basic block vectorization*). Depending on the code to be vectorized, one method or the other may be more suitable. In some cases, it may be even advantageous to use a combination of both methods.

Well-established vectorization methods developed for classical vector computers [91] operate on the loop level. In short, these methods (*i*) assert that the data is properly aligned and that there are no side-effects in the loop considered for vectorization, (*ii*) analyze data dependencies to identify data parallelism of loop iterations, (*iii*) join a fixed number of loop iterations to one equivalent piece of code utilizing SIMD vector instructions, (*iv*) emit the vectorized loop and a scalar loop used to handle any remaining iterations.

Loop-based vectorization ignores any kind of parallelism already present in basic blocks, i. e., single entry, single exit sequences of primitive instructions. Unlike SIMD-style vector computers, SIMD floating-point instruction-set extensions on general purpose processors operate on very short vectors. As this allows expressing parallelism on a very low level, not only loop-based vectorization techniques, but also more fine-grained ones, that extract the parallelism already present within a basic block, can be utilized. Basic block based vectorization tries to extract the parallelism already present *within* a block, trying to maximally cover a scalar DAG with SIMD instructions natively supported by the target machine [54, 55].

Generality of Approach. Some vectorization approaches are general purpose by their nature, for instance, as they are applicable to any program that features loops or to any basic block. Other methods depend on the particular internal structure of some class of algorithms, for instance, fast signal transform algorithms like complex FFTs.

Any of these approaches involves tradeoffs between portability and generality on the one hand and achievable performance on the other one.

To produce high-quality numerical code to be used in signal transform libraries, the Vienna MAP vectorizer [51, 52, 56] has been developed. This vectorizer automatically extracts 2-way SIMD parallelism out of given numerical straight-line code, and has been applied successfully to straight-line code produced by Fftw, Spiral, and Atlas—automatically vectorizing a large variety of numerical codes ranging from FFTs and other DSP transforms to Blas kernels.

This chapter introduces the most recent version of the MAP vectorizer, its features, its limitations, as well its novelties and modifications in comparison to previous work.

Related Work

The three major ways of utilizing SIMD extensions for accelerating signal processing codes—(*i*) hand-coding, (*ii*) the use of general purpose vectorizing compilers, and (*iii*) the adaptation of domain-specific code generators—are reflected by several publications.

Hand-coding. There are well-known hand-coded vector FFT algorithms like Stockham's FFT algorithm [80, 45] as well as vector computer libraries like the Sciport library [53]. However, without further adaptation to the requirements of deep memory hierarchies, these algorithms lead to disappointingly low performance on SIMD architectures present in common-place processors.

A general purpose hand-coding method utilizing instruction-level parallelism is to implement a given algorithm by using the complex arithmetic of C99 [4] and let an appropriate compiler map the complex operations to sequences of 2-way SIMD vector instructions. However, code generators like Fftw's `genfft` and Spiral's code generator usually map signal transform code to real arithmetic, as this allows further reducing the instruction count, by performing sophisticated optimization on real arithmetic [29].

A standard approach to exploiting an algorithm's intrinsic parallelism is to use a blended mixture of the ad-hoc utilization of the instruction-level parallelism inherent in a given program and the hand-vectorization of loops. In this approach, the programmer formulates the parallelism within the algorithm by using proprietary language extensions. As a negative result, the produced codes are incompatible across different SIMD extensions.

In the field of DSP transforms, these hand-coding approaches are used in SIMD-enabled vendor libraries (examples include Intel's MKL and IPP [43], Apple's vDSP [5] and vBigDSP [14], as well as AMD's core math library ACML [1]), application notes (Intel's split radix FFT [41]), and free implementations like the NEC V80R FFT [64] or the Linux SIMD library libSIMD [65]. SIMD-vectorized wavelet transforms are presented in [11] and a SIMD-vectorized FFT library is presented in [72].

The automatic performance tuning system Atlas allows for the insertion of hand-coded kernels featuring SIMD instructions into its optimization cycle. Atlas depends on programmers contributing such hand-coded kernels for new architectures [87]. To get the maximum performance, the respective kernel routines are typically coded in assembly language.

Vectorizing Compilers. There are many research and production-quality compilers for SIMD extensions, including Intel's C++ compiler [42], IBM's XL C compiler for Blue Gene [3], a vectorizing extension to the SUIF compiler [77], CodePlay's Vector C compiler [12], and the SWAR compiler `scc` [23, 24].

Automatic general-purpose loop-vectorizing compiler technology originates from vector computer research and is included in most vectorizing compilers [89]. These algorithms were designed for long vector lengths and other characteristics of conventional vector computers, like constant non-unit stride memory access. Some of these implicit assumptions simply do not hold on machines having short vector SIMD extensions. In addition, vectorization- and locality-enhancing loop transformations often have conflicting goals. Compilers therefore require user-supplied hints—for instance, by compiler directives called pragmas—in order to successfully vectorize loops [27].

Automatic general purpose methods based on the extraction of instruction-level parallelism in basic blocks are used in many compilers (e. g., Intel C, Codeplay VectorC, and IBM XL C compilers). These projects originate from EPIC research [54, 23], and their vectorization algorithms search for code sub-blocks that feature parallelism. In order to map the full computation, these parallel blocks have to be connected, either by scalar operations or by data shuffling operations, which can introduce considerable vectorization overhead. Due to an exploding search space and to a lack of knowledge about the code's intrinsic structure, these algorithms fail on large basic blocks having complicated structure. Experiments show that for signal transform codes, these vectorization approaches produce only negligible speed-up, and in most cases even slow down the code.

A graph-based *code selection* technique for DSPs with SIMD support has been introduced in [55]. Techniques for SIMD utilization in the context of energy-aware compilation for DSPs are presented in [58].

Methods Used in Domain-Specific Code Generators. The latest versions of Spiral include generic SIMD vectorization facilities [26], implemented by the addition of new SIMD-specific rules to Spiral's formula manipulation system. These new rules [69] exploit specific algebraic identities of tensor products to transform into a form suitable for extracting SIMD-style parallelism. Focusing on the extraction of vertical-style SIMD operations, this extension of Spiral uniformly supports (*i*) all currently available classical 2-way and 4-way SIMD floating-point instruction sets, and also (*ii*) 8-way 16-bit fixed-point and 16-way 8-bit fixed-point SIMD integer instruction sets.

The code generator of Fftw 3, the most recent version of Fftw, includes instruction-level vectorization for 2-way SIMD vector extensions that is based on properties of complex FFTs and utilizes C language extensions. For 4-way SIMD vector extensions, Fftw 3 applies complex FFT specific methods in combination with 2-way loop vectorization. For AMD Athlon processors, which feature the 2-way 3DNow! floating-point SIMD instruction-set extension, Fftw 3 includes a predecessor of the MAP tool-chain, the code generator of Fftw-Gel [50], developed in the course of 2001. Fftw-Gel's code generator has been tightly integrated with Fftw's `genfft` code generator, and is part of the main distribution of Fftw 3.

Direct Comparison

The approach taken by the MAP vectorizer bears some similarity with previously existing work for vectorizing DSP compilers [54, 55, 23]. The MAP vectorizer is strongly biased towards different assumptions—both about the class of input codes and about particular properties of the target hardware.

(*i*) As linear transform codes are highly structured, any divide-and-conquer based vectorization approach incurs high costs when connecting vectorized subgraphs.

(*ii*) Unlike SIMD instruction-sets of some DSPs, SIMD instruction set extensions of general purpose processors do not allow scalar operations and SIMD operations working on the same data sets to be mixed efficiently. Therefore, MAP's vectorization mandates that *all* computation is performed by SIMD instructions, while attempting to keep the SIMD reordering overhead reasonably small.

(*iii*) Numerical kernels used in automatic performance tuning systems can be very large, thus finding a compromise between vectorization runtime and code quality is a key issue.

(*iv*) The interleaved complex array data layout used with most complex FFT kernels massively constrains the size of the search space to be traversed by vectorization. For the vectorization of certain categories of codes, e. g., real FFT kernels, the vectorizer cannot prune the search space in a similar way, as it possibly has to consider all possible pairs of inputs and outputs.

The vectorization approaches taken by the SIMD-version of Spiral and by the MAP vectorizer are somewhat orthogonal. Spiral operates in a top-down way, explicitly represents knowledge about the domain in its proprietary signal processing language SPL, and has a relatively narrow set of SIMD instructions. Contrary to this, the MAP vectorizer operates bottom-up by synthesizing blocks from individual instructions, implicitly encodes knowledge about the domain within the vectorization algorithm, and uses as many SIMD instructions as are available on the particular target architecture.

An experimental comparison of these two approaches [27] on AMD Athlon systems suggested a slight advantage of the bottom-up approach taken by the MAP vectorizer, due to fact that the MAP vectorizer extracts horizontal-style SIMD code, which can be improved more easily than the vertical-style SIMD code that Spiral finds. However, MAP's vectorization technique is restricted to 2-way SIMD, whereas Spiral's is not.

4.1 Basic Properties

The MAP vectorizer extracts 2-way SIMD parallelism out of basic blocks, aiming at a reduction of the overall instruction count. In the best case, the number of SIMD instructions output is exactly 50% the number of scalar instructions.

Global Structure. SIMD vectorization is a source-to-source code transformation that—unlike local approaches like peephole optimization—changes the global structure of the input code. The vectorizer does not directly deal with any local optimizations like extracting SIMD fused multiply-add instructions or eliminating SIMD swap instructions, but relies on the subsequent parts of the tool chain, most importantly the MAP peephole optimizer, to perform these tasks.

Data Layouts. All commonly used layouts for data arrays, including interleaved and split data formats for arrays of complex numbers, are supported.

Supported SIMD Operations. Linear transform codes and many codes in linear algebra utilize only a small set of basic operations for memory access and floating-point calculations: (*i*) Loads from and stores to some array, (*ii*) loads of some numerical constants, and (*iii*) floating-point addition, subtraction, and multiplication operations. This allowed to restrict the selection of supported SIMD operations to cover exactly these cases, keeping the vectorizer as simple as possible.

4.2 Implementation Basics

Aiming at covering scalar DAGs with SIMD instructions actually supported by the SIMD target architecture, the MAP vectorizer maps (*i*) scalar variables to

SIMD variables, (*ii*) scalar constants to SIMD constants, and (*iii*) scalar instructions to SIMD instructions.

To accomplish this goal, the vectorizer alternately performs the following two steps, either until the scalar DAG is covered, or any attempts have proven futile. (*i*) It picks two scalar variables that have not yet been combined to form a SIMD variable. (*ii*) It picks two scalar instructions, whose destination operands have both been already combined earlier into one SIMD variable, and emits the corresponding SIMD pseudo-instruction.

There are many degrees of freedom in selecting the next scalar variables to be combined and in choosing *how* to combine these two scalar instructions. Some of these possibilities may only represent a local solution that is not part of any global solution. But one particular local solution may just as well be part of some global solutions. This type of non-determinism is usually refered to as *don't know non-determinism* [79], because not all decisions taken in the course of action turn out to lead to some global solution, as it is the case with *don't care non-determinism.* To explore all possibilities in a systematic way, the search process handles non-deterministic choice by using *depth-first search* (DFS) with chronological backtracking [84, 34], the basic built-in execution mechanism of the logic programming languages Prolog and Mercury.

Extraction of SIMD Variables. The MAP vectorizer represents 2-way SIMD variables as unordered pairs of two distinct scalar variables. Using unordered pairs—as opposed to using ordered pairs—for the representation of SIMD variables eased the implementation of the vectorizer.

The MAP vectorizer does not insert actual SIMD swap instructions into the vectorized code, until vectorization has succeeded and committed to one particular global solution. An auxiliary SIMD swap instruction is required whenever a pair (x, y) is written by some SIMD instruction and the mirrored pair (y, x) is read by some other SIMD instruction. Within the MAP tool chain, it is the responsibility of the peephole optimizer—and not of the vectorizer—to minimize the number of auxiliary swap instructions.

Whenever the vectorizer attempts to combine two scalar variables to one SIMD variable, it asserts that the respective scalar variables do not belong to any other unordered pair representing a SIMD variable. If these two particular scalar variables were previously unbound, the extraction of a SIMD pseudo-instruction writing the respective SIMD variable is triggered, which combines the two scalar instructions producing the respective scalar variables.

Extraction of SIMD Constants. Similar to the case of scalar variables, the vectorizer combines scalar constants to SIMD constants. However, arbitrary replication of scalar constants is allowed, i. e., one particular scalar constant may occur in more than one SIMD constant. As a consequence, vectorization may increase the number of constants used from n scalar constants to n^2 SIMD constants in the worst case. As this large number of constants may have a negative impact on performance, the vectorizer allows (*i*) imposing a limit on the number of SIMD constants and (*ii*) minimizing the number of SIMD constants used.

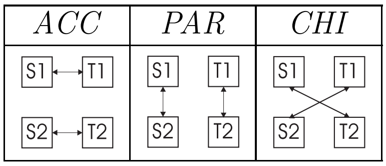

Figure 4.1: SIMD Binding Types. Two scalar binary instructions, `(op1,S1,T1,D1)` and `(op2,S2,T2,D2)`, can be combined in one of three different ways. Intra-operand SIMD instructions are supported by the *ACC* type, whereas the types *PAR* and *CHI* are used for inter-operand SIMD instructions. Because of the features offered by the targeted SIMD instruction sets, the vectorizer uses horizontal style (*ACC*) exclusively for additions and subtractions. The vertical style (*PAR* or *CHI*) is used for additions, subtractions, and multiplications.

Extraction of SIMD Instructions. Not all conceivable pairs of scalar instruction types can be combined, as some of them are not supported efficiently by common-place SIMD instruction sets. When two scalar instructions are combined into one SIMD instruction, the vectorizer asserts that the two scalar instructions are compatible with each other, i. e., they can be mapped to exactly one SIMD pseudo-instruction.

Pairs of scalar instructions that may be combined into one SIMD pseudo-instruction include (*i*) two load instructions, (*ii*) two store instructions, (*iii*) two additions, (*iv*) two subtractions, (*v*) one addition and one subtraction, and (*vi*) two multiplications. For load and store instructions, additional constraints may be imposed, e. g., to enforce the use of consecutive SIMD memory operations for accessing interleaved complex array data.

Combining two given scalar instructions can often be done in different ways (see Fig. 4.1), resulting in different SIMD pseudo-instructions being extracted, as shown in Fig. 4.2.

Traversal of the Scalar DAG. The extraction steps mentioned above—combining scalar variables, scalar constants, and scalar instructions to their SIMD counterparts—are iterated until either the scalar DAG is fully covered with SIMD instructions, or failure is recognized.

Domain-Specific Assumptions. Several restrictions were taken into account in the design of the vectorizer, to allow for an instruction-based "bottom-up" vectorization approach based on exhaustive search, even for relatively large codes. (*i*) A scalar variable is—for all of its uses—part of exactly one SIMD variable. (*ii*) All SIMD data reordering occurs either implicitly by using horizontal or vertical SIMD instructions or explicitly by using SIMD swap instructions.

On one hand these restrictions reduce the class of input codes the vectorizer can handle, but—more importantly—on the other hand they tremendously reduce the size of the search space to be traversed during vectorization.

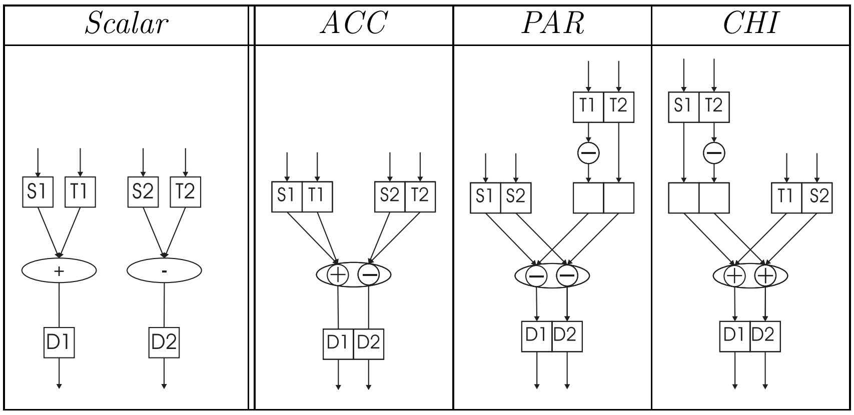

Figure 4.2: Vectorization Alternatives. Two scalar instructions, one addition and one subtraction, are transformed into an equivalent sequence of SIMD instructions in three different ways. Depending on the features supported by the actual target instruction set, one method may be preferable to the other ones. On instructions sets supporting both vertical and horizontal style SIMD instructions like the ones present in AMD 3DNow!, the *ACC* style gives the best results, as it facilitates the peephole optimization of the resulting code. For Blue Gene systems, vertical style vectorization (*PAR* or *CHI*) is to be preferred, as shown in Table 3.7 on page 43.

4.3 Implementation Details

While the basic concepts of the vectorizer have remained largely unchanged from the first working prototypes developed in late 2000 until the present day, the new version of the MAP vectorizer, developed in the context of this thesis, departs from previous work [51, 52] in a number of ways. Design goals of the new vectorizer were (*i*) runtime reductions of the vectorization process, (*ii*) quality enhancements of the SIMD vectorized code, and (*iii*) enhanced adaptability of the vectorizer to new SIMD target architectures.

Modes of Vectorization. The vectorizer operates in two different modes.

Full vectorization tries to find an optimal cover, guaranteeing that $2n$ scalar instructions are mapped to *exactly* n SIMD pseudo-instructions, which also inhibits any duplicate calculation due to SIMD vectorization. All input codes exclusively consisting of complex arithmetic can be handled using this method.

Semi vectorization allows for sub-optimal SIMD utilization, thus enabling a larger class of programs to be handled, e. g., real FFT codes. In semi vectorization mode, the MAP vectorizer lifts two rigid restrictions, allowing for SIMD pseudo-instructions having only 50% SIMD utilization. Firstly, it permits to perform multiplications by constants on one half of a SIMD vector, while leaving the other half unchanged. Secondly, it supports a selection of SIMD pseudo-instructions that can only be translated to a sequence of two or more natively supported SIMD instructions, e. g., performing a combination of one scalar load instruction and one scalar addition or subtraction.

Clearly, full vectorization is used whenever possible. If all vectorization attempts in that preferred mode fail, the vectorizer falls back to the sub-optimal vectorization method.

Preparation Passes. Before the actual vectorization process is started, several preparatory passes are performed to constrain the search space from the very beginning. This goal is achieved by filtering out pairs of scalar variables that cannot possibly occur as parts of any solution of the vectorization process.

Effects of Different Graph Traversal Orders. The order in which the nodes of the scalar DAG are traversed has a profound impact both on the solution order and on vectorization runtime.

Prior versions of the vectorizer [56] always started at the outputs of the DAG, i. e., store instructions, traversing the DAG in a bottom-up fashion. This method worked fine for codes consisting of complex arithmetic only, but it had its shortcomings for codes consisting of real arithmetic.

To improve on this, the current version of the MAP vectorizer adds two new traversal methods. First, it allows traversal both in bottom-up as well as top-down style. Secondly, it borrows the idea of *domain variables* from constraint programming [81]. Domain variables allow the vectorizer to keep track of all pairs of scalar variables that may be formed in the future. When traversing the scalar graph, the scalar variable that occurs in the smallest number of pairs, is picked as the next point in search space to be visited (*first-fail principle*).

The combination of these three traversal methods enable the vectorizer to find the optimal vectorization even for relatively large codes that use real arithmetic solely—a class of codes notoriously hard to vectorize.

"Look-ahead" Capabilities. The vectorizer tries to detect failure branches as soon as possible, as this can prune the search tree significantly. That way, the vectorizer can traverse a part of the search space strongly reduced in size, without missing any relevant part.

Acceleration of Backtracking. Prior versions of the vectorizer simulated backtracking using monads, i. e., algebraic constructs used in functional languages for the abstraction of control and data flow. While this technique is convenient, it does not allow for cheap reclaiming of memory upon backtracking, which is a significant obstacle to fast backtracking. The newly development version of the MAP vectorizer is implemented in Mercury, a high-performance logic programming language that includes support for efficient backtracking [76].

Selection of the Best Solution. Vectorization may yield more than one solution. All prior versions of the vectorizer immediately committed to the first solution found, thus keeping the vectorization runtime low. Outdated vectorizers only relied on a local adaptation of the order in which vectorization alternatives were tried. With regard to code quality, however, picking the best among all solutions would be the optimal strategy. Clearly, this cannot be done in a straightforward way by generating and testing all solution candidates, except for very small codes. To account for this dilemma, the new version of the MAP vectorizer uses a branch-and-bound method (together with a restart mechanism) to get the best solution, while keeping the vectorization runtime reasonably low.

The implemented technique allows interrupting the vectorization process at any given point, returning the best solution found so far. This is important for very large codes, as finding a good or maybe even an optimal solution usually takes much less time than proving that some solution actually *is* optimal, which always is the last step of the vectorization.

Enhanced Adaptability. All older prototypes of the vectorizer have been specifically adapted to exactly one target architecture. While deriving a new port out of existing ones was not very complicated, the resulting code bloat led to increased maintenance effort.

The new version of vectorizer unifies all target architectures by adding support for a *penalty vector*, which directly reflects the peculiarities with regard to optimal choices to be taken by the vectorizer—similar to the data shown in Tables 3.7 and 3.8 on page 43. This enables the vectorizer to adequately support a large variety of target instruction set architectures, ranging from well-established ones, like Intel's SSE2 and AMD 3DNow!, to new ones, like Intel's SSE3 and IBM's "Double FPU" present in the PowerPC 440 FP2.

4.3.1 Vectorization Example

Fig. 4.3 gives an example of SIMD vectorization, showing in the left-hand part the input DAG that represents a 3-point complex FFT produced by `genfft`, and in the right hand part one particular output of 2-way SIMD vectorization.

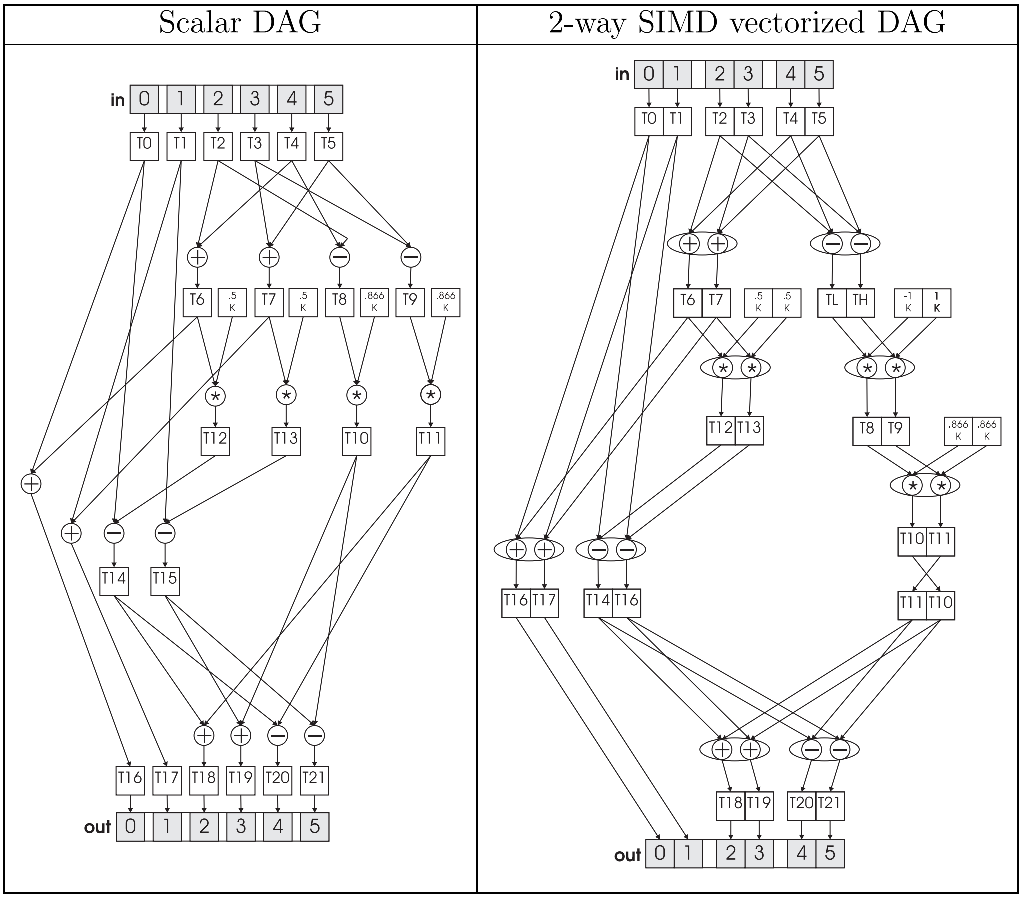

Figure 4.3: Vectorization of a Scalar FFT of Size 3. The scalar DAG in the left part of the illustration is computationally equivalent to the SIMD vectorized DAG depicted in the right part. Note the extra auxiliary SIMD multiplication using the constant $(-1, 1)$, that is used to change the sign of the lower part of one of the SIMD variables. This auxiliary instruction corresponds to the expansion of a SIMD pseudo-instruction, performing one addition and one subtraction. Instruction combinations are subsequently improved by MAP's peephole optimizer.

Chapter 5

The MAP Optimizer

The MAP optimizer is the middle part of the Vienna MAP compiler tool chain, processing the output of the vectorizer and producing the input of the backend. It supports both scalar and SIMD codes, adapting the corresponding DAG according to one or more sets of transformation rules that aim at both a reduction in code size and at a minimization of runtime.

Input and Output Code. The optimizer operates on input code in static single assignment (SSA) form representing a directed acyclic graph (DAG). The optimizer performs a source-to-source transformation, using the same high-level language for input and output.

Assignment of Responsibilities. The optimizer is implemented as a separate module, instead of being integrated directly into the vectorizer. This separation shows several advantages. Firstly, this helps to keep both parts as simple as possible. Secondly, this allows assigning clear responsibilities with regard to code quality to the individual components of the tool chain. While the vectorizer is responsible for the global structure of the code, the optimizer solely cares about its local structure.

Optimization Focus. The optimizer tries to adequately meet various code quality criteria. While most rules focus on code size as the optimization criterion, other rules focus on speed, yet others on properties of the corresponding DAG, which affect code size or speed only in an indirect way, like path lengths, the number of different operands used, etc.

Peephole Optimization. The MAP optimizer is based on peephole optimization, which is a common-place technique for improving code based on successively rewriting parts of the code. A window (peephole) slides over several instructions that are logically connected by direct (producer-consumer) data dependencies, attempting to match the instructions within the peephole with some predefined patterns. If a match succeeds, the instructions within the window are rewritten to a semantically equivalent sequence of instructions. This match-and-replace procedure is repeated in a loop until all matching attempts fail, i. e., the procedure terminates as soon as a fixed point is reached.

Locality. As each optimization rule only looks at a peephole of some fixed size, usually being smaller than a handful of instructions, peephole optimization is a completely local optimization approach that does not alter the global structure of the input code significantly. Still, the resulting output code is not likely to be similar to the original input code, especially if intensive rewriting has taken place.

Implementation Basics. Implemented as a committed-choice term rewriting system, the peephole optimizer uses one or more sets of rules, each with a different priority. Out of all applicable rules, the rewriting engine selects one having the highest priority and commits to it, i. e., the engine disregards all possible alternatives that *could* have been, but did not get selected. The chosen rule is then used to substitute the sequence of instructions within the peephole by a semantically equivalent but optimized sequence of instructions. This step of looking for an applicable rule, committing to it, and applying it to the code is iterated until no further rule is applicable, i. e., until a fixed point is reached.

Precision Aspects. Like all other components of the MAP tool chain, the optimizer supports numerical data with arbitrary precision to ensure that no precision is lost during the compilation process. Numerical data is converted to the requested target precision (single or double precision) not before the last compilation step.

Dynamic Reordering. The order in which rules are considered for application can have a profound impact on the speed of the rewriting process, which is particularly important when compiling very large code segments consisting of thousands of instructions, like the codes automatically produced by the program generators of FFTW and SPIRAL.

To improve the performance of the peephole optimizer, the MAP optimizer allows dynamically adapting the order in which rules are tried in a most-recently used (MRU) fashion, favoring rules that have been applicable in the recent past.

However, while speeding up the rewriting process, dynamic reordering may also have negative effects. Excessive reordering may impede the development and debugging of large rule sets, as it obstructs understanding the rewriting process. To assist the development of larger sets of rules with predictable rewriting behavior, the optimizer allows confining the amount of reordering by partitioning the rules into classes according to their priority. Dynamic reordering is then limited to each one of these classes.

Term Rewriting Properties. Even relatively simple term rewriting systems may exhibit complex behavior. Therefore, it is often advisable to concentrate not on the arcane details of each individual rewriting step, but rather to shift the focus on two distinguished properties of the entire rewriting process.

A term rewriting system is said to be *confluent* if all potential ways of applying rules leads to the same result. In principle, this is an important and desirable property, as it allows disregarding the concrete rule selection mechanism when trying to get an overall understanding of the whole term rewriting system. The term rewriting system implemented within the MAP optimizer cannot be guaranteed to be confluent, but instead operates in a committed-choice style, to ease the comprehension of the rewriting process.

A term rewriting system *terminates* if it produces an answer after a finite number of steps. Especially in the context of large rule sets that include rules pursuing different goals, i. e., "rules that work against each other", termination is a critical issue. For instance, some rules might try to move instructions of some particular type towards the inputs of the DAG, while other rules might aim at moving the same instructions into the opposite direction. To avoid cycles, sets of antagonistic rules must be split into several passes that are carried out sequentially. The optimizer then uses one rule set after the other to rewrite the DAG until a fixed point is reached and then continues with the next rule set.

Different Kinds of Rewriting Rules. The MAP optimizer uses two kinds of rules working in synergy: (*i*) Improving rules and (*ii*) assisting rules.

Improving rules aim at an immediate improvement in code quality. Examples include rules for fusing two neighboring instructions into one, or rules handling horizontal SIMD instructions with neighboring SIMD swaps. Fig. 5.4 illustrates some improving rules for accumulate instructions with neighboring swaps.

Assisting rules do not immediately improve the code, but adapt the DAG by changing the position of some instructions relative to their neighbors, e. g., by moving SIMD swap instructions or SIMD multiplications by a constant, eventually allowing to apply some improving rule. In most cases, multiple applications of assisting rules are needed, before an opportunity to apply some improving rule arises. Most optimization rules implemented in the MAP optimizer are assisting rules. Figs. 5.2 and 5.3 show the working of an assisting rule operating on a SIMD binary instruction and on one or more neighboring SIMD swap instructions.

Current Implementation Status. Like other components of the MAP tool chain, the current version of the optimizer is written in OCAML [15], a strict functional programming language with static typing. The optimizer uses monads abstracting both control and data flow to implement the backtracking required for the rule matching in a compact way and provides precise control over rewriting, e. g., by offering multiple rule selection methods.

Future Work. The most critical point in the current implementation is that the rule code itself is not well-suited for the purpose of presentation.

To solve that issue, work on a peephole optimizer generator has already begun. The optimizer generator takes a simple, yet versatile representation of the rules, and translates the rules to high-level code written in the MERCURY programming language [75], using the high-level built-in language features for efficiently realizing backtracking.

Because it is operating on a higher level, an optimizer generator significantly cuts down development time, easing both writing and debugging, as well as the adaptation to a new target architecture.

Representing rules explicitly as data instead of code offers a number of advantages: (*i*) Efficiency can be increased, as the optimizer generator may create several specialized variants of rule codes. (*ii*) It is possible to reason about specific properties of the term rewriting system properties, particularly about termination. (*iii*) It allows automatically deriving an equivalent graphical representation of optimization rules including all relevant information, like preconditions, the pattern being matched, auxiliary goals, and the optimized code pattern.

5.1 Scalar Rules

The optimizer includes support for scalar floating-point code, performing both FMA extraction and standard compiler optimization techniques [63] like dead code elimination, copy propagation, constant folding, as well as the removal of redundant operations.

As the MAP tool chain primarily targets SIMD vectorized code, earlier prototypes of the optimizer [50, 52] did not even include support for scalar code. Support has been added to cover the cases when vectorization fails, and for the purpose of evaluating the backend (with the vectorizer explicitly turned off).

Fig. 5.1 illustrates a non-trivial case of FMA extraction that does not only produce one FMA instruction, but also creates the opportunity for extracting a second one. SIMD FMA extraction works analogously to the scalar variant, because SIMD FMAs are parallel inter-operand instructions.

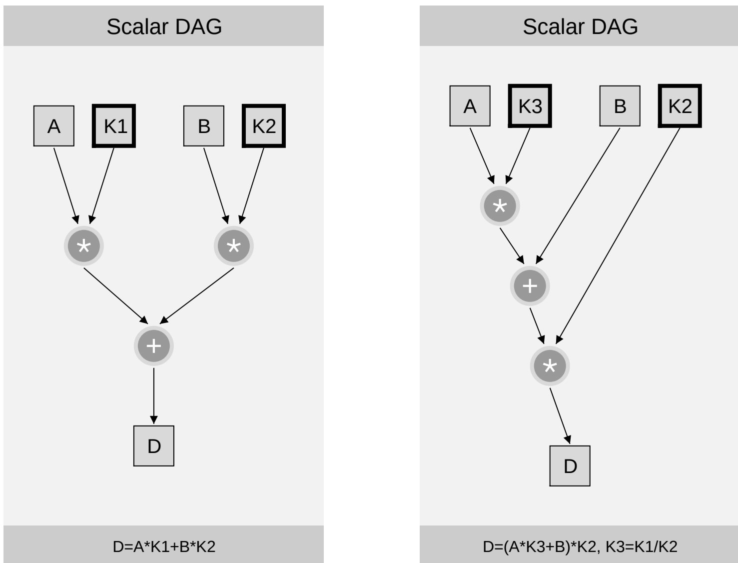

Figure 5.1: Scalar FMA Extraction. The DAG depicted on the left shows a frequently occuring code pattern, comprising two multiplications by constants and one dependent addition. While this pattern could immediately be mapped onto one multiplication and one dependent FMA, one multiplication may also be propagated towards the consumers of D, as shown in the right half of the illustration. This method usually gives a better result, as the (temporary) multiplication can be fused with a dependent addition or multiplication by a constant.

FMA extraction may also have adverse effects, as it may increase the path lengths and contribute to a larger number of numerical constants. Still, such negative effects are usually of minor concern because of the much more important positive impact of the higher throughput and smaller code size brought about by the usage of FMA instructions.

5.2 Generic SIMD Rules

While only a modest number of rules—needed for dead code elimination, copy propagation, constant folding, and the removal of redundant operations—are *fully* target architecture independent, still a relatively large number of rules are beneficial for all target architectures, because of general similarities of the respective 2-way SIMD instruction set architecture (ISA) extensions.

Implemented Rules. Whenever possible, two neighboring unary instructions ("multiplication by a constant", swap, and sign-change) are combined into one. Several subsets of rules cover combinations of binary instructions (addition, subtraction, multiplication, and unpack) and neighboring unary instructions (swap and sign-change).

Figs. 5.2 and 5.3 illustrate the optimization of a commonly occuring code pattern consisting of some vertical SIMD binary instruction and a number of neighboring auxiliary SIMD swap instructions. Similar rewriting is implemented also for ternary SIMD instructions (FMAs).

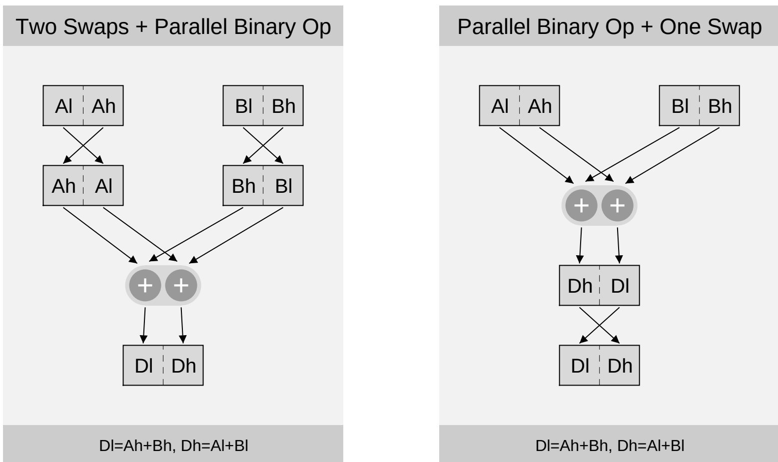

Figure 5.2: SIMD Vertical Instruction Optimization. Parallel SIMD instructions with several neighboring swap instructions can be optimized as shown in the illustration. In the left half, a parallel SIMD instruction is preceded by two swaps. In the right half, the two swaps have been replaced by one swap following the binary instruction.

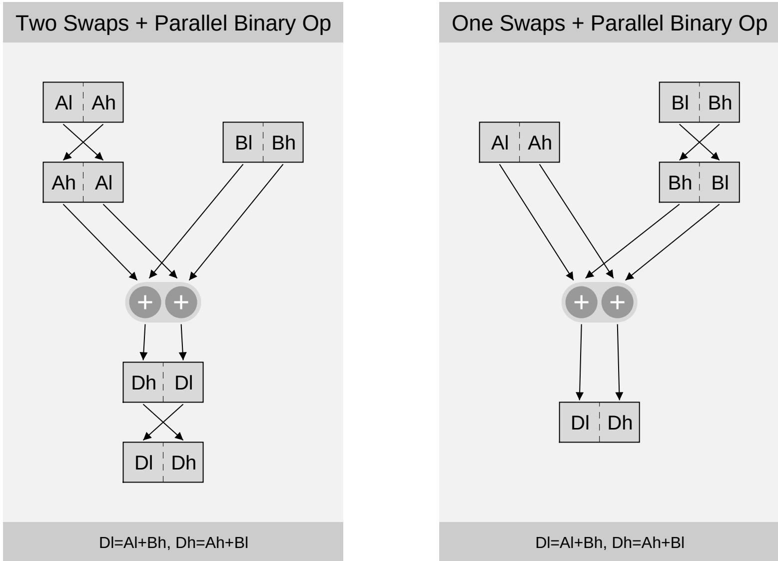

Figure 5.3: SIMD Vertical Instruction Optimization. Another common combination of "Swap" instructions and parallel binary SIMD instructions allows minimizing the number of SIMD reordering instructions. In the left picture, both one source and the destination of the SIMD binary operation are swapped. In the right picture, the other source operand is swapped, reducing the number of SIMD swap instructions by one.

SIMD auxiliary instructions like swaps or sign changes may be propagated both towards the output of the DAG or towards the input of the DAG. Fig. 5.2 shows how a SIMD swap is propagated towards the output of the DAG. In Fig. 5.3 this SIMD swap is moved towards the input of the DAG.

SIMD Binary Reordering. Although some rules cover sequences containing unpack instructions, the optimizer typically does not reduce the number of unpacks significantly, because of the locality of peephole optimization. Within the MAP compiler tool chain, it is the responsibility of the producer of the input to the optimizer, i. e., the program generator or the MAP vectorizer, to minimize the number of unpack instructions in the first place.

5.3 Target Architecture Specific SIMD Rules

One set of target architecture specific rules tries to discover instruction sequences that can be rewritten to *exactly one* target instruction.

To create sequences of this kind, another set of rules tries to move some instructions (swap instructions, multiplication by a constant) to some other place of the DAG, eventually enabling the first set of rules to optimize the respective sequences.

A third group of rules covers code patterns frequently occuring in FFTs and other linear signal transforms.

Accumulate Specific Optimizations. These optimizations are only relevant for ISA extensions natively supporting intra-operand style (accumulate) SIMD instructions (AMD 3DNow! and Intel SSE3). Fig. 5.4 illustrates the rewriting of two code patterns consisting of some horizontal SIMD instruction and neighboring SIMD swap instructions into their optimized form.

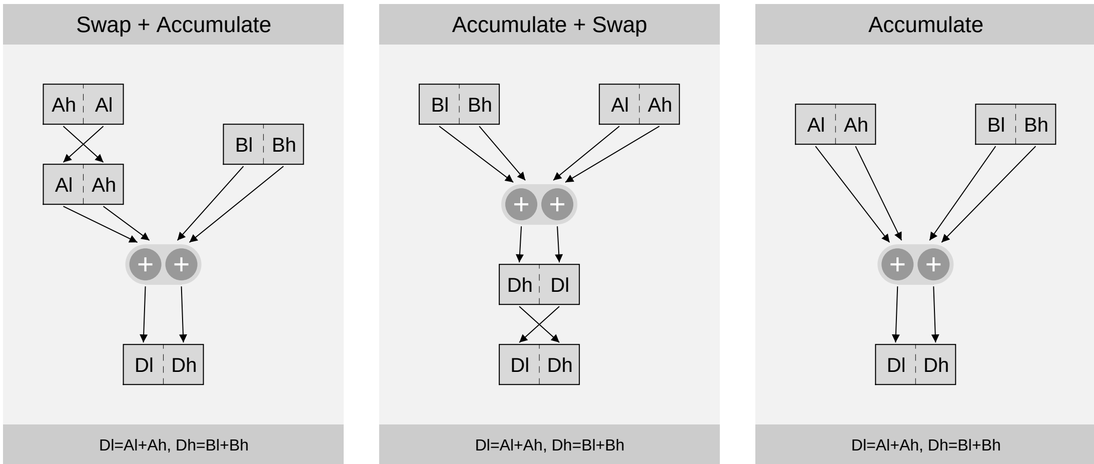

Figure 5.4: SIMD Accumulate Optimization. SIMD accumulate instructions can be combined with neighboring "Swap" instruction if the respective basic scalar operation is commutative. This reduces the SIMD reordering instruction count and shortens the path length from inputs to output. The left two pictures show unoptimized SIMD DAGs, the right one the optimized variant.

For IBM's Blue Gene Double FPU and other targets including Intel's IA64 and SSE2, accumulate instructions are rewritten to a combination of unpack and parallel SIMD instructions.

FMA Specific Optimizations. These optimization rules are suitable for all targets supporting FMA instructions, including the IBM PowerPC 440 FP2 (Blue Gene) and the Intel Itanium (IA64).

Blue Gene Specific Optimizations. Several rules aim at the utilization of Blue Gene specific FP2 instructions offering both calculation *and* data movement (cross, cross copy). This set includes rules for making use of cross copy (primary/secondary) multiplication instructions for performing complex number multiplication (without any additional reordering instructions). The majority of the rules is dedicated to eliminating swaps and sign changes by utilizing variants of cross FMA instructions.

5.4 Substitution Rules

Substitution rules are a special case of peephole optimization with the peephole size being exactly one. As the substitution rules currently implemented in the optimizer cannot possibly trigger any further rule application, the optimizer performs a separate substitution pass, as soon as all other rewriting has finished.

In the MAP optimizer, substitution rules are used for the following purposes.

Rewriting Trivial FMAs. Certain SIMD FMA instructions using the specific constants $(+1.0, +1.0)$ or $(-1.0, -1.0)$ and scalar FMA instructions using the specific constants $+1.0$ or -1.0 are rewritten to ordinary additions or subtractions. This reduces the register pressure slightly, as fewer source operands are needed. Also, this rewriting contributes to lowering the number of constant loads.

Rewriting Unsupported Pseudo-Instructions. A small number of SIMD instructions extracted by the MAP vectorizer may not be supported by the particular target architecture. For instance, the vectorizer may extract horizontal SIMD instructions, even though the target offers only vertical SIMD instructions, if vectorization is not possible otherwise. After optimization has finished, such pseudo instructions need to be rewritten to a sequence of actually supported instructions. For instance, horizontal SIMD instructions are mapped onto binary SIMD reordering (unpack) instructions as well as vertical SIMD instructions.

Reduction of the Number of SIMD Constants. Vectorized code may use much more constants than its scalar counterpart, because of two reasons. Firstly, 2-way vectorization might blow up the number of constants from n to n^2. Secondly, the merging of sign-change operations with neighboring multiplications might increase the total number of constants by up to a factor of four. Although these worst-case scenarios hardly ever occur in practice, reducing the number of constants still is an important issue in producing optimized vector code.

While the vectorizer is responsible for solving the first problem, the optimizer tackles the second problem by rewriting FMAs with some 2-way constant $(-k_1, +k_2)$ to FMAs with the constant $(+k_1, -k_2)$, thus effectively propagating the sign-change into the FMA instruction.

Chapter 6

The MAP Backend for Blue Gene

The MAP backend is the final stage of the Vienna MAP compiler tool chain, compiling the high-level intermediate representation of the source code to target specific assembly code.

Previous Development. The earliest prototypes of the MAP tool chain date back to the fall of the year 2000. At that time, no reasonable C compiler support for SIMD intrinsics was available for the targeted platform, i. e., the x86 compatible AMD Athlon processor with 3DNow! SIMD ISA. So the backend was initially developed, not because the code produced by available C compilers was not satisfactory in quality, but to get SIMD object code produced at all.

Since 2000, general purpose compilers (like GNU C, Intel C, or IBM XL C) gradually included support for SIMD intrinsics. Still, having an own backend in the MAP tool chain is an important advantage, as it allows taking care of the specific requirements of signal transform codes. The properties of these target codes have been directly reflected in the design of the backend.

Experiments [30, 49, 27] show that MAP's x86 backend produces code of higher quality than available general purpose C compilers (GNU C, Intel C) when compiling FFTW codelets.

Focus on Straight Line Code. The targeted source codes are automatically generated signal transform codes that spend most of their runtime executing straight line code. Accordingly, the MAP backend focuses on this case, which helped keep the design as simple as possible.

The kernel codes created by the program generators of FFTW and SPIRAL may be sometimes very large in size, much larger than normal hand-written codes, which might push compilers to their limit [28]. To handle such source codes satisfactorily, the backend's design had to avoid any unnecessary limits with regard to the size of the input code, the number of variables used, etc.

To deal with less performance critical parts of the code, as well as codes containing control structures for iteration, conditionals, and recursion, the backend works in tandem with a general purpose C compiler.

Focus on Address Generation. The targeted input codes operate on data stored in variably-strided arrays, i. e., one-dimensional arrays having strides that are unknown at compile time. Variably-strided one-dimensional arrays are often used to access particular parts of multi-dimensional arrays. To account for the frequent use of this kind of array access, the MAP backend pays particular attention to optimizing the integer code required for effective address calculation.

Portability. Although any backend has to be target architecture specific, most parts of the MAP backend are kept very flexible and generic, and thus can be easily adapted to new hardware architectures, just by using suitable parameter files. Similarly, within one particular processor family, the backend can be adjusted to specific micro-architecture peculiarities by parameter adaptation.

Room for Experimentation. A lean backend is the prerequisite for carrying out experiments in ways that would be extremely time-consuming with full-fledged open-source compilers like GNU C, and impossible with commercial closed-box compilers like Intel C or IBM XL C.

(*i*) The MAP backend offers precise control over all kinds of optimization and over the order of their application, which turned out to be a key factor to achieving good performance.

(*ii*) The backend includes several types of optimization done in an ad-hoc style, similar to the ones done by hand-coders, to effectively deal with some of the specific limitations of the primary target platforms, i. e., IBM's Blue Gene systems. In a sense, the development of the MAP compiler tool chain bears some similarity with the work done by skilled hand-coders. However, the experience and knowledge gained through extensive experimentation is not lost in the future, as it is encoded into the compiler.

Parts of the Backend. Unlike other parts of the tool chain, the MAP backend consists of a number of relatively small components that are dedicated to either (*i*) resource allocation or to (*ii*) the scheduling of code.

Phase Ordering. Register allocation and instruction scheduling have conflicting goals. While the register allocator tries to minimize the number of memory transfer operations (register spills, register reloads, and constant data loads), the instruction scheduler tries to maximally utilize the available execution units by spreading out instructions according to their respective latencies.

The conflicting goals of these two phases can be harmonized in different ways. (*i*) Register allocation and instruction-level scheduling may be combined into one single complex pass [36]. (*ii*) Register allocation and scheduling may be applied in an iterative style, starting with only a few register assignment constraints and adding more and more constraints with each iteration.

Earlier prototypes of the backend used a fairly complex allocation-scheduling-reallocation loop, controlled by a runtime estimate produced by the instruction-level scheduler. The current version proceeds in a simpler, much more intuitive way that takes maximum advantage of being in the restricted domain of signal transform codes. First, the domain-specific high-level scheduler of `genfft` is used to produce a topological sort of the DAG, which minimizes register pressure. Next, a medium-level scheduler adapts the instruction order, taking the instruction latencies into account, but neglecting the effects of super-scalar execution. This kind of scheduling tries to avoid all dispensable movement of instructions, to preserve as much of the domain-specific high-level schedule as possible. After the two scheduling passes, register allocation using the Belady-Min policy [8] is carried out. Finally, a list-scheduler performs the instruction-level scheduling of the code, based on a model that is a super-scalar in-order approximation of the target architecture. All schedulers present in the MAP backend are target-independent and are configured towards some particular target architecture by machine description parameter files.

6.1 Resource Allocation

High-level SSA code, as output by the peephole optimizer (see Chapter 5), implicitly assumes that unlimited resources are available. Before SSA code can be executed on some target machine, all referenced resources need to be mapped to the finite resources actually available on the target architecture, which comprises two tasks. (*i*) All temporary floating-point (SIMD) variables need to be mapped to a finite set of logical floating-point (SIMD) registers available on the target architecture. (*ii*) All floating-point (SIMD) memory access operations need to be translated to instruction sequences consisting of auxiliary integer instructions and floating-point (SIMD) memory access instructions (loads/stores).

Hardware Assumptions. For the sake of simplicity, the backend assumes that (*i*) all computations are done operating on floating-point scalars or SIMD vectors, and that (*ii*) there are at least two distinct register files—one for integers, and at least one for floating-point data. These assumptions have held true on all architectures targeted by the different versions of the backend produced so far.

Software Assumptions. The backend requires that all calculations in the code to be compiled do not operate on data stored in general purpose integer registers. Thus, all integer instructions are auxiliary instructions, devoted to either effective address generation or to fulfilling the procedure calling convention.

6.1.1 Register Allocation

Register allocation assigns temporary variables to logical registers, such that variables holding alive values are not mapped to the same registers. For most codes, the number of temporary variables alive at some particular time outnumbers the number of logical registers by far. Accordingly, register allocation has to insert auxiliary data-transfer instructions between registers and stacks, i. e., spill and reload instructions.

The backend performs register allocation for all floating-point scalar and (SIMD) vector register files. It uses the “farthest-first” Belady-Min policy [8] for choosing a spill victim, which has proven useful for minimizing the number of auxiliary register transfer operations [38].

The register allocator of the MAP backend can be used in different ways. (*i*) Allocation may precede instruction scheduling. (*ii*) Allocation may follow instruction scheduling. (*iii*) Allocation may follow some previous allocation, trying to "improve" false dependencies introduced during register allocation by renaming logical registers using a least-recently-used (LRU) heuristic. Register reallocation may further reduce the number of spills and reloads.

In the current version of the backend, the code is scheduled on several levels (domain-specific high-level scheduling, medium-level scheduling) before registers are allocated. Medium-level scheduling is aware of both register allocation and instruction scheduling, and tries to avoid unnecessarily constraining the choices either one has. In practice, this has been a reasonable strategy.

Parameterization of the Register Allocator

The register allocator is adapted to a particular target architecture by using the following directives shown in Fig. 6.1, which are ground (variable-free) PROLOG terms, as defined by the predicate `is_regallocDirective`. The actual source code of the register allocator accepting these directives is presented in Appendix A.5.

```
is_regallocDirective(src_dst_stack(Src,Dst,S)) :-
        is_atom(Src),
        is_atom(Dst),
        is_atom(S).
is_regallocDirective(dst_forceReuse_set(Dst,Force,B)) :-
        is_atom(Dst),
        is_bool(Force),
        is_baseSet(B).
is_regallocDirective(stack_spill_reload_set(S,Spill,Reload,B)) :-
        is_atom(S),
        is_atomOrCompound(Spill),
        is_atomOrCompound(Reload),
        is_baseSet(B).
is_regallocDirective(constantLoads(Cs)) :-
        is_constantLoads(Cs).
```

Figure 6.1: Register Allocation Directives. Four directives control the mapping of source operands to destination operands, using some stacks as temporary storage. Multiple sets of source and destination operands may be used. Source operands may or may not be mapped to the same destination operands, which is useful for architectures that use the same register file for holding scalar and vector data. Instructions loading constant values can be declared to the allocator to avoid the spilling of constant data.

The directives shown in Fig. 6.1 have the following meaning.

`src_dst_stack(Src,Dst,S)` lets the register allocator map operands of type *Src* to destination operations of type *Dst*, using the stack S to hold values not fitting into the destination set.

`dst_forceReuse_set(D,F,B)` defines that the destination set D is associated with with the buffer B. The parameter F controls optional caching.

`stack_spill_reload_set(S,St,Ld,B)` links the stack named S with the buffer B, and declares *St* and *Ld* as the access operations for this particular stack.

`constantLoads(Cs)` states that all instructions matching any of the patterns in the list *Cs* load some constants, which allows avoiding spilling constant data.

Buffers are defined by the predicate `is_buffer`, as shown in Fig. 6.2.

```
is_buffer(first_then(FirstSet,ThenSet)) :-
        is_baseSet(FirstSet),
        is_baseSet(ThenSet).
is_buffer(modestly_fifo_creation(M,FIFO,BaseSet)) :-
        is_bool(M),
        is_bool(FIFO),
        is_baseSet(BaseSet).
```

Figure 6.2: Buffers. Every destination operand used during register allocation is associated with a buffer, defining the concrete allocation policy to be used.

`first_then(F,T)` denotes an uncached buffer comprising two base sets. Whenever the register allocator needs a fresh destination operand, it first tries to draw a member from the base set F, or if F has become empty, from the base set T. Whenever a source operand is no longer alive, the register allocator releases the corresponding destination operand, putting it back into the second base set, T.

`modestly_fifo_creation(M,F,C)` denotes a buffer having an internal cache X. That buffer draws entries from the cache X or, if this cache is empty, from the base set C. If M equals *true*, released entries are put back into the internal cache X, which minimizes the number of different destination operands used. If M equals *false*, released entries are put back into the base set C, which minimizes the number of false dependencies introduced by register allocation. If F equals *true*, the cache X is used in a first-in first-out (FIFO) style. If F equals *false*, it is used in a last-in first-out (LIFO) style.

Base sets are defined by the predicate `is_baseSet`, as shown in Fig. 6.3.

```
is_baseSet(infiniteFrom(I)) :-
        is_nonNegativeInteger(I).
is_baseSet(finite_from_fifo_modestly(N,I,F,M)) :-
        is_nonNegativeInteger(N),
        is_nonNegativeInteger(I),
        is_bool(F),
        is_bool(M).
```

Figure 6.3: Base Sets. Base sets may be finite or unlimited in size. Finite base sets are used for representing logical registers offered by the target architecture, while base sets of unlimited size are used for representing stacks.

`finite_from_fifo_modestly(N,I,F,M)` represents a cached finite base set of N ascending integers, starting at I. Like some buffers, finite base sets have an associated cache, and act similarly to buffers with regard to the caching behavior (LIFO or FIFO style, modest or generous allocation).

`infiniteFrom(I)` represents a base set of unlimited size, comprising ascending integers starting at the non-negative integer I. Unlike finite base sets, base sets of unlimited size discard released entries returned to them.

Register Allocation Parameters for Blue Gene Processors

The register allocation directives used to customize the allocator for the IBM PowerPC 440 FP2 processor are shown in Fig. 6.4.

```
stack_spill_reload_set(
   fpstack, fv_st, fv_ld,
      modestly_fifo_creation(false,true,infiniteFrom(0))).
stack_spill_reload_set(
   fp2stack, f2v_st2(p), f2v_ld2(p),
      modestly_fifo_creation(false,true,infiniteFrom(0))).

dst_forceReuse_set(
   fp2r, true,
      first_then(finite_from_fifo_modestly(14,0,false,false),
                 finite_from_fifo_modestly(18,14,true,true))).

src_dst_stack(fvr, fp2r, fpstack).
src_dst_stack(f2vr,fp2r,fp2stack).

constantLoads([i(f2v_ld2(p),[c(_)],[_],_), i(fv_ld, [c(_)],[_],_)]).
```

Figure 6.4: Register Allocation Directives for the IBM PowerPC 440 FP2. Similar directives can be used for all common-place RISC-style processors having one shared register file for scalar and vector floating-point data.

The directives presented in Fig. 6.4 have the following effect.

`stack_spill_reload_set`, used twice in Fig. 6.4, defines two distinct stacks, both of which are unlimited in size. The stack *fpstack* is used exclusively for scalar data, the stack *fp2stack* exclusively for vector data. Data transfer operations for the stack holding scalar data are *fv_st* and *fv_ld*, for the stack holding vector data *f2v_st2(p)* and *f2v_ld2(p)*.

`dst_forceReuse_set` defines that there are 32 distinct destination operands of the form *fp2r*, where the first 14 are to be used generously and the remaining 18 are to be used modestly. The reason for this peculiarity is the calling convention defined by the PowerPC application binary interface, which declares the first 14 floating-point (SIMD vector) registers as caller-saved, and all remaining ones as callee-saved. Minimizing the number of callee-saved registers used results in a minimization of the procedure prolog and epilog code, which is particularly beneficial for small and middle-sized codes.

`src_dst_stack`, used twice in Fig. 6.4, associates both the scalar and vector source operands (*fvr* and *f2vr*) with the same destination operands of type *fp2r*. However, when data needs to be spilled from the register file to the stack, scalar values are spilled to (and reloaded from) the stack *fpstack*, whereas SIMD values are spilled to (and reloaded from) a different stack, *fp2stack*. This helps conserve stack space, and allows using scalar memory access instructions for scalar values, which is significantly cheaper than using SIMD memory access instructions, because of the larger number of supported addressing modes.

`constantLoads` specifies two instruction patterns (for scalar and SIMD code, respectively) that load some constant *c(_)*.

6.1.2 Effective Address Generation

All integer instructions in any code produced by the MAP compiler are devoted to either fulfilling the ABI calling convention or to calculating effective addresses.

While the code for fulfilling the calling convention has a constant size (regardless of the actual size of the procedure to be compiled) for procedure prolog and epilog, the code needed for effective address calculation may grow linearly with the number of memory access operations in the procedure to be compiled.

Experiments [49] have shown that the portion of the code needed for the calculation of effective addresses often has a significant negative performance impact in case of algorithms having a high ratio of the memory access count compared to the number of arithmetic operations. All fast algorithms for linear signal transforms possess this property.

Minimizing the number of integer auxiliary instructions is important for all target architectures, i. e., Intel x86, AMD64, and IBM PowerPC. However, producing high quality code for the IBM PowerPC 440 FP2 processor is particularly challenging, as there is no "register+immediate" addressing mode available for SIMD instructions accessing memory, i. e., Blue Gene processors have DSP-like addressing mode limitations for SIMD loads/stores. These limitations do not only affect accesses to variably-strided arrays, but also access operations to the stack and to numerical constants, considerably increasing the amount of code needed for accessing the stack and numerical constants. Minimizing the number of these extra instructions poses a challenge not present when generating code for general purpose processors.

To meet the requirements of IBM PowerPC 440 FP2 processors, the address generation part of the MAP backend had to be redesigned from scratch, which led to a significant improvement of code quality. The backend now handles all kinds of memory access operations (to array data, the stack, or constants) uniformly in one pass, combining integer instruction selection and integer register allocation.

Basic Problem. The calculation of effective addresses of elements of variably strided arrays can be done straightforwardly by using integer multiplication instructions. However, these instructions are expensive (low throughput, high latency) on all general purpose processors, including the IBM PowerPC 440 FP2.

Strength Reduction. The commonplace approach to addressing this problem is strength reduction [63], which aims at replacing complex non-pipelined (low throughput, high latency) instructions like integer multiplications by sequences of simpler pipelined (high throughput, low latency) instructions like integer additions, subtractions, and arithmetic shifts.

Depending on the contents of integer register file, which holds both procedure arguments and temporary values calculated earlier, there is a large number of different ways of doing strength reduction.

Doing strength reduction in a hard-coded fashion implies making instruction selection decisions without properly considering the temporal context, thereby missing opportunities to (*i*) reuse already calculated factors still residing in the register file and (*ii*) pick factors that could be beneficial for some proximate address calculation to be carried out in the near future.

This common hard-coded approach to performing strength reduction is simple, keeps compilation times low, and works well for many applications—but unfortunately not for numerical calculations in fast linear transform algorithms.

Implementation Idea. To produce high-quality code, the MAP backend interleaves integer instruction selection and integer register allocation, thus removing a classical compiler optimization barrier. The backend avoids premature commitment to one particular factorization or reduction, but considers a larger number of combinations, utilizing a blended mixture of well-established AI search methods, depth-first iterative deepening [48] (DFID) and dynamic programming (DP).

As exhaustive search for an optimal solution is too time consuming for all but the smallest codes, the backend (*i*) looks at reasonably sized sub-problems, (*ii*) solves these sub-problems optimally, and (*iii*) combines the respective optima to a hopefully well-performing solution of the original problem. The quality of this solution depends on the amount of overlap of the sub-problems considered and on the their size.

To control the amount of search performed, the backend offers a set of parameters to directly control the speed and quality of the search, allowing to trade compilation time for code quality, by specifying the size and the amount of overlap of the sub-problems.

Implementation. The backend starts out the search with the register file only holding the procedure arguments, but not holding any temporary values.

One base step of the search algorithm utilizes a bottom-up forward-chaining approach to find sequences of integer instructions calculating the next n effective addresses. Depth first iterative deepening is used to find the minimal number of instructions sufficient for calculating all effective addresses, in the original order. If some effective address calculation requires more than m auxiliary integer instructions, the strength of that particular expression is not reduced, but a multiplication instruction is issued instead to keep the instruction count low. In practice, $m = 3$ has turned out to be a good compromise, as the vast majority of effective addresses can be calculated with three or less simple integer instructions.

The backend commits to the first k instructions of the found sequence, i. e., it updates the register file entries accordingly, allowing for future reuse of the newly calculated temporary values, now present in registers. It then strips off the effective addresses covered by these k instructions. This procedure is repeated until code for calculating all requested effective addresses has been generated.

Hand-Coding Stratagems. To accelerate the search process for effective address calculation code and to improve the quality of the generated code, the MAP backend for Blue Gene utilizes some clever hand-coding devices.

When the integer register `r0` is used as the base of the "register+register" addressing mode, the processor does not use the content of `r0` as the base, but the absolute value of 0 instead. This helps operating system code, as it allows fast access to data lying in the first (or last) 32kB of the address space, but is of no practical use for application code. The MAP backend uses `r0` to hold the integer constant 16, i. e., the size of a 2-way SIMD vector in double-precision. That constant is used frequently, e. g., when accessing SIMD constants stored in double precision, when spilling data to (or reloading it from) the stack, or when accessing unit-strided arrays of complex numbers.

The IBM PowerPC FP2 efficiently supports consecutive memory accesses by offering an update form of the "register+register" addressing-mode, which calculates the effective address as the sum of contents of the two given registers, and updates the base register.

To take maximum advantage of update forms, constants are stored in the order, in which they are actually used. This guarantees consecutive access for all constant loads, but may cause some constants to be stored more than once. As it turned out, this replication of data is not critical with the codes of interest, even if the number of constants is large.

Similarly, the backend tries to use consecutive accesses for accessing the stack, by utilizing two stack pointers instead of one. One stack pointer is used solely for writing to the stack, while the other stack pointer is solely used for reading from the stack. Entries are always written consecutively, but are usually read back non-consecutively. The opposite behavior, i. e., all reads are consecutive while writes are done non-consecutively, is supported, but not used, as it is somewhat less cache-friendly. The backend does not reuse stack cells no longer used, which minimizes instruction scheduling constraints, but increases memory requirements. Reusing unused stack cells is also supported, but would increase the number of auxiliary integer instructions, as it causes both reads from and writes to the stack to be non-consecutive.

Pragmatic Aspects. Previous implementations of the backend [52] followed a similar approach, but never looked at more than one effective address calculation at a time, i. e., $k = n = 1$. The current implementation significantly improves upon this by supporting much larger window sizes ($n > 10$). Even with a high amount of overlap ($k=1$), this compilation step usually requires no more than a few minutes, even for the largest codes ever occuring in practice.

Interestingly, the search for good effective address code gets easier with a larger number of logical integer registers being available. This appears to be counter-intuitive at first sight, as a larger number of registers results in a larger search space. However, with a larger number of temporary values being held in registers, optimal sequences of auxiliary integer instructions tend to be shorter, which significantly speeds up depth-first iterative deepening search.

Related Work. Techniques similar to the ones presented in this thesis are used in different contexts, including (*i*) applications working with arbitrary precision integers, (*ii*) DSP compilers, and (*iii*) special-purpose hardware circuit design.

Optimizing effective address generation code is a very important topic for *digital signal processor* (DSP) compilers, as the compute intensive codes executed often spent a considerable amount of their runtime for effective address generation. State-of-the-art compilation techniques for DSPs [57, 86] focus on utilizing special addressing modes offered by modern DSPs but absent on most general purpose processors. These addressing modes include the update-forms like the ones offered by the PowerPC 440 FP2, but also other variants useful for accessing unit-strided arrays, like auto-increment/auto-decrement forms.

Multiple constant multiplication (MCM), i. e., the multiplication of values by a series of integer or fixed-point constants, is highly relevant in special-purpose hardware design [83]. To minimize the total hardware area, the strength of multiple multiplications is reduced, such that the area required by “cheap” basic hardware building-blocks (adders and arithmetic shifters) is minimized.

Because of hardware properties of the IBM PowerPC 440 FP2 processor on one hand and characteristics of the codes targeted by the MAP tool chain on the other hand, these techniques can neither be directly employed nor easily adapted.

Like many DSPs, the IBM PowerPC 440 FP2 has addressing-mode restrictions and update-form addressing modes. Unlike codes for DSPs, however, the input codes of the MAP compiler, like the codes produced by the program generator of Fftw, extensively use memory access operations having a variable stride. Also, the PowerPC 440 FP2 has a much larger number of integer registers to be used as address registers than DSPs.

MCM approaches would be usable for optimizing multiple variably strided accesses. However, it is unclear they can be adapted to (*i*) utilize DSP-like update-form addressing modes, (*ii*) deal with integer register file limits, and (*iii*) exploit composite arithmetic instructions (like the shift+add instruction `lea` present on x86 and AMD64).

An effective solution for utilizing update-form addressing modes could be the extensive adaptation of the order of array access operations. However, this would—while drastically reducing the effective address generation costs—completely break the high-level schedule produced by `genfft`, leading to poor locality of reference and an increased number of (SIMD) floating-point register spills and reloads.

6.1.3 Optimizing Procedure Prolog and Epilog Code

An application binary interface (ABI) describes the object-code level interfaces between (*i*) applications and the operating system, (*ii*) applications and libraries, and (*iii*) different modules of an application.

An important part of any ABI is the calling convention to be employed when invoking procedures residing in separately-compiled compilation units (program modules). This calling convention specifies (*i*) how parameters are passed to procedures, (*ii*) how values are returned from procedures, (*iii*) sets of volatile (caller-saved) registers, and (*iv*) sets of non-volatile (callee-saved) registers.

Callee-saved registers may be altered by the invoked procedure, but only if the called procedure restores their original state before returning control to the calling procedure. Caller-saved registers are freely available as temporary storage to the invoked procedure. Procedure prolog code sets up the processor state (registers and flags), such that the procedure body may execute, which includes saving the callee-saved registers used within the procedure body. Procedure epilog code restores the processor state according to the requirements of the calling convention, which includes restoring all callee-saved registers, whose content has been altered by some part of the procedure body, to their original content before returning from the procedure.

The maximum size of the prolog and epilog code required for fulfilling the procedure calling convention as defined by the application binary interface is constant. Still, optimizing that code is worth while, because of a number of reasons. (*i*) Depending on the application binary interface to be used, the actual constant may be quite large, particularly decreasing the performance of small and middle-sized procedures. (*ii*) The MAP tool chain does not directly support complex control-flow, and compiles each basic block as a separate procedure, which raises the total number of procedure calls. (*iii*) Optimizing prolog/epilog code both minimizes code size and minimizes runtime.

Existing approaches aiming at a minimization of the procedure calling overhead are (*i*) using adapted calling conventions, possibly in combination with (*ii*) performing whole-program optimization. Both techniques cannot be applied in the present context, not because of limitations of the MAP tool chain, but rather because they would require opening up existing proprietary closed-source compilers. But even for open-source compilers, these approaches are infeasible in practice, due to the amount of development and maintenance work required.

The PowerPC ABI defines approximately one half of the logical registers to be callee-saved. If some procedure uses all integer and all floating-point (SIMD) registers, it has to save and restore 19 integer and 18 floating-point registers, amounting to a total of 74 auxiliary load/store instructions for the procedure prolog and epilog. As the PowerPC 440 FP2 can execute at most one load or store instruction per cycle, this task requires up to 74 extra cycles.

For minimizing the code size, using two separate procedures for prolog and epilog, hand-coded in assembly language, is the method of choice. However, this is not a good solution with regard to code speed, which is the main objective of the MAP tool chain. For minimizing the total runtime spent for executing a procedure, it is better to merge the prolog and epilog load/store instructions with the procedure body code. This increases code size, but allows the instruction scheduler to hide a large part of the costs by interleaving prolog/epilog instructions with floating-point instructions of the procedure body.

Minimizing the Floating-Point Prolog/Epilog Size. For floating-point registers, the register allocator generously assigns caller-saved registers (`f0` to `f13`), but sparingly assigns callee-saved registers (`f14` to `f31`). As a result, callee-saved registers are only used if the maximum register pressure requires them to be used.

Minimizing the Integer Prolog/Epilog Size. Minimizing the number of integer registers used and minimizing the number of auxiliary integer instructions for effective address calculation are conflicting goals. The MAP backend performs search for effective address generation code not only once but several times, and then picks from all generated codes the one that attains the best compromise between both optimization criteria.

Elimination of Dead Integer Code. The combined instruction selection and register allocation for integer code calculating effective addresses has a limited optimization focus, due to the problem complexity. Because of optimization boundaries, dead code may be inserted and update-forms for load/store instructions may be used unnecessarily, which can impede following compilation stages like instruction scheduling. Although this situation rarely occurs in practice, the backend removes the resulting dead code and also simplifies load/store instructions that unnecessarily use update-forms.

Minimizing Instruction Scheduling Constraints. Choices made by resource allocation may inadvertently limit the amount of prolog/epilog instructions that can be hidden. To improve code quality without unnecessarily increasing the complexity of the other allocation stages, the MAP backend tries to promote the use of caller-saved registers near the beginning and near the end of procedures. This increases the likelihood that—after instruction scheduling—prolog/epilog code is executed in parallel with the inner parts of a procedure, thus effectively hiding most of the cost associated with the procedure call.

To promote the use of caller-saved registers near the beginning of a procedure, the register allocator first uses caller-saved registers, before it starts using callee-saved registers.

Promoting the use of caller-saved registers near the end of a procedure is achieved by partially reallocating integer and floating-point registers, substituting callee-saved with caller-saved registers, such that the life-spans of the respective registers do not overlap. The procedure quickly reaches a fixed point, i. e., it runs out of caller-saved registers. Thus, it leaves most of the prior allocation unchanged.

Reallocation of floating-point and integer registers is very similar, with one noticeable exception. For integer registers, the reallocator assumes that instructions using the same register as input and as output are load/store instructions in update-form. The reallocator adds an additional equality constraint, asserting that these particular instructions have the same form after reallocation has finished, i. e., their sets of input and output registers still overlap. For floating-point registers, this assumption is not made.

6.2 Scheduling

The MAP backend implements a set of different schedulers, covering a wide range from domain-specific high-level scheduling to target-processor specific low-level code reordering. Within the tool chain, both high-level and the medium-level scheduler precede register allocation.

6.2.1 High-Level Scheduling

Two high-level schedulers are part of the MAP backend. Both schedulers aim at a minimization of register pressure.

Scheduler HL1. The first high-level scheduler is directly derived from the scheduler of `genfft`, the program generator of FFTW. It is implemented as an FFT specific topological sort of the computation DAG and uses a "global view" to enhance locality of access, thereby minimizing variable life-span [28, 31].

Scheduler HL2. The second high-level scheduler realizes a series of bubble-sort like local code reordering strategies, trying to further reduce register pressure. This scheduler works particularly well for DAGs exhibiting a non-regular structure like SIMD-vectorized FFT DAGs.

6.2.2 Medium-Level Scheduling

Both high-level schedulers of the MAP backend do not utilize any information about instruction latencies, solely aiming at minimizing register pressure, which may introduce too many instruction scheduling constraints.

Minimally adapting (i. e., staying as close as possible to the original) code emitted by high-level schedulers prior to register allocation to take instruction latencies into account can increase both register pressure and performance. It is beneficial if the target processor has (*i*) a small number of logical registers, (*ii*) a small number of functional units operating in parallel, (*iii*) high instruction latencies, or (*iv*) no out-of-order execution support.

The actual impact of these criteria on performance depends on various properties of the input code, like dependency chain lengths, the number of different constants used, the number of temporary variables used, variable life-span, etc. Also, these criteria are clearly interdependent, e. g., logical registers can be locked up for a considerable amount of time if long latency instructions are used or the number of functional units operating in parallel is small.

The FP2 instruction set operates on a relatively large register file, comprising 32 vector registers. Large SIMD codes, as produced by program generators, operate on several hundreds of temporary variables and on a few hundred numerical constants. SIMD vectorization can significantly increase the number of constants and decrease the average number of uses per constant. FMA extraction may increase the actual amount of work being done and can have an adverse effect on register pressure, as it (*i*) increases the number of (SIMD) constants and (*ii*) raises the number of instructions requiring three (as opposed to two) input operands, which contributes to longer life-spans of SIMD constants. Analysis of the register allocation process shows that, with large SIMD codes, up to 50% of the register file is used for holding numerical SIMD constants, but is not used for holding input array elements or temporary values calculated by some preceeding instruction.

The IBM PowerPC 440 FP2 directly meets criterion (*ii*), as it is able to execute at most one (SIMD) floating-point instruction per cycle.

Both the arithmetic SIMD instructions used in signal transform codes for calculation (addition, subtraction, multiplication, and fused multiply-add) and SIMD load/store operations are fully pipelined on FP2. These instructions have relatively short latencies of five and four (assuming a data is present in L1 cache) cycles, respectively. But, depending on the data dependencies within the code, even such short latencies may quickly add up to large amounts. For instance, the multiplication of two complex numbers usually takes 4 multiplications and 2 additions, which can be mapped to 3 general purpose SIMD floating-point instructions or to 2 Blue Gene specific SIMD fused-multiply add instructions. Assuming the best case—all data is present in the required layout in registers, all instructions are issued as soon as possible, and there are no pipeline stalls—these two dependent Blue Gene SIMD FMAs produce an output after 10 cycles.

The PowerPC 440 FP2 offers only very limited support for out-of-order execution, allowing for instructions dispatching to different functional units to be reordered, but keeping the execution of instructions executing on the same functional unit strictly in-order. So the processor can, for instance, dynamically adapt the order of some load-store instruction with regard to an independent (SIMD) floating-point (or integer) instruction, but is not able to adapt the order of two independent (SIMD) floating-point instructions.

Scheduler ML. This scheduler aims at a good compromise between minimizing register pressure and minimizing instruction scheduling constraints. It reorders instructions to take instruction latencies into account, increasing the register pressure. Inspired by stable sorting algorithms, this scheduler avoids all dispensable instruction movement, preserving the instruction order obtained by high-level (domain-specific) scheduling as far as possible.

6.2.3 Low-Level Scheduling

The backend comprises two low-level schedulers, both of which specifically address execution properties of the target processor.

Scheduler LL1. The first low-level scheduler implements the list-scheduling algorithm [63, 78], based on an in-order, super-scalar execution model of the target processor and handles both pipelined and non-pipelined instructions well. The scheduler provides a runtime estimate, that is can be used to control an optimization loop [52] of successive instruction scheduling and register reallocation.

Execution models include information about (*i*) instruction latencies, (*ii*) instruction throughput, (*iii*) issuing and decoding constraints, (*iv*) the mapping of instructions to functional units, and (*v*) register forwarding features. The current version of the compiler includes various machine models for the IBM PowerPC 440 FP2 and the IBM PowerPC 970 (G5) processors.

Execution models may include assumptions about the presence of data in caches, e. g., specifying that all constants are present in L1 cache, while all input and output arrays reside in L2 cache. Especially in the context of automatic performance tuning software, this looks like a promising direction of future improvement of the backend. For the sake of simplicity, however, all currently implemented models assume that all data accessed by loads and stores reside in L1 cache.

Scheduler LL2. The second low-level scheduler addresses target-processor specific instruction decoding, issuing, and completion constraints. Depending on the target architecture to generate code for, LL2 realizes different strategies.

For the IBM PowerPC 440 FP2, the backend does not perform any decoding or issueing related reordering following the scheduler `LL1`.

For the IBM PowerPC 970 (G5) processor, LL2 reorders instructions within a decoding group consisting of up to four instructions to avoid issue queue congestion due to unfavorable clustering within decoding windows.

In an experimental version of the backend for x86 compatible machines [49], the scheduler tries to deal with address generation interlocks. An address generation interlock occurs whenever a memory access instruction involving complex address calculation immediately precedes a memory access instruction having simple address calculation. As on x86 all memory accesses occur in-order, the latter instruction is blocked until the effective address of the prior instruction has been calculated, even if the two instructions access completely different parts of the memory, e. g., if the first one accesses data in some array, while the latter one accesses the stack. The x86 backend tries to avoid address generation interlocks of that type by moving memory access instructions having simple address calculation over immediately neighboring memory instructions having complex address calculation, after list-based instruction scheduling has been performed.

Parameterization of the Low-Level List Scheduler

The low-level list scheduler LL1 is customized towards some particular architecture by using the following directives, which are ground PROLOG terms, as defined by the predicate `is_llschedulerDirective` shown in Fig. 6.5.

```
is_llschedulerDirective(pickingHeuristics(P)) :-      % selection heuristics
        is_pickingHeuristics(P).
is_llschedulerDirective(maxInstrsPerCycle(M)) :-      % max # of instrs/cycle
        is_positiveInteger(M).
is_llschedulerDirective(resources(Rs)) :-             % available resources
        is_resources(Rs).
is_llschedulerDirective(favor(Fs)) :-                 % scheduling bias
        is_favors(Fs).
is_llschedulerDirective(favorReleasers(Rs)) :-        % scheduling bias
        is_favorReleasers(Rs).
is_llschedulerDirective(scheduling(Cs)) :-            % scheduling clauses
        is_schedulingClauses(Cs).                     %   for instructions
```

Figure 6.5: Low-Level List Scheduler Directives. The directives specify the primary selection heuristics, the number of instructions that can be issued each cycle, the available resources, various scheduling biases, and scheduling properties of all supported instructions.

The directives shown in Fig. 6.5 have the following meaning.

`pickingHeuristics(P)` sets the used selection heuristic to P, which may be either *original_order* or *critical_path*. If P equals *original_order*, the scheduler selects among all instructions ready for issue the instruction that occurs the earliest in the original program text. If P equals *critical_path*, the scheduler selects among all instructions ready for issue the instruction having the longest critical path.

`maxInstrsPerCycle(M)` specifies the maximum number M of instructions that can be fetched, decoded, issued, executed, and completed each cycle. Reasonable values are $M=1$ for non-super-scalar execution, $M=2$ for dual-issue machines like the IBM PowerPC 440 FP2, $M=3$ for triple-issue machines like the AMD Athlon, the Intel Pentium III, and the Intel Pentium 4, and $M=4$ for quadruple-issue machines like most modern super-scalar RISC processors, including the IBM PowerPC 970 (G5).

`resources(Rs)` declares that the *Rs* includes all execution resources available. If the given list contains duplicates, then multiple resources of that particular type are assumed to be available. Resources used in the execution model may either be functional units, or other finite (pseudo) resources, like a limited number of data-paths present on the target processor.

`favor(Fs)` and `favorReleasers(Rs)` can be used for fine-tuning the selection of the next ready instruction to be issued, providing control over situations, where the primary heuristics for selecting the next instruction to be issued does not uniquely define any instruction to be chosen. For instance, with the *critical_path* heuristics, it is reasonable to favor instructions that mark the last use of some value, freeing a logical register.

`favor(Fs)` with *Fs* being a list of entries *i_howmuch(I,N)* adapts the bias for selecting ready instructions matching the pattern *I* to the integer *N*.

`favorReleasers(Rs)` with *Rs* being a list of entries *d_howmuch(D,N)* further adapts the bias of all instructions that mark the end of the life-span of some operand of type *D* by *N*.

`scheduling(Cs)` specifies a list of clauses *Cs* describing the execution requirements and effects of instructions matching some specific pattern. Individual clauses satisfy the predicate `is_schedulingClause` shown in Fig. 6.6. Auxiliary predicate definitions are shown in Fig. 6.7.

```
is_schedulingClause(i_annoreq_lats_xsrcs_xdsts_res(W,A,Ls,XS,XD,R) :-
        is_instrOrPair(W),
        is_annotations(A),
        is_latencies(Ls),
        is_resources(XS),
        is_resources(XD),
        is_requiredResources(R).
is_schedulingClause(i_lats_res(W,Ls,R)) :-
        is_instrOrPair(W),
        is_latencies(Ls),
        is_requiredResources(R).
```

Figure 6.6: List Scheduling Clauses. A clause defines the latency of some producer (or alternatively, between some producer and some consumer) and the resources required for executing the producer. Annotations and pseudo-operands may be used to constrain the scheduling.

`i_annoreq_lats_xsrcs_xdsts_res(I,A,Ls,XS,XD,R)` is a scheduling clause denoting that all instructions matching the pattern *I* and having the annotations *A* require the execution resources *R* to produce some particular output after some designated time *Ls*, possibly reading (and writing) some auxiliary extra operands, specified by *XS* and *XD*, respectively. If *I* is no instruction pattern, but of the form *s_d(S,D)*, with *S* and *D* being instruction patterns, then the clause is applied whenever pairs of producers and consumers matching the pair *(S,D)* is encountered, which allows modeling forwarding paths from some functional unit to another.

As instructions may produce more than one visible output, the list *Ls* includes, for each output operand type, entries of the form *dst_lat(D,L)*, denoting that operands of type D are written after L cycles. A common case of instructions producing multiple outputs are load instructions with a "update" addressing mode, which write the respective floating-point (or SIMD) register after a few (say 5) cycles, but update the input integer register, which was used as the base register, after one cycle.

The auxiliary extra operands *XS* and *XD* allow to preserve parts of the order of the original input code.

Required execution resources R have the form defined by the predicate `is_requiredResources`, shown in Fig. 6.7.

```
is_instrOrPair(I) :-                          % From instruction I
        is_instr(I).                          %   to any other
is_instrOrPair(s_d(S,D)) :-                   % From instruction S
        is_instr(S),                          %   to instruction D
        is_instr(D).

is_latency(dst_lat(D,L)) :-                   % Operand D is written
        is_resource(D),                       %   after L cycles
        is_nonNegativeInteger(L).

is_requiredResources(none).                   % No resources required
is_requiredResources(and(Rs)) :-              % All resources in Rs
        is_requiredResourcesList(Rs).         %   are required
is_requiredResources(or(Rs)) :-               % At least one resource
        is_requiredResourcesList(Rs).         %   in Rs is required
is_requiredResources(unit_blocked(R,N)) :-    % Resource R is blocked
        is_resource(R),                       %   for N cycles
        is_nonNegativeInteger(N).
```

Figure 6.7: List Scheduling Clause Auxiliary Predicates. These predicates are used by definition of scheduling clauses.

For the commonly occuring case that some clause does not use any annotations or extra operands, a short form `i_lats_res(I,Ls,R)` is offered, which is equivalent to *i_annoreq_lats_xsrcs_xdsts_res(I,[],Ls,[],[],R)*.

Required execution resources are usually terms of the form *unit_blocked(U,N)*, specifying that the respective instruction blocks the resource U for N cycles. This allows the scheduler to uniformly model both pipelined ($N=1$) and non-pipelined ($N>1$) instructions. Execution resources may also be compound resources, consisting of several alternatives (*or*) or a selection of multiple resources (*and*). If some instruction of some type does not require any resources at all, except for the decoding and issuing slot, then the constant *none* is used.

List Scheduling Parameters for Blue Gene Processors

A simple model (shown in Fig. 6.8) is used for scheduling SIMD code that does not have explicit integer code for effective address generation.

```
maxInstrsPerCycle(2).                                  % dual-issue
resources([lsu, fp]).
scheduling([
   i_annoreq_lats_xsrcs_xdsts_res(                     % SIMD data load
      i(f2v_ld2(_),[ae(_,_,_)],_,_), [],
         [dst_lat(fp2r(_),4)], [memstate], [],
         unit_blocked(lsu,1)),
   i_annoreq_lats_xsrcs_xdsts_res(                     % SIMD data store
      i(f2v_st2(_),_,[ae(_,_,_)],_), [],
         [], [], [memstate],
         unit_blocked(lsu,1)),
   i_lats_res(                                         % other SIMD load
      i(f2v_ld2(_),_,_,_),
         [dst_lat(fp2r(_),4)], unit_blocked(lsu,1)),
   i_lats_res(                                         % SIMD spill
      i(f2v_st2(_,_),_,_,_),
         [], unit_blocked(lsu,1)),
   i_lats_res(                                         % SIMD FP op
      i(f2v_N(_),_,_,_),
         [dst_lat(fp2r(_),5)], unit_blocked(fp,1)),
   i_lats_res(_, [], none)                             % (all other)
]).
```

Figure 6.8: Scheduling Parameters for the IBM PowerPC 440 FP2. This simple model only covers SIMD load, SIMD store, and SIMD floating-point instructions. Clauses for SIMD data load and store instructions accessing some array *ae(_,_,_)* are kept in the original order, by using a pseudo-operand, *memstate*. Instructions not covered by any of these clauses are assumed to require no processor resources, except for the decoding and issuing slot.

Once effective address code has been generated, more elaborate parameterizations than the ones shown in Fig. 6.8 are used, to adequately cover integer and (scalar and SIMD) floating-point instructions. The actual OCAML source code of the low-level list scheduler LL1 is listed in Appendix A.4.

Chapter 7

Compilation Examples

Throughout the compilation process, the MAP tool chain represents input and intermediate code as human-readable ground (variable-free) PROLOG terms, as defined in the Sections 7.1 and 7.2. Section 7.3 presents several examples of code processing, keeping track of all relevant stages of compilation.

7.1 Input Code Representation

The input of the MAP tool chain is a scalar computation DAG, corresponding to a single-entry single-exit block of straight line code in *static single assignment* (SSA) form. Input codes are ground PROLOG terms as defined by the predicate `is_procedure` (see Fig. 7.1). To produce output in this representation—instead of C code—the code generator of FFTW had to be adapted slightly. Other program generators could be adapted in a similar way, requiring only minor effort.

```
is_procedure(procedure1(Name,Arrays,Args,Constants,Instrs)) :-
        is_string(Name),
        is_arraydecls(Arrays),
        is_arguments(Args),
        is_constdecls(Constants),
        is_instrs(Instrs).
```

Figure 7.1: Input Representation of a Procedure.

A procedure input to the MAP tool chain comprises a procedure name, several declarations and the procedure body. The declarations include information about (*i*) all input and output arrays used, (*ii*) the procedure's arguments, and (*iii*) all constants used by instructions of the procedure body. The procedure body is a list of scalar instructions.

All scalar constants used in the procedure body are declared by a list of terms, as defined by `is_constdecl` (see Fig. 7.2). The textual representation of the constants specified may have arbitrary precision.

```
is_constdecl(constDecl1(P,TextualRepresentation)) :-
        is_precision(P),
        is_string(TextualRepresentation).
```

Figure 7.2: Constant Declarations.

Arrays are declared by a list of terms, as defined by `is_arraydecl` (see Fig. 7.3). Note that array properties like stride are declared as properties of that particular array, not of the particular array access.

```
is_arraydecl(arraydecl(ArrayName,Stride,Size,Precision)) :-
        is_string(ArrayName),
        is_stride(Stride),
        is_positiveInteger(Size),
        is_precision(Precision).

is_precision(single).
is_precision(double).

is_stride(variable(Name)) :-
        is_string(Name).
is_stride(fixed(I)) :-
        is_positiveInteger(I).
```

Figure 7.3: Array Declarations.

The instructions of the procedure body to be compiled are a list of terms, as defined by `is_instr` (see Fig. 7.4). All instructions operate on variables of type *f(_)*. Source operands of store instructions and binary instructions are augmented *t(K, V)*, denoting that the contents of the variable V are multiplied by the K-th constant before the actual operation. If K equals 0, then the multiplication is omitted.

```
is_instr(f_Load(S,D)) :-
        is_memoryLocationRW(S),
        is_variable(D).
is_instr(f_Store(S,D)) :-
        is_augmentedVariable(S),
        is_memoryLocationRW(D).
is_instr(f_BinOp(Op,S1,S2,D)) :-
        member_of(Op, [add,sub,mul]),
        is_augmentedVariable(S1),
        is_augmentedVariable(S2),
        is_variable(D).

is_variable(f(I)) :-
        is_nonNegativeInteger(I).

is_augmentedVariable(t(K,V)) :-
        is_nonNegativeInteger(K),
        is_variable(V).
```

Figure 7.4: Scalar Instructions.

7.2 Intermediate Code Representation

For all internal processing within the tool chain, the MAP compiler uses the intermediate representation defined by the predicate `is_procedureIR`, shown in Fig. 7.5.

```
is_procedureIR(procedure(Name,Arrays,Args,Constants,Instrs)) :-
        is_atom(Name),
        is_arrayDeclsIR(Arrays),
        is_atoms(Args),
        is_constDeclsIR(Constants),
        is_instrsIR(Instrs).

is_arrayDeclIR(array_stride_multitude_precision(A,S,M,P)) :-
        is_atom(Array),
        is_strideIR(S),
        is_positiveInteger(M),
        is_precision(P).

is_strideIR(S) :-
        is_atom(S).
is_strideIR(I) :-
        is_positiveInteger(I).

is_constDeclIR(f1_const(Id,P,C)) :-
        is_atom(Id),
        is_precision(P),
        is_atom(C).
is_constDeclIR(f2_const(Id,P,C_lo,C_hi)) :-
        is_atom(Id),
        is_precision(P),
        is_atom(C_lo),
        is_atom(C_hi).

is_instrIR(i(Op,Srcs,Dsts,Annotations)) :-
        is_op(Op),
        is_operands(Srcs),
        is_operands(Dsts),
        is_annotations(Annotations).
```

Figure 7.5: Intermediate Representation of a Procedure.

The part of the definition of intermediate representation shown in Fig. 7.5 is the same for all target architectures.

Defined operations and pseudo-operations, operands, and supported annotations differ from target to target. For Blue Gene systems, the MAP tool chain uses the definition shown in Fig. 7.6.

```
is_op(f2v_ld1).                         % load operations
is_op(f2v_ld1(LH)) :-
        is_loHi(LH).
is_op(f2v_ld1(P,LH)) :-
        is_precision(P), is_loHi(LH).
is_op(f2v_ld1acc(Op)) :-
        is_addSub(Op).
is_op(f2v_ld2(M)) :-
        is_parCross(M).
is_op(f2v_ld2(P,M)) :-
        is_precision(P),
        is_parCross(M).
is_op(f2v_st1(LH)) :-                   % store operations
        is_loHi(LH).
is_op(f2v_st1(P,LH)) :-
        is_precision(P),
        is_loHi(LH).
is_op(f2v_st2(M)) :-
        is_parCross(M).
is_op(f2v_st2(P,M)) :-
        is_precision(P),
        is_parCross(M).
is_op(f2v_N(SimdOp)) :-                 % n-ary arithmetic operation
        is_simdop(SimdOp).

is_simdop(paddsub(Op1,Op2)) :-          % addadd|addsub|subadd|subsub
        is_addSub(Op1),
        is_addSub(Op2).
is_simdop(copy).                        % unary copy
is_simdop(swap).                        % unary swap (=cross)
is_simdop(chs(LH)) :-                   % unary sign change
        is_loHi(LH).
is_simdop(pmul).                        % parallel multiplication
is_simdop(xmul).                        % cross multiplication
is_simdop(xpmul).                       % cross-copy primary multiplication
is_simdop(xsmul).                       % cross-copy secondary multiplication
is_simdop(pmadd).                       % parallel multiply-add
is_simdop(pmsub).                       % parallel multiply-sub
is_simdop(pnmadd).                      % parallel negative multiply-add
is_simdop(pnmsub).                      % parallel negative multiply-sub
is_simdop(xmadd).                       % cross multiply-add
is_simdop(xmsub).                       % cross multiply-sub
is_simdop(xnmadd).                      % cross negative multiply-add
is_simdop(xnmsub).                      % cross negative multiply-sub
is_simdop(xcsnmsub).                    % cross copy-secondary negative multiply-sub
is_simdop(xcxnpma).                     % cross complex nsub-primary multiply-add
is_simdop(xcxnsma).                     % cross complex nsub-secondary multiply-add

is_loHi(lo).                            % lower part of 2-way SIMD vector
is_loHi(hi).                            % higher part of 2-way SIMD vector

is_parCross(p).                         % unary SIMD copy
is_parCross(x).                         % unary SIMD swap

is_addSub(add).                         % scalar addition
is_addSub(sub).                         % scalar subtraction

is_operand(f2vr(_)).                    % SIMD FP variable
is_operand(fvr(_)).                     % Scalar FP variable
is_operand(fp2r(_)).                    % FP2 SIMD register
is_operand(r(_)).                       % Integer register
is_operand(ae(_,_,_)).                  % 1D array element
is_operand(c(_)).                       % Numerical constant

is_annotation(s(_)).                    % Uses hidden source
is_annotation(d(_)).                    % Uses hidden destination
```

Figure 7.6: Blue Gene Specific Parts of the Intermediate Representation.

7.3 FFT Example Codes

This section presents several codes and describes their way through the MAP tool chain, demonstrating the effects of the most important compilation stages.

All codes have been automatically generated by an adapted version FFTW's program generator `genfft` that produces output in MAP's input format instead of plain C code. In comparison with codes occuring in practice, the presented codes are all relatively short. However, they differ considerably, and thus illustrate different aspects of the functioning of the MAP compiler.

7.3.1 3-point Forward Complex FFT (`fn_3`)

The first code `fn_3` is a forward 3-point complex no-twiddle FFT codelet.

Example `fn_3`: Scalar code produced by `genfft`

```
procedure1("fftw_no_twiddle_3",
           [
            arrayDecl("input", variable("istride"), 2, double),
            arrayDecl("output", variable("ostride"), 2, double)
           ],
           [ "input", "output", "istride", "ostride" ],
           [
            constDecl1(double, "+0.5"),
            constDecl1(double, "+0.866025403784438646763723170752936183471402627")
           ],
           [
            f_Load(ae("input",0,0), f(1)),
            f_Load(ae("input",0,1), f(16)),
            f_Load(ae("input",1,0), f(2)),
            f_Load(ae("input",2,0), f(3)),
            f_BinOp(add, t(0,f(2)), t(0,f(3)), f(4)),
            f_BinOp(sub, t(0,f(3)), t(0,f(2)), f(14)),
            f_Load(ae("input",1,1), f(8)),
            f_Load(ae("input",2,1), f(9)),
            f_BinOp(sub, t(0,f(8)), t(0,f(9)), f(10)),
            f_BinOp(add, t(0,f(8)), t(0,f(9)), f(17)),
            f_BinOp(add, t(0,f(1)), t(0,f(4)), f(5)),
            f_Store(t(0,f(5)), ae("output",0,0)),
            f_BinOp(sub, t(0,f(1)), t(1,f(4)), f(7)),
            f_BinOp(sub, t(0,f(7)), t(2,f(10)), f(12)),
            f_BinOp(add, t(0,f(7)), t(2,f(10)), f(13)),
            f_Store(t(0,f(12)), ae("output",2,0)),
            f_Store(t(0,f(13)), ae("output",1,0)),
            f_BinOp(add, t(0,f(16)), t(0,f(17)), f(22)),
            f_Store(t(0,f(22)), ae("output",0,1)),
            f_BinOp(sub, t(0,f(16)), t(1,f(17)), f(19)),
            f_BinOp(add, t(2,f(14)), t(0,f(19)), f(20)),
            f_BinOp(sub, t(0,f(19)), t(2,f(14)), f(21)),
            f_Store(t(0,f(20)), ae("output",1,1)),
            f_Store(t(0,f(21)), ae("output",2,1))
           ]
).
```

The modified version of `genfft` also produces C code, which includes declarations and possibly stub code calling code compiled by the MAP tool chain.

```
extern void fftw_no_twiddle_3(const fftw_complex *, fftw_complex *, int, int);

fftw_codelet_desc fftw_no_twiddle_3_desc = {
        "fftw_no_twiddle_3",
        (void (*)()) fftw_no_twiddle_3,
        3,
        FFTW_FORWARD,
        FFTW_NOTW,
        67,
        0,
        (const int *) 0,
};
```

Example fn_3: Vectorization

Vectorization transforms the scalar code into the following SIMD vector code. Note that the instructions are in no particular order, as the instruction sequence represents a DAG.

```
procedure(fftw_no_twiddle_3,
          [
           array_stride_multitude_precision(input,istride,2,double),
           array_stride_multitude_precision(output,ostride,2,double)
          ],
          [ input, output, istride, ostride ],
          [
           f2_const('0',double,'+0.5','+0.5'),
           f2_const('1',double,'+0.866025403784438646763723170752936183471402627',
                               '+0.866025403784438646763723170752936183471402627')
          ],
          [
           i(f2v_N(paddsub(sub,sub)),[f2vr(18),f2vr(24)],[f2vr(25)],[]),
           i(f2v_N(swap),[f2vr(25)],[f2vr(1)],[]),
           i(f2v_N(pmul),[f2vr(20),f2vr(23)],[f2vr(24)],[]),
           i(f2v_N(swap),[f2vr(8)],[f2vr(23)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(18),f2vr(21)],[f2vr(22)],[]),
           i(f2v_N(swap),[f2vr(22)],[f2vr(0)],[]),
           i(f2v_N(pmul),[f2vr(20),f2vr(19)],[f2vr(21)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(20)],[]),
           i(f2v_N(swap),[f2vr(8)],[f2vr(19)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(15),f2vr(17)],[f2vr(18)],[]),
           i(f2v_N(pmul),[f2vr(16),f2vr(12)],[f2vr(17)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(16)],[]),
           i(f2v_N(swap),[f2vr(5)],[f2vr(15)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(13),f2vr(12)],[f2vr(14)],[]),
           i(f2v_N(swap),[f2vr(14)],[f2vr(2)],[]),
           i(f2v_N(swap),[f2vr(5)],[f2vr(13)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(10),f2vr(11)],[f2vr(12)],[]),
           i(f2v_N(swap),[f2vr(3)],[f2vr(11)],[]),
           i(f2v_N(swap),[f2vr(4)],[f2vr(10)],[]),
           i(f2v_N(chs(hi)),[f2vr(9)],[f2vr(8)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(7),f2vr(6)],[f2vr(9)],[]),
           i(f2v_N(swap),[f2vr(4)],[f2vr(7)],[]),
           i(f2v_N(swap),[f2vr(3)],[f2vr(6)],[]),
           i(f2v_ld2(p),[ae(input,0,0)],[f2vr(5)],[]),
           i(f2v_ld2(p),[ae(input,1,0)],[f2vr(4)],[]),
           i(f2v_ld2(p),[ae(input,2,0)],[f2vr(3)],[]),
           i(f2v_st2(p),[f2vr(2)],[ae(output,0,0)],[]),
           i(f2v_st2(p),[f2vr(1)],[ae(output,2,0)],[]),
           i(f2v_st2(p),[f2vr(0)],[ae(output,1,0)],[])
          ]
).
```

In this code, all addition, subtraction, and multiplication operations are carried out by SIMD instructions. Still, there are many auxiliary SIMD instructions (`chs` and `swap`), some redundant instructions, and explicit multiplication instructions (`pmul`) that could be hidden in SIMD FMAs. Highlighted parts of the above code fragment show these instructions.

Example `fn_3`: Peephole Optimization

Peephole optimization performs various local optimizations in this code. For example, all SIMD swap instructions are eliminated, either directly or by merging them with neighboring Blue Gene specific SIMD instructions. The instruction count is further minimized by the extraction of Blue Gene specific SIMD FMA instructions.

```
procedure(fftw_no_twiddle_3,
          [
           array_stride_multitude_precision(input,istride,2,double),
           array_stride_multitude_precision(output,ostride,2,double)
          ],
          [
           input,
           output,
           istride,
           ostride
          ],
          [
           f2_const('0',double,'0.5','0.5'),
           f2_const('1',double,'0.8660254037844386467637232',
                               '0.8660254037844386467637232')
          ],
          [
           i(f2v_st2(p),[f2vr(15)],[ae(output,1,0)],[]),
           i(f2v_st2(p),[f2vr(16)],[ae(output,2,0)],[]),
           i(f2v_ld2(p),[ae(input,0,0)],[f2vr(5)],[]),
           i(f2v_ld2(p),[ae(input,1,0)],[f2vr(4)],[]),
           i(f2v_ld2(p),[ae(input,2,0)],[f2vr(3)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(4),f2vr(3)],[f2vr(11)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(4),f2vr(3)],[f2vr(12)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(5),f2vr(12)],[f2vr(13)],[]),
           i(f2v_st2(p),[f2vr(13)],[ae(output,0,0)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(119)],[]),
           i(f2v_N(xcsnmsub),[f2vr(119),f2vr(12),f2vr(5)],[f2vr(14)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(120)],[]),
           i(f2v_N(xcxnsma),[f2vr(120),f2vr(11),f2vr(14)],[f2vr(15)],[]),
           i(f2v_N(xcxnpma),[f2vr(120),f2vr(11),f2vr(14)],[f2vr(16)],[])
          ]
).
```

The highlighted parts show Blue Gene specific operations extracted by the peephole optimizer.

By this stage of compilation, all modules disregarded an existing instruction order and produces code in no particular order. Beginning with the following stage, the order of the instructions becomes important.

Example `fn_3`: High-Level Scheduling

The high-level schedulers `HL1` and `HL2` produce a topological sort of the code, i. e., they adapt the order of the instructions occuring in the DAG such that they are sequentially executable.

```
procedure(fftw_no_twiddle_3,
          [
           array_stride_multitude_precision(input,istride,2,double),
           array_stride_multitude_precision(output,ostride,2,double)
          ],
          [
           input,
           output,
           istride,
           ostride
          ],
          [
           f2_const('0',double,'0.5',
                                  '0.5'),
           f2_const('1',double,'0.8660254037844386467637232',
                                  '0.8660254037844386467637232')
          ],
          [
           i(f2v_ld2(p),[ae(input,2,0)],[f2vr(3)],[]),
           i(f2v_ld2(p),[ae(input,1,0)],[f2vr(4)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(4),f2vr(3)],[f2vr(12)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(4),f2vr(3)],[f2vr(11)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(119)],[]),
           i(f2v_ld2(p),[ae(input,0,0)],[f2vr(5)],[]),
           i(f2v_N(xcsnmsub),[f2vr(119),f2vr(12),f2vr(5)],[f2vr(14)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(5),f2vr(12)],[f2vr(13)],[]),
           i(f2v_st2(p),[f2vr(13)],[ae(output,0,0)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(120)],[]),
           i(f2v_N(xcxnpma),[f2vr(120),f2vr(11),f2vr(14)],[f2vr(16)],[]),
           i(f2v_st2(p),[f2vr(16)],[ae(output,2,0)],[]),
           i(f2v_N(xcxnsma),[f2vr(120),f2vr(11),f2vr(14)],[f2vr(15)],[]),
           i(f2v_st2(p),[f2vr(15)],[ae(output,1,0)],[])
          ]
).
```

The input and output array elements have been highlighted.

As these schedulers solely aim at the minimization of variable life-spans, they push load instructions down and store instructions up, moving them as close as possible to their respective consumer or producer. While this technique keeps register pressure as small as possible, it completely disregards instruction scheduling constraints.

Example fn_3: Medium-Level Scheduling

Medium-level scheduling takes instruction latencies into account. As a result, load instructions are moved up, while store instructions are moved down slightly.

```
procedure(fftw_no_twiddle_3,
          [
           array_stride_multitude_precision(input,istride,2,double),
           array_stride_multitude_precision(output,ostride,2,double)
          ],
          [
           input,
           output,
           istride,
           ostride
          ],
          [
           f2_const('0',double,'0.5',
                               '0.5'),
           f2_const('1',double,'0.86602540378443864676372 32',
                               '0.86602540378443864676372 32')
          ],
          [
           i(f2v_ld2(p),[ae(input,2,0)],[f2vr(3)],[]),
           i(f2v_ld2(p),[ae(input,1,0)],[f2vr(4)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(119)],[]),
           i(f2v_ld2(p),[ae(input,0,0)],[f2vr(5)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(120)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(4),f2vr(3)],[f2vr(12)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(4),f2vr(3)],[f2vr(11)],[]),
           i(f2v_N(xcsnmsub),[f2vr(119),f2vr(12),f2vr(5)],[f2vr(14)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(5),f2vr(12)],[f2vr(13)],[]),
           i(f2v_st2(p),[f2vr(13)],[ae(output,0,0)],[]),
           i(f2v_N(xcxnpma),[f2vr(120),f2vr(11),f2vr(14)],[f2vr(16)],[]),
           i(f2v_N(xcxnsma),[f2vr(120),f2vr(11),f2vr(14)],[f2vr(15)],[]),
           i(f2v_st2(p),[f2vr(16)],[ae(output,2,0)],[]),
           i(f2v_st2(p),[f2vr(15)],[ae(output,1,0)],[])
          ]
).
```

Instructions that have been moved significantly have been highlighted.

This code still operates on unlimited sets of SIMD floating-point operands, i. e., Prolog terms of the form `f2vr(_)`. In the next compilation step, SIMD floating-point register allocation, these terms will be mapped to logical registers, i. e., Prolog terms of the form `fp2r(_)`.

Example fn_3: Floating-Point Register Allocation

Next, SIMD floating-point registers are allocated. In this example, no spill and reload instructions are required, as all operands fit into the large (32 entry) register file assumed to be present.

```
procedure(fftw_no_twiddle_3,
          [
           array_stride_multitude_precision(input,istride,2,double),
           array_stride_multitude_precision(output,ostride,2,double)
          ],
          [
           input,
           output,
           istride,
           ostride
          ],
          [
           f2_const('0',double,'0.5',
                                 '0.5'),
           f2_const('1',double,'0.86602540378443864676372 32',
                                 '0.8660254037844386467637232')
          ],
          [
           i(f2v_ld2(p),[ae(input,2,0)],[fp2r(0)],[]),
           i(f2v_ld2(p),[ae(input,1,0)],[fp2r(1)],[]),
           i(f2v_ld2(p),[c('0')],[fp2r(2)],[]),
           i(f2v_ld2(p),[ae(input,0,0)],[fp2r(3)],[]),
           i(f2v_ld2(p),[c('1')],[fp2r(4)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(1),fp2r(0)],[fp2r(5)],[]),
           i(f2v_N(paddsub(sub,sub)),[fp2r(1),fp2r(0)],[fp2r(1)],[]),
           i(f2v_N(xcsnmsub),[fp2r(2),fp2r(5),fp2r(3)],[fp2r(2)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(3),fp2r(5)],[fp2r(3)],[]),
           i(f2v_st2(p),[fp2r(3)],[ae(output,0,0)],[]),
           i(f2v_N(xcxnpma),[fp2r(4),fp2r(1),fp2r(2)],[fp2r(6)],[]),
           i(f2v_N(xcxnsma),[fp2r(4),fp2r(1),fp2r(2)],[fp2r(4)],[]),
           i(f2v_st2(p),[fp2r(6)],[ae(output,2,0)],[]),
           i(f2v_st2(p),[fp2r(4)],[ae(output,1,0)],[])
          ]
).
```

In the above example, the newly assigned floating point SIMD registers of the form `fp2r(_)` have been highlighted.

In the MAP tool chain, register allocation for SIMD floating-point instructions precedes integer register allocation and procedure prolog/epilog code generation, as these steps depend on the outcome of floating-point register allocation.

Example fn_3: Effective Address Generation

Now, procedure prolog and epilog code are inserted and code for effective address calculation is generated.

```
procedure(fftw_no_twiddle_3,
          [
           array_stride_multitude_precision(input,istride,2,double),
           array_stride_multitude_precision(output,ostride,2,double)
          ],
          [
           input, output, istride, ostride
          ],
          [
           f2_const(0,double,'0.5','0.5'),
           f2_const(1,double,'0.866025403784438646763723',
                              '0.866025403784438646763723')
          ],
          [
           i(i_copyI(16),[],[r(0)],[]),
           i(i_shiftLeft(4),[r(5)],[r(5)],[]),
           i(i_shiftLeft(4),[r(6)],[r(6)],[]),
           i(i_addIS('Consts__fftw_no_twiddle_3@ha'),[imm(0)],[r(12)],[]),
           i(i_addI('Consts__fftw_no_twiddle_3@l'),[r(12)],[r(12)],[]),
           i(i_add,[r(5),r(5)],[r(2)],[]),
           i(f2v_ld2(double,p),[r(3),r(2)],[fp2r(0)],[s(ae(input,2,0))]),
           i(f2v_ld2(double,p),[r(3),r(5)],[fp2r(1)],[s(ae(input,1,0))]),
           i(f2v_ld2(double,p),[r(12)],[fp2r(2)],[s(const(0))]),
           i(f2v_ld2(double,p),[r(3)],[fp2r(3)],[s(ae(input,0,0))]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(4)],[s(const(16))]),
           i(f2v_N(paddsub(add,add)),[fp2r(1),fp2r(0)],[fp2r(5)],[]),
           i(f2v_N(paddsub(sub,sub)),[fp2r(1),fp2r(0)],[fp2r(1)],[]),
           i(f2v_N(xcsnmsub),[fp2r(2),fp2r(5),fp2r(3)],[fp2r(2)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(3),fp2r(5)],[fp2r(3)],[]),
           i(f2v_N(xcxnpma),[fp2r(4),fp2r(1),fp2r(2)],[fp2r(6)],[]),
           i(f2v_N(xcxnsma),[fp2r(4),fp2r(1),fp2r(2)],[fp2r(4)],[]),
           i(f2v_st2(double,p),[fp2r(3),r(4)],[],[d(ae(output,0,0))]),
           i(i_add,[r(6),r(6)],[r(7)],[]),
           i(f2v_st2(double,p),[fp2r(6),r(4),r(7)],[],[d(ae(output,2,0))]),
           i(f2v_st2(double,p),[fp2r(4),r(4),r(6)],[],[d(ae(output,1,0))])
          ]
).
```

Highlighted parts are (*i*) auxiliary integer instructions needed for effective address calculation and (*ii*) adapted parts of SIMD load/store instructions.

Load and store instructions now fully operate on logical registers. Aliasing information is preserved through the use of annotations. For instance, the annotation `d(ae(output,2,0))` denotes that the second element of array `output` is written by that particular instruction.

Example fn_3: List-based Instruction Scheduling

After all integer and floating-point instructions have been generated, the code is scheduled by the list-scheduling algorithm.

```
procedure(fftw_no_twiddle_3,
          [
           array_stride_multitude_precision(input,istride,2,double),
           array_stride_multitude_precision(output,ostride,2,double)
          ],
          [
           input,
           output,
           istride,
           ostride
          ],
          [
           f2_const(0,double,'0.5',
                             '0.5'),
           f2_const(1,double,'0.866025403784438646763723',
                             '0.866025403784438646763723')
          ],
          [
  /* 0 */  i(i_shiftLeft(4),[r(5)],[r(5)],[]),
  /* 0 */  i(i_addIS('Consts__fftw_no_twiddle_3@ha'),[imm(0)],[r(12)],[]),
  /* 1 */  i(i_add,[r(5),r(5)],[r(2)],[]),
  /* 1 */  i(f2v_ld2(double,p),[r(3),r(5)],[fp2r(1)],[s(ae(input,1,0))]),
  /* 2 */  i(f2v_ld2(double,p),[r(3),r(2)],[fp2r(0)],[s(ae(input,2,0))]),
  /* 2 */  i(i_addI('Consts__fftw_no_twiddle_3@l'),[r(12)],[r(12)],[]),
  /* 3 */  i(f2v_ld2(double,p),[r(3)],[fp2r(3)],[s(ae(input,0,0))]),
  /* 3 */  i(i_copyI(16),[],[r(0)],[]),
  /* 4 */  i(f2v_ld2(double,p),[r(12)],[fp2r(2)],[s(const(0))]),
  /* 4 */  i(i_shiftLeft(4),[r(6)],[r(6)],[]),
  /* 5 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(4)],[s(const(16))]),
  /* 5 */  i(i_add,[r(6),r(6)],[r(7)],[]),
  /* 6 */  i(f2v_N(paddsub(add,add)),[fp2r(1),fp2r(0)],[fp2r(5)],[]),
  /* 7 */  i(f2v_N(paddsub(sub,sub)),[fp2r(1),fp2r(0)],[fp2r(1)],[]),
 /* 11 */  i(f2v_N(xcsnmsub),[fp2r(2),fp2r(5),fp2r(3)],[fp2r(2)],[]),
 /* 12 */  i(f2v_N(paddsub(add,add)),[fp2r(3),fp2r(5)],[fp2r(3)],[]),
 /* 16 */  i(f2v_N(xcxnpma),[fp2r(4),fp2r(1),fp2r(2)],[fp2r(6)],[]),
 /* 17 */  i(f2v_N(xcxnsma),[fp2r(4),fp2r(1),fp2r(2)],[fp2r(4)],[]),
 /* 17 */  i(f2v_st2(double,p),[fp2r(3),r(4)],[],[d(ae(output,0,0))]),
 /* 21 */  i(f2v_st2(double,p),[fp2r(6),r(4),r(7)],[],[d(ae(output,2,0))]),
 /* 22 */  i(f2v_st2(double,p),[fp2r(4),r(4),r(6)],[],[d(ae(output,1,0))])
          ]
).
```

For this short example code, the main visible effect of instruction scheduling is the interleaving of integer instructions and SIMD load instructions. Along with the scheduled sequence of instructions, the list-scheduler also emits the estimated time t when some particular instructions will execute.

Internally (within the tool chain), that estimation is represented as PROLOG terms of the form `t_cplen_i(T,Cpl,I)`, denoting that the instruction I having the critical-path length *Cpl* is assumed to execute in cycle $\#T$. For the purpose of illustration, however, the estimates have been inserted into the previous code fragment in the form of C style comments (`/* t */`).

According to the estimate, the execution of the procedure body takes at least 23 cycles. Quite clearly, this code will spend most of its time waiting for load instructions to complete. This situation is significantly better with larger input codes.

Example `fn_3`: Assembly Output

Finally, the intermediate representation is converted to PowerPC assembly code.

```
.data
     .align 3
Consts__fftw_no_twiddle_3:
     .double 0.5, 0.5
     .double 0.8660254037844386467637232, 0.8660254037844386467637232
.text
     .align 3
     .globl _fftw_no_twiddle_3
_fftw_no_twiddle_3:
     slwi      r5, r5, 4
     addis     r12, 0, Consts__fftw_no_twiddle_3@ha
     add       r2, r5, r5
     lfpdx     f1, r3, r5
     lfpdx     f0, r3, r2
     addi      r12, r12, Consts__fftw_no_twiddle_3@l
     lfpdx     f3, 0, r3
     addi      r0, 0, 16
     lfpdx     f2, 0, r12
     slwi      r6, r6, 4
     lfpdx     f4, r12, r0
     add       r7, r6, r6
     fpadd     f5, f1, f0
     fpsub     f1, f1, f0
     fxcsnmsub f2, f2, f5, f3
     fpadd     f3, f3, f5
     fxcxnpma  f6, f4, f1, f2
     fxcxnsma  f4, f4, f1, f2
     stfpdx    f3, 0, r4
     stfpdx    f6, r4, r7
     stfpdx    f4, r4, r6
     blr
```

7.3.2 6-point Forward Real FFT (`frc_6`)

The example code `frc_6` is a forward 6-point real no-twiddle FFT codelet.

Example `frc_6`: Scalar code produced by `genfft`

```
procedure1(
           "fftw_real2hc_6",
           [
            arrayDecl("input", variable("istride"), 1, double),
            arrayDecl("real_output", variable("real_ostride"), 1, double),
            arrayDecl("imag_output", variable("imag_ostride"), 1, double)
           ],
           [ "input", "real_output", "imag_output", "istride", "real_ostride", "imag_ostride" ],
           [
            constDecl1(double, "+0.866025403784438646763723170752936183471402627"),
            constDecl1(double, "+0.5"),
            constDecl1(double, "-0.866025403784438646763723170752936183471402627")
           ],
           [
            f_Load(ae("input",0,0), f(1)),
            f_Load(ae("input",3,0), f(2)),
            f_BinOp(sub, t(0,f(1)), t(0,f(2)), f(3)),
            f_BinOp(add, t(0,f(1)), t(0,f(2)), f(20)),
            f_Load(ae("input",2,0), f(4)),
            f_Load(ae("input",5,0), f(5)),
            f_BinOp(sub, t(0,f(4)), t(0,f(5)), f(6)),
            f_BinOp(add, t(0,f(4)), t(0,f(5)), f(17)),
            f_Load(ae("input",4,0), f(7)),
            f_Load(ae("input",1,0), f(8)),
            f_BinOp(sub, t(0,f(7)), t(0,f(8)), f(9)),
            f_BinOp(add, t(0,f(7)), t(0,f(8)), f(16)),
            f_BinOp(add, t(0,f(6)), t(0,f(9)), f(10)),
            f_BinOp(sub, t(0,f(9)), t(0,f(6)), f(14)),
            f_BinOp(sub, t(0,f(16)), t(0,f(17)), f(18)),
            f_BinOp(add, t(0,f(17)), t(0,f(16)), f(21)),
            f_Store(t(1,f(14)), ae("imag_output",1,0)),
            f_BinOp(add, t(0,f(20)), t(0,f(21)), f(24)),
            f_Store(t(0,f(24)), ae("real_output",0,0)),
            f_BinOp(sub, t(0,f(20)), t(2,f(21)), f(23)),
            f_Store(t(0,f(23)), ae("real_output",2,0)),
            f_Store(t(3,f(18)), ae("imag_output",2,0)),
            f_BinOp(add, t(0,f(3)), t(0,f(10)), f(13)),
            f_Store(t(0,f(13)), ae("real_output",3,0)),
            f_BinOp(sub, t(0,f(3)), t(2,f(10)), f(12)),
            f_Store(t(0,f(12)), ae("real_output",1,0))
           ]
).
```

The modified version of `genfft` also produces the following C declarations.

```
extern void fftw_real2hc_6(const fftw_real *, fftw_real *, fftw_real *, int, int, int);

fftw_codelet_desc fftw_real2hc_6_desc = {
     "fftw_real2hc_6",
     (void (*)()) fftw_real2hc_6,
     6,
     FFTW_FORWARD,
     FFTW_REAL2HC,
     134,
     0,
     (const int *) 0,
};
```

Example frc_6: Vectorization

Out of this scalar code, the MAP vectorizer produces the following SIMD code.

```
procedure(fftw_real2hc_6,
          [
           array_stride_multitude_precision(input,istride,1,double),
           array_stride_multitude_precision(real_output,real_ostride,1,double),
           array_stride_multitude_precision(imag_output,imag_ostride,1,double)
          ],
          [
           input,
           real_output,
           imag_output,
           istride,
           real_ostride,
           imag_ostride
          ],
          [
           f2_const('0',double,'+0.866025403784438646763723170752936183471402627',
                               '-0.866025403784438646763723170752936183471402627'),
           f2_const('1',double,'+0.5',
                               '+0.5')
          ],
          [
           i(f2v_ld1,[ae(input,0,0),ae(input,3,0)],[f2vr(25)],[]),
           i(f2v_N(swap),[f2vr(25)],[f2vr(22)],[]),
           i(f2v_N(paddsub(add,sub)),[f2vr(22),f2vr(23)],[f2vr(24)],[]),
           i(f2v_N(swap),[f2vr(24)],[f2vr(5)],[]),
           i(f2v_N(swap),[f2vr(22)],[f2vr(23)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(16),f2vr(20)],[f2vr(21)],[]),
           i(f2v_N(swap),[f2vr(21)],[f2vr(6)],[]),
           i(f2v_N(swap),[f2vr(13)],[f2vr(20)],[]),
           i(f2v_N(chs(hi)),[f2vr(19)],[f2vr(18)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(16),f2vr(17)],[f2vr(19)],[]),
           i(f2v_N(swap),[f2vr(18)],[f2vr(2)],[]),
           i(f2v_N(swap),[f2vr(13)],[f2vr(17)],[]),
           i(f2v_N(paddsub(add,sub)),[f2vr(11),f2vr(12)],[f2vr(16)],[]),
           i(f2v_ld1,[ae(input,2,0),ae(input,4,0)],[f2vr(15)],[]),
           i(f2v_N(swap),[f2vr(15)],[f2vr(11)],[]),
           i(f2v_ld1,[ae(input,5,0),ae(input,1,0)],[f2vr(14)],[]),
           i(f2v_N(swap),[f2vr(14)],[f2vr(12)],[]),
           i(f2v_N(paddsub(sub,add)),[f2vr(11),f2vr(12)],[f2vr(13)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(5),f2vr(9)],[f2vr(10)],[]),
           i(f2v_N(swap),[f2vr(10)],[f2vr(0)],[]),
           i(f2v_N(pmul),[f2vr(8),f2vr(6)],[f2vr(9)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(8)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(5),f2vr(6)],[f2vr(7)],[]),
           i(f2v_N(swap),[f2vr(7)],[f2vr(1)],[]),
           i(f2v_st1(hi),[f2vr(4)],[ae(imag_output,2,0)],[]),
           i(f2v_st1(lo),[f2vr(4)],[ae(imag_output,1,0)],[]),
           i(f2v_N(pmul),[f2vr(3),f2vr(2)],[f2vr(4)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(3)],[]),
           i(f2v_st1(hi),[f2vr(1)],[ae(real_output,3,0)],[]),
           i(f2v_st1(lo),[f2vr(1)],[ae(real_output,0,0)],[]),
           i(f2v_st1(hi),[f2vr(0)],[ae(real_output,1,0)],[]),
           i(f2v_st1(lo),[f2vr(0)],[ae(real_output,2,0)],[])
          ]
).
```

This SIMD code still has a lot of auxiliary SIMD instructions that have been inserted by the vectorizer.

Example frc_6: Peephole Optimization

Peephole optimization reduces the instruction count by eliminating auxiliary SIMD instructions and by exploiting special Blue Gene SIMD instructions.

```
procedure(fftw_real2hc_6,
          [
           array_stride_multitude_precision(input,istride,1,double),
           array_stride_multitude_precision(real_output,real_ostride,1,double),
           array_stride_multitude_precision(imag_output,imag_ostride,1,double)
          ],
          [
           input,
           real_output,
           imag_output,
           istride,
           real_ostride,
           imag_ostride
          ],
          [
           f2_const('0',double,'1.0',
                               '1.0'),
           f2_const('1',double,'0.5',
                               '0.5'),
           f2_const('2',double,'0.86602540378443864676372 32',
                               '0.86602540378443864676372 32')
          ],
          [
           i(f2v_st1(lo),[f2vr(107)],[ae(real_output,1,0)],[]),
           i(f2v_st1(hi),[f2vr(107)],[ae(real_output,2,0)],[]),
           i(f2v_st1(lo),[f2vr(18)],[ae(imag_output,2,0)],[]),
           i(f2v_st1(hi),[f2vr(18)],[ae(imag_output,1,0)],[]),
           i(f2v_ld1,[ae(input,0,0),ae(input,3,0)],[f2vr(25)],[]),
           i(f2v_ld1,[ae(input,2,0),ae(input,4,0)],[f2vr(15)],[]),
           i(f2v_st1(hi),[f2vr(104)],[ae(real_output,3,0)],[]),
           i(f2v_st1(lo),[f2vr(104)],[ae(real_output,0,0)],[]),
           i(f2v_ld1,[ae(input,1,0),ae(input,5,0)],[f2vr(12)],[]),
           i(f2v_ld2(p),[c('2')],[f2vr(1025)],[]),
           i(f2v_N(pmul),[f2vr(1025),f2vr(19)],[f2vr(18)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(1026)],[]),
           i(f2v_N(xcxnsma),[f2vr(1026),f2vr(12),f2vr(15)],[f2vr(20)],[]),
           i(f2v_N(xmadd),[f2vr(106),f2vr(1026),f2vr(20)],[f2vr(21)],[]),
           i(f2v_N(xmadd),[f2vr(105),f2vr(1026),f2vr(21)],[f2vr(104)],[]),
           i(f2v_N(xnmsub),[f2vr(106),f2vr(1026),f2vr(20)],[f2vr(19)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(1030)],[]),
           i(f2v_N(xnmsub),[f2vr(21),f2vr(1030),f2vr(105)],[f2vr(107)],[]),
           i(f2v_N(xcxnpma),[f2vr(1026),f2vr(12),f2vr(15)],[f2vr(106)],[]),
           i(f2v_N(xcxnpma),[f2vr(1026),f2vr(25),f2vr(25)],[f2vr(105)],[])
   ]
).
```

In this code, the peephole optimizer has moved all but one SIMD multiplications into neighboring SIMD additions or subtractions, extracting SIMD FMAs.

Example frc_6: High-Level Scheduling

The high-level scheduler uses domain-specific knowledge to produce a topological sort of the computation DAG.

```
procedure(fftw_real2hc_6,
          [
           array_stride_multitude_precision(input,istride,1,double),
           array_stride_multitude_precision(real_output,real_ostride,1,double),
           array_stride_multitude_precision(imag_output,imag_ostride,1,double)
          ],
          [
           input,
           real_output,
           imag_output,
           istride,
           real_ostride,
           imag_ostride
          ],
          [
           f2_const('0',double,'0.1e1',
                               '0.1e1'),
           f2_const('1',double,'0.5',
                               '0.5'),
           f2_const('2',double,'0.866025403784438646763723',
                               '0.866025403784438646763723')
          ],
          [
           i(f2v_ld1,[ae(input,0,0),ae(input,3,0)],[f2vr(25)],[]),
           i(f2v_ld1,[ae(input,2,0),ae(input,4,0)],[f2vr(15)],[]),
           i(f2v_ld1,[ae(input,1,0),ae(input,5,0)],[f2vr(12)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(1028)],[]),
           i(f2v_N(xcxnpma),[f2vr(1028),f2vr(25),f2vr(25)],[f2vr(105)],[]),
           i(f2v_N(xcxnsma),[f2vr(1028),f2vr(12),f2vr(15)],[f2vr(20)],[]),
           i(f2v_N(xcxnpma),[f2vr(1028),f2vr(12),f2vr(15)],[f2vr(1006)],[]),
           i(f2v_N(xmadd),[f2vr(1006),f2vr(1028),f2vr(20)],[f2vr(21)],[]),
           i(f2v_N(xnmsub),[f2vr(1006),f2vr(1028),f2vr(20)],[f2vr(19)],[]),
           i(f2v_ld2(p),[c('2')],[f2vr(1025)],[]),
           i(f2v_N(pmul),[f2vr(1025),f2vr(19)],[f2vr(18)],[]),
           i(f2v_st1(lo),[f2vr(18)],[ae(imag_output,2,0)],[]),
           i(f2v_st1(hi),[f2vr(18)],[ae(imag_output,1,0)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(1030)],[]),
           i(f2v_N(xnmsub),[f2vr(21),f2vr(1030),f2vr(1005)],[f2vr(1007)],[]),
           i(f2v_N(xmadd),[f2vr(1005),f2vr(1028),f2vr(21)],[f2vr(1004)],[]),
           i(f2v_st1(lo),[f2vr(1007)],[ae(real_output,1,0)],[]),
           i(f2v_st1(hi),[f2vr(1007)],[ae(real_output,2,0)],[]),
           i(f2v_st1(hi),[f2vr(1004)],[ae(real_output,3,0)],[]),
           i(f2v_st1(lo),[f2vr(1004)],[ae(real_output,0,0)],[])
          ]
).
```

Example frc_6: Medium-Level Scheduling

The medium-level scheduler tries to relax some of the instruction scheduling constraints that the high-level scheduler inserted.

```
procedure(fftw_real2hc_6,
          [
           array_stride_multitude_precision(input,istride,1,double),
           array_stride_multitude_precision(real_output,real_ostride,1,double),
           array_stride_multitude_precision(imag_output,imag_ostride,1,double)
          ],
          [
           input,
           real_output,
           imag_output,
           istride,
           real_ostride,
           imag_ostride
          ],
          [
           f2_const('0',double,'0.1e1',
                               '0.1e1'),
           f2_const('1',double,'0.5',
                               '0.5'),
           f2_const('2',double,'0.866025403784438646763723',
                               '0.866025403784438646763723')
          ],
          [
           i(f2v_ld1,[ae(input,0,0),ae(input,3,0)],[f2vr(25)],[]),
           i(f2v_ld1,[ae(input,2,0),ae(input,4,0)],[f2vr(15)],[]),
           i(f2v_ld1,[ae(input,1,0),ae(input,5,0)],[f2vr(12)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(1028)],[]),
           i(f2v_ld2(p),[c('2')],[f2vr(1025)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(1030)],[]),
           i(f2v_N(xcxnpma),[f2vr(1028),f2vr(25),f2vr(25)],[f2vr(1005)],[]),
           i(f2v_N(xcxnsma),[f2vr(1028),f2vr(12),f2vr(15)],[f2vr(20)],[]),
           i(f2v_N(xcxnpma),[f2vr(1028),f2vr(12),f2vr(15)],[f2vr(1006)],[]),
           i(f2v_N(xmadd),[f2vr(1006),f2vr(1028),f2vr(20)],[f2vr(21)],[]),
           i(f2v_N(xnmsub),[f2vr(1006),f2vr(1028),f2vr(20)],[f2vr(19)],[]),
           i(f2v_N(pmul),[f2vr(1025),f2vr(19)],[f2vr(18)],[]),
           i(f2v_N(xnmsub),[f2vr(21),f2vr(1030),f2vr(1005)],[f2vr(1007)],[]),
           i(f2v_N(xmadd),[f2vr(1005),f2vr(1028),f2vr(21)],[f2vr(1004)],[]),
           i(f2v_st1(lo),[f2vr(18)],[ae(imag_output,2,0)],[]),
           i(f2v_st1(hi),[f2vr(18)],[ae(imag_output,1,0)],[]),
           i(f2v_st1(lo),[f2vr(1007)],[ae(real_output,1,0)],[]),
           i(f2v_st1(hi),[f2vr(1007)],[ae(real_output,2,0)],[]),
           i(f2v_st1(hi),[f2vr(1004)],[ae(real_output,3,0)],[]),
           i(f2v_st1(lo),[f2vr(1004)],[ae(real_output,0,0)],[])
          ]
).
```

As a result, the medium-level scheduler tends to push loads upwards and store downwards.

Example frc_6: Floating-Point Register Allocation

The register allocator maps all temporary floating-point variables `f2vr` to logical floating-point registers `fp2r`. Also, the allocator splits composite load instructions `f2v_ld1`, that load two scalar values into one 2-way SIMD vector register, into two load instructions `f2v_ld1(lo)` and `f2v_ld1(hi)`.

```
procedure(fftw_real2hc_6,
          [
           array_stride_multitude_precision(input,istride,1,double),
           array_stride_multitude_precision(real_output,real_ostride,1,double),
           array_stride_multitude_precision(imag_output,imag_ostride,1,double)
          ],
          [
           input,
           real_output,
           imag_output,
           istride,
           real_ostride,
           imag_ostride
          ],
          [
           f2_const('0',double,'0.1e1',
                               '0.1e1'),
           f2_const('1',double,'0.5',
                               '0.5'),
           f2_const('2',double,'0.866025403784438646763723 2',
                               '0.8660254037844386467637232')
          ],
          [
           i(f2v_ld1(lo),[ae(input,0,0)],[fp2r(0)],[]),
           i(f2v_ld1(hi),[ae(input,3,0),fp2r(0)],[fp2r(0)],[]),
           i(f2v_ld1(lo),[ae(input,2,0)],[fp2r(1)],[]),
           i(f2v_ld1(hi),[ae(input,4,0),fp2r(1)],[fp2r(1)],[]),
           i(f2v_ld1(lo),[ae(input,1,0)],[fp2r(2)],[]),
           i(f2v_ld1(hi),[ae(input,5,0),fp2r(2)],[fp2r(2)],[]),
           i(f2v_ld2(p),[c('0')],[fp2r(3)],[]),
           i(f2v_ld2(p),[c('2')],[fp2r(4)],[]),
           i(f2v_ld2(p),[c('1')],[fp2r(5)],[]),
           i(f2v_N(xcxnpma),[fp2r(3),fp2r(0),fp2r(0)],[fp2r(0)],[]),
           i(f2v_N(xcxnsma),[fp2r(3),fp2r(2),fp2r(1)],[fp2r(6)],[]),
           i(f2v_N(xcxnpma),[fp2r(3),fp2r(2),fp2r(1)],[fp2r(2)],[]),
           i(f2v_N(xmadd),[fp2r(2),fp2r(3),fp2r(6)],[fp2r(7)],[]),
           i(f2v_N(xnmsub),[fp2r(2),fp2r(3),fp2r(6)],[fp2r(2)],[]),
           i(f2v_N(pmul),[fp2r(4),fp2r(2)],[fp2r(4)],[]),
           i(f2v_N(xnmsub),[fp2r(7),fp2r(5),fp2r(0)],[fp2r(5)],[]),
           i(f2v_N(xmadd),[fp2r(0),fp2r(3),fp2r(7)],[fp2r(0)],[]),
           i(f2v_st1(lo),[fp2r(4)],[ae(imag_output,2,0)],[]),
           i(f2v_st1(hi),[fp2r(4)],[ae(imag_output,1,0)],[]),
           i(f2v_st1(lo),[fp2r(5)],[ae(real_output,1,0)],[]),
           i(f2v_st1(hi),[fp2r(5)],[ae(real_output,2,0)],[]),
           i(f2v_st1(hi),[fp2r(0)],[ae(real_output,3,0)],[]),
           i(f2v_st1(lo),[fp2r(0)],[ae(real_output,0,0)],[])
          ]
).
```

Example frc_6: Effective Address Generation

Effective address generation transforms all array accesses ae(_,_,_) and all constant accesses c(_) to corresponding sequences of auxiliary integer instructions and uses a suitable addressing mode.

```
procedure(fftw_real2hc_6,
          [
           array_stride_multitude_precision(input,istride,1,double),
           array_stride_multitude_precision(real_output,real_ostride,1,double),
           array_stride_multitude_precision(imag_output,imag_ostride,1,double)
          ],
          [ input, real_output, imag_output, istride, real_ostride, imag_ostride ],
          [
           f2_const(0,double,'1.0','1.0'),
           f2_const(1,double,'0.5','0.5'),
           f2_const(2,double,'0.866025403784438646763723','0.866025403784438646763723')
          ],
          [
           i(i_shiftLeft(3),[r(6)],[r(6)],[]),
           i(i_shiftLeft(3),[r(7)],[r(7)],[]),
           i(i_shiftLeft(3),[r(8)],[r(8)],[]),
           i(i_addIS('Consts__fftw_real2hc_6@ha'),[imm(0)],[r(12)],[]),
           i(i_addI('Consts__fftw_real2hc_6@l'),[r(12)],[r(12)],[]),
           i(f2v_ld2(double,p),[r(12)],[fp2r(3)],[s(const(0))]),
           i(i_add,[r(6),r(6)],[r(2)],[]),
           i(f2v_ld1(double,lo),[r(3),r(2)],[fp2r(1)],[s(ae(input,2,0))]),
           i(i_shiftLeft(2),[r(6)],[r(9)],[]),
           i(f2v_ld1(double,hi),[fp2r(1),r(3),r(9)],[fp2r(1)],[s(ae(input,4,0))]),
           i(f2v_ld1(double,lo),[r(3),r(6)],[fp2r(2)],[s(ae(input,1,0))]),
           i(i_add,[r(3),r(9)],[r(10)],[]),
           i(f2v_ld1(double,hi),[fp2r(2),r(10),r(6)],[fp2r(2)],[s(ae(input,5,0))]),
           i(f2v_ld1(double,lo),[r(3)],[fp2r(0)],[s(ae(input,0,0))]),
           i(i_sub,[r(9),r(6)],[r(11)],[]),
           i(f2v_ld1(double,hi),[fp2r(0),r(3),r(11)],[fp2r(0)],[s(ae(input,3,0))]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(5),r(12)],[s(const(16))]),
           i(f2v_N(xcxnpma),[fp2r(3),fp2r(0),fp2r(0)],[fp2r(0)],[]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(4),r(12)],[s(const(32))]),
           i(f2v_N(xcxnsma),[fp2r(3),fp2r(2),fp2r(1)],[fp2r(6)],[]),
           i(f2v_N(xcxnpma),[fp2r(3),fp2r(2),fp2r(1)],[fp2r(2)],[]),
           i(f2v_N(xmadd),[fp2r(2),fp2r(3),fp2r(6)],[fp2r(7)],[]),
           i(f2v_N(xnmsub),[fp2r(2),fp2r(3),fp2r(6)],[fp2r(2)],[]),
           i(f2v_N(xnmsub),[fp2r(7),fp2r(5),fp2r(0)],[fp2r(5)],[]),
           i(f2v_N(xmadd),[fp2r(0),fp2r(3),fp2r(7)],[fp2r(0)],[]),
           i(f2v_N(pmul),[fp2r(4),fp2r(2)],[fp2r(4)],[]),
           i(f2v_st1(double,lo),[fp2r(5),r(4),r(7)],[],[d(ae(real_output,1,0))]),
           i(i_add,[r(7),r(7)],[r(11)],[]),
           i(f2v_st1(double,hi),[fp2r(5),r(4),r(11)],[],[d(ae(real_output,2,0))]),
           i(i_add,[r(11),r(7)],[r(9)],[]),
           i(f2v_st1(double,hi),[fp2r(0),r(4),r(9)],[],[d(ae(real_output,3,0))]),
           i(f2v_st1(double,lo),[fp2r(0),r(4)],[],[d(ae(real_output,0,0))]),
           i(i_add,[r(8),r(8)],[r(2)],[]),
           i(f2v_st1(double,lo),[fp2r(4),r(5),r(2)],[],[d(ae(imag_output,2,0))]),
           i(f2v_st1(double,hi),[fp2r(4),r(5),r(8)],[],[d(ae(imag_output,1,0))])
          ]
).
```

In this translation step, annotations are added to the respective SIMD memory transfer instruction, such that no aliasing information is lost.

Example frc_6: List-based Instruction Scheduling

```
procedure(fftw_real2hc_6,
          [
           array_stride_multitude_precision(input,istride,1,double),
           array_stride_multitude_precision(real_output,real_ostride,1,double),
           array_stride_multitude_precision(imag_output,imag_ostride,1,double)
          ],
          [
           input,
           real_output,
           imag_output,
           istride,
           real_ostride,
           imag_ostride
          ],
          [
           f2_const(0,double,'1.0',
                             '1.0'),
           f2_const(1,double,'0.5',
                             '0.5'),
           f2_const(2,double,'0.86602540378443864676372 32',
                             '0.86602540378443864676372 32')
          ],
          [
  /* 0 */  i(i_shiftLeft(3),[r(6)],[r(6)],[]),
  /* 0 */  i(i_addIS('Consts__fftw_real2hc_6@ha'),[imm(0)],[r(12)],[]),
  /* 1 */  i(i_shiftLeft(2),[r(6)],[r(9)],[]),
  /* 1 */  i(i_add,[r(6),r(6)],[r(2)],[]),
  /* 2 */  i(i_addI('Consts__fftw_real2hc_6@l'),[r(12)],[r(12)],[]),
  /* 2 */  i(i_add,[r(3),r(9)],[r(10)],[]),
  /* 3 */  i(f2v_ld1(double,lo),[r(3),r(6)],[fp2r(2)],[s(ae(input,1,0))]),
  /* 3 */  i(i_sub,[r(9),r(6)],[r(11)],[]),
  /* 4 */  i(f2v_ld1(double,lo),[r(3),r(2)],[fp2r(1)],[s(ae(input,2,0))]),
  /* 4 */  i(i_copyI(16),[],[r(0)],[]),
  /* 5 */  i(f2v_ld2(double,p),[r(12)],[fp2r(3)],[s(const(0))]),
  /* 5 */  i(i_shiftLeft(3),[r(7)],[r(7)],[]),
  /* 6 */  i(f2v_ld1(double,hi),[fp2r(2),r(10),r(6)],[fp2r(2)],[s(ae(input,5,0))]),
  /* 6 */  i(i_shiftLeft(3),[r(8)],[r(8)],[]),
  /* 7 */  i(f2v_ld1(double,hi),[fp2r(1),r(3),r(9)],[fp2r(1)],[s(ae(input,4,0))]),
  /* 7 */  i(i_add,[r(8),r(8)],[r(2)],[]),
  /* 8 */  i(f2v_ld1(double,lo),[r(3)],[fp2r(0)],[s(ae(input,0,0))]),
  /* 9 */  i(f2v_ld1(double,hi),[fp2r(0),r(3),r(11)],[fp2r(0)],[s(ae(input,3,0))]),
  /* 9 */  i(i_add,[r(7),r(7)],[r(11)],[]),
 /* 10 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(5),r(12)],[s(const(16))]),
 /* 10 */  i(i_add,[r(11),r(7)],[r(9)],[]),
 /* 11 */  i(f2v_N(xcxnsma),[fp2r(3),fp2r(2),fp2r(1)],[fp2r(6)],[]),
 /* 11 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(4)],[s(const(32))]),
 /* 12 */  i(f2v_N(xcxnpma),[fp2r(3),fp2r(2),fp2r(1)],[fp2r(2)],[]),
 /* 13 */  i(f2v_N(xcxnpma),[fp2r(3),fp2r(0),fp2r(0)],[fp2r(0)],[]),
 /* 17 */  i(f2v_N(xmadd),[fp2r(2),fp2r(3),fp2r(6)],[fp2r(7)],[]),
 /* 18 */  i(f2v_N(xnmsub),[fp2r(2),fp2r(3),fp2r(6)],[fp2r(2)],[]),
 /* 22 */  i(f2v_N(xnmsub),[fp2r(7),fp2r(5),fp2r(0)],[fp2r(5)],[]),
 /* 23 */  i(f2v_N(xmadd),[fp2r(0),fp2r(3),fp2r(7)],[fp2r(0)],[]),
 /* 24 */  i(f2v_N(pmul),[fp2r(4),fp2r(2)],[fp2r(4)],[]),
 /* 27 */  i(f2v_st1(double,lo),[fp2r(5),r(4),r(7)],[],[d(ae(real_output,1,0))]),
 /* 28 */  i(f2v_st1(double,hi),[fp2r(5),r(4),r(11)],[],[d(ae(real_output,2,0))]),
 /* 29 */  i(f2v_st1(double,hi),[fp2r(0),r(4),r(9)],[],[d(ae(real_output,3,0))]),
 /* 30 */  i(f2v_st1(double,lo),[fp2r(0),r(4)],[],[d(ae(real_output,0,0))]),
 /* 31 */  i(f2v_st1(double,lo),[fp2r(4),r(5),r(2)],[],[d(ae(imag_output,2,0))]),
 /* 32 */  i(f2v_st1(double,hi),[fp2r(4),r(5),r(8)],[],[d(ae(imag_output,1,0))]),
          ]
).
```

Example frc_6: Assembly Output

```
.data
     .align 3
Consts__fftw_real2hc_6:
     .double 1.0, 1.0, 0.5, 0.5, 0.8660254037844386467637232, 0.8660254037844386467637232
.text
     .align 3
     .globl _fftw_real2hc_6
_fftw_real2hc_6:
     slwi       r6, r6, 3
     addis      r12, 0, Consts__fftw_real2hc_6@ha
     slwi       r9, r6, 2
     add        r2, r6, r6
     addi       r12, r12, Consts__fftw_real2hc_6@l
     add        r10, r3, r9
     lfdx       f2, r3, r6
     sub        r11, r9, r6
     lfdx       f1, r3, r2
     addi       r0, 0, 16
     lfpdx      f3, 0, r12
     slwi       r7, r7, 3
     lfsdx      f2, r10, r6
     slwi       r8, r8, 3
     lfsdx      f1, r3, r9
     add        r2, r8, r8
     lfdx       f0, 0, r3
     lfsdx      f0, r3, r11
     add        r11, r7, r7
     lfpdux     f5, r12, r0
     add        r9, r11, r7
     fxcxnsma   f6, f3, f2, f1
     lfpdx      f4, r12, r0
     fxcxnpma   f2, f3, f2, f1
     fxcxnpma   f0, f3, f0, f0
     fxmadd     f7, f2, f3, f6
     fxnmsub    f2, f2, f3, f6
     fxnmsub    f5, f7, f5, f0
     fxmadd     f0, f0, f3, f7
     fpmul      f4, f4, f2
     stfdx      f5, r4, r7
     stfsdx     f5, r4, r11
     stfsdx     f0, r4, r9
     stfdx      f0, 0, r4
     stfdx      f4, r5, r2
     stfsdx     f4, r5, r8
     blr
```

7.3.3 8-point Backward Real FFT (fcr_8)

This example code is a backward 8-point real no-twiddle FFT codelet.

Example fcr_8: Scalar code produced by genfft

genfft produces the following C stub file, comprising FFTW declarations.

```
extern void fftw_hc2real_8(const fftw_real *, const fftw_real *, fftw_real *, int, int, int);

fftw_codelet_desc fftw_hc2real_8_desc = {
     "fftw_hc2real_8",
     (void (*)()) fftw_hc2real_8,
```

```
     8,
     FFTW_BACKWARD,
     FFTW_HC2REAL,
     191,
     0,
     (const int *) 0,
};
```

The following scalar code is emitted by `genfft`.

```
procedure1("fftw_hc2real_8",
           [
            arrayDecl("real_input", variable("real_istride"), 1, double),
            arrayDecl("imag_input", variable("imag_istride"), 1, double),
            arrayDecl("output", variable("ostride"), 1, double)
           ],
           ["real_input", "imag_input", "output", "real_istride", "imag_istride", "ostride"],
           [
            constDecl1(double, "+2.0"),
            constDecl1(double, "+1.414213562373095048801688724209698078569671875")
           ],
           [
            f_Load(ae("real_input",2,0), f(4)),
            f_Load(ae("imag_input",2,0), f(14)),
            f_Load(ae("real_input",0,0), f(1)),
            f_Load(ae("real_input",4,0), f(2)),
            f_BinOp(add, t(0,f(1)), t(0,f(2)), f(3)),
            f_BinOp(sub, t(0,f(1)), t(0,f(2)), f(13)),
            f_BinOp(add, t(0,f(3)), t(1,f(4)), f(6)),
            f_BinOp(sub, t(0,f(3)), t(1,f(4)), f(30)),
            f_BinOp(sub, t(0,f(13)), t(1,f(14)), f(16)),
            f_BinOp(add, t(0,f(13)), t(1,f(14)), f(25)),
            f_Load(ae("real_input",1,0), f(7)),
            f_Load(ae("real_input",3,0), f(8)),
            f_BinOp(add, t(0,f(7)), t(0,f(8)), f(9)),
            f_BinOp(sub, t(0,f(7)), t(0,f(8)), f(17)),
            f_Load(ae("imag_input",3,0), f(18)),
            f_Load(ae("imag_input",1,0), f(19)),
            f_BinOp(add, t(0,f(18)), t(0,f(19)), f(20)),
            f_BinOp(sub, t(0,f(19)), t(0,f(18)), f(31)),
            f_BinOp(sub, t(0,f(17)), t(0,f(20)), f(21)),
            f_BinOp(add, t(0,f(17)), t(0,f(20)), f(26)),
            f_BinOp(sub, t(0,f(6)), t(1,f(9)), f(11)),
            f_Store(t(0,f(11)), ae("output",4,0)),
            f_BinOp(add, t(0,f(6)), t(1,f(9)), f(12)),
            f_Store(t(0,f(12)), ae("output",0,0)),
            f_BinOp(sub, t(0,f(16)), t(2,f(21)), f(23)),
            f_BinOp(add, t(0,f(16)), t(2,f(21)), f(24)),
            f_Store(t(0,f(23)), ae("output",5,0)),
            f_Store(t(0,f(24)), ae("output",1,0)),
            f_BinOp(add, t(0,f(30)), t(1,f(31)), f(34)),
            f_Store(t(0,f(34)), ae("output",6,0)),
            f_BinOp(sub, t(0,f(30)), t(1,f(31)), f(33)),
            f_Store(t(0,f(33)), ae("output",2,0)),
            f_BinOp(sub, t(0,f(25)), t(2,f(26)), f(28)),
            f_BinOp(add, t(0,f(25)), t(2,f(26)), f(29)),
            f_Store(t(0,f(28)), ae("output",3,0)),
            f_Store(t(0,f(29)), ae("output",7,0))
           ]
).
```

Example fcr_8: Vectorization

```
procedure(fftw_hc2real_8,
          [
           array_stride_multitude_precision(real_input,real_istride,1,double),
           array_stride_multitude_precision(imag_input,imag_istride,1,double),
           array_stride_multitude_precision(output,ostride,1,double)
          ],
          [ real_input, imag_input, output, real_istride, imag_istride, ostride ],
          [
           f2_const('0',double,'+1.4142135623730950488016887242096980785696718753',
                               '+1.4142135623730950488016887242096980785696718753'),
           f2_const('1',double,'+2.0','+1.0'),
           f2_const('2',double,'+1.0','+2.0'),
           f2_const('3',double,'+2.0','+2.0')
          ],
          [
           i(f2v_ld1,[ae(real_input,0,0),ae(real_input,4,0)],[f2vr(26)],[]),
           i(f2v_ld1,[ae(real_input,1,0),ae(imag_input,3,0)],[f2vr(35)],[]),
           i(f2v_N(swap),[f2vr(35)],[f2vr(32)],[]),
           i(f2v_N(paddsub(sub,add)),[f2vr(34),f2vr(32)],[f2vr(20)],[]),
           i(f2v_N(swap),[f2vr(31)],[f2vr(34)],[]),
           i(f2v_N(paddsub(add,sub)),[f2vr(32),f2vr(33)],[f2vr(15)],[]),
           i(f2v_N(swap),[f2vr(31)],[f2vr(33)],[]),
           i(f2v_ld1,[ae(real_input,3,0),ae(imag_input,1,0)],[f2vr(31)],[]),
           i(f2v_N(paddsub(sub,add)),[f2vr(30),f2vr(28)],[f2vr(19)],[]),
           i(f2v_N(pmul),[f2vr(13),f2vr(29)],[f2vr(30)],[]),
           i(f2v_N(swap),[f2vr(27)],[f2vr(29)],[]),
           i(f2v_N(pmul),[f2vr(10),f2vr(27)],[f2vr(28)],[]),
           i(f2v_ld1acc(add),[ae(real_input,2,0),f2vr(26)],[f2vr(27)],[]),
           i(f2v_ld1acc(sub),[ae(imag_input,2,0),f2vr(26)],[f2vr(9)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(19),f2vr(24)],[f2vr(25)],[]),
           i(f2v_N(swap),[f2vr(25)],[f2vr(3)],[]),
           i(f2v_N(pmul),[f2vr(21),f2vr(20)],[f2vr(24)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(19),f2vr(22)],[f2vr(23)],[]),
           i(f2v_N(swap),[f2vr(23)],[f2vr(2)],[]),
           i(f2v_N(pmul),[f2vr(21),f2vr(20)],[f2vr(22)],[]),
           i(f2v_ld2(p),[c('3')],[f2vr(21)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(4),f2vr(17)],[f2vr(18)],[]),
           i(f2v_N(swap),[f2vr(18)],[f2vr(1)],[]),
           i(f2v_N(pmul),[f2vr(6),f2vr(5)],[f2vr(17)],[]),
           i(f2v_N(paddsub(add,sub)),[f2vr(15),f2vr(16)],[f2vr(5)],[]),
           i(f2v_N(swap),[f2vr(15)],[f2vr(16)],[]),
           i(f2v_N(paddsub(add,sub)),[f2vr(11),f2vr(14)],[f2vr(4)],[]),
           i(f2v_N(pmul),[f2vr(13),f2vr(12)],[f2vr(14)],[]),
           i(f2v_ld2(p),[c('2')],[f2vr(13)],[]),
           i(f2v_N(swap),[f2vr(9)],[f2vr(12)],[]),
           i(f2v_N(pmul),[f2vr(10),f2vr(9)],[f2vr(11)],[]),
           i(f2v_ld2(p),[c('1')],[f2vr(10)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(4),f2vr(7)],[f2vr(8)],[]),
           i(f2v_N(swap),[f2vr(8)],[f2vr(0)],[]),
           i(f2v_N(pmul),[f2vr(6),f2vr(5)],[f2vr(7)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(6)],[]),
           i(f2v_st1(hi),[f2vr(3)],[ae(output,2,0)],[]),
           i(f2v_st1(lo),[f2vr(3)],[ae(output,4,0)],[]),
           i(f2v_st1(hi),[f2vr(2)],[ae(output,6,0)],[]),
           i(f2v_st1(lo),[f2vr(2)],[ae(output,0,0)],[]),
           i(f2v_st1(hi),[f2vr(1)],[ae(output,3,0)],[]),
           i(f2v_st1(lo),[f2vr(1)],[ae(output,5,0)],[]),
           i(f2v_st1(hi),[f2vr(0)],[ae(output,7,0)],[]),
           i(f2v_st1(lo),[f2vr(0)],[ae(output,1,0)],[])
          ]
).
```

Example fcr_8: Peephole Optimization

```
procedure(fftw_hc2real_8,
          [
           array_stride_multitude_precision(real_input,real_istride,1,double),
           array_stride_multitude_precision(imag_input,imag_istride,1,double),
           array_stride_multitude_precision(output,ostride,1,double)
          ],
          [ real_input, imag_input, output, real_istride, imag_istride, ostride ],
          [
           f2_const('(0.5,-0.2e1)',double,'0.5','-0.2e1'),
           f2_const('2',double,'0.7071067811865475244008444','-0.1414213562373095048801689e1'),
           f2_const('(0.1e1,0.1e1)',double,'0.1e1','0.1e1'),
           f2_const('(0.1e1,-0.5)',double,'0.1e1','-0.5'),
           f2_const('(0.1e1,-0.1e1)',double,'0.1e1','-0.1e1'),
           f2_const('(0.1e1,-0.2e1)',double,'0.1e1','-0.2e1'),
           f2_const('0',double,'0.1414213562373095048801689e1','0.1414213562373095048801689e1'),
           f2_const('3',double,'0.2e1','0.2e1')
          ],
          [
           i(f2v_st1(hi),[f2vr(1008)],[ae(output,7,0)],[]),
           i(f2v_st1(lo),[f2vr(1008)],[ae(output,1,0)],[]),
           i(f2v_ld1,[ae(real_input,0,0),ae(real_input,4,0)],[f2vr(26)],[]),
           i(f2v_ld1(lo),[f2vr(1030),ae(real_input,2,0)],[f2vr(27)],[]),
           i(f2v_ld1(lo),[f2vr(1029),ae(imag_input,2,0)],[f2vr(9)],[]),
           i(f2v_st1(lo),[f2vr(25)],[ae(output,2,0)],[]),
           i(f2v_st1(hi),[f2vr(25)],[ae(output,4,0)],[]),
           i(f2v_st1(lo),[f2vr(23)],[ae(output,6,0)],[]),
           i(f2v_st1(hi),[f2vr(23)],[ae(output,0,0)],[]),
           i(f2v_st1(lo),[f2vr(18)],[ae(output,3,0)],[]),
           i(f2v_st1(hi),[f2vr(18)],[ae(output,5,0)],[]),
           i(f2v_ld1,[ae(imag_input,3,0),ae(real_input,1,0)],[f2vr(32)],[]),
           i(f2v_ld1,[ae(imag_input,1,0),ae(real_input,3,0)],[f2vr(34)],[]),
           i(f2v_N(xmul),[f2vr(1040),f2vr(1013)],[f2vr(1008)],[]),
           i(f2v_N(xmul),[f2vr(1040),f2vr(1009)],[f2vr(23)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(1042)],[]),
           i(f2v_N(pmul),[f2vr(1042),f2vr(1010)],[f2vr(18)],[]),
           i(f2v_ld2(p),[c('3')],[f2vr(1043)],[]),
           i(f2v_N(pmul),[f2vr(1043),f2vr(1011)],[f2vr(25)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.1e1)')],[f2vr(1044)],[]),
           i(f2v_N(pmadd),[f2vr(34),f2vr(1044),f2vr(32)],[f2vr(15)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.5)')],[f2vr(1045)],[]),
           i(f2v_N(pnmadd),[f2vr(1007),f2vr(1045),f2vr(20)],[f2vr(1011)],[]),
           i(f2v_N(pnmsub),[f2vr(32),f2vr(1044),f2vr(34)],[f2vr(20)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.2e1)')],[f2vr(1040)],[]),
           i(f2v_N(pnmsub),[f2vr(20),f2vr(1040),f2vr(1007)],[f2vr(1009)],[]),
           i(f2v_N(xcxnsma),[f2vr(1048),f2vr(15),f2vr(15)],[f2vr(5)],[]),
           i(f2v_ld2(p),[c('(0.1e1,0.1e1)')],[f2vr(1048)],[]),
           i(f2v_N(xmadd),[f2vr(26),f2vr(1048),f2vr(26)],[f2vr(1030)],[]),
           i(f2v_ld2(p),[c('2')],[f2vr(1053)],[]),
           i(f2v_N(xmsub),[f2vr(1012),f2vr(1053),f2vr(5)],[f2vr(1010)],[]),
           i(f2v_N(xmsub),[f2vr(26),f2vr(1048),f2vr(26)],[f2vr(1029)],[]),
           i(f2v_ld2(p),[c('(0.5,-0.2e1)')],[f2vr(1052)],[]),
           i(f2v_N(xnmsub),[f2vr(27),f2vr(1052),f2vr(27)],[f2vr(1007)],[]),
           i(f2v_N(xnmsub),[f2vr(5),f2vr(1053),f2vr(1012)],[f2vr(1013)],[]),
           i(f2v_N(xnmsub),[f2vr(9),f2vr(1052),f2vr(9)],[f2vr(1012)],[])
          ]
).
```

Example fcr_8: High-Level Scheduling

```
procedure(fftw_hc2real_8,
          [
           array_stride_multitude_precision(real_input,real_istride,1,double),
           array_stride_multitude_precision(imag_input,imag_istride,1,double),
           array_stride_multitude_precision(output,ostride,1,double)
          ],
          [ real_input, imag_input, output, real_istride, imag_istride, ostride ],
          [
           f2_const('(0.5,-0.2e1)',double,'0.5','-0.2e1'),
           f2_const('2',double,'0.70710678118654752440084444','-0.14142135623730950488016889e1'),
           f2_const('(0.1e1,0.1e1)',double,'0.1e1','0.1e1'),
           f2_const('(0.1e1,-0.5)',double,'0.1e1','-0.5'),
           f2_const('(0.1e1,-0.1e1)',double,'0.1e1','-0.1e1'),
           f2_const('(0.1e1,-0.2e1)',double,'0.1e1','-0.2e1'),
           f2_const('0',double,'0.14142135623730950488016889e1','0.14142135623730950488016889e1'),
           f2_const('3',double,'0.2e1','0.2e1')
          ],
          [
           i(f2v_ld1,[ae(imag_input,1,0),ae(real_input,3,0)],[f2vr(34)],[]),
           i(f2v_ld1,[ae(imag_input,3,0),ae(real_input,1,0)],[f2vr(32)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.1e1)')],[f2vr(1044)],[]),
           i(f2v_N(pmadd),[f2vr(34),f2vr(1044),f2vr(32)],[f2vr(15)],[]),
           i(f2v_N(pnmsub),[f2vr(32),f2vr(1044),f2vr(34)],[f2vr(20)],[]),
           i(f2v_ld1,[ae(real_input,0,0),ae(real_input,4,0)],[f2vr(26)],[]),
           i(f2v_ld2(p),[c('(0.1e1,0.1e1)')],[f2vr(1048)],[]),
           i(f2v_N(xcxnsma),[f2vr(1048),f2vr(15),f2vr(15)],[f2vr(5)],[]),
           i(f2v_N(xmadd),[f2vr(26),f2vr(1048),f2vr(26)],[f2vr(1030)],[]),
           i(f2v_N(xmsub),[f2vr(26),f2vr(1048),f2vr(26)],[f2vr(1029)],[]),
           i(f2v_ld1(lo),[f2vr(1030),ae(real_input,2,0)],[f2vr(27)],[]),
           i(f2v_ld2(p),[c('(0.5,-0.2e1)')],[f2vr(1052)],[]),
           i(f2v_N(xnmsub),[f2vr(27),f2vr(1052),f2vr(27)],[f2vr(1007)],[]),
           i(f2v_ld1(lo),[f2vr(1029),ae(imag_input,2,0)],[f2vr(9)],[]),
           i(f2v_N(xnmsub),[f2vr(9),f2vr(1052),f2vr(9)],[f2vr(1012)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.5)')],[f2vr(1045)],[]),
           i(f2v_N(pnmadd),[f2vr(1007),f2vr(1045),f2vr(20)],[f2vr(1011)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.2e1)')],[f2vr(1040)],[]),
           i(f2v_N(pnmsub),[f2vr(20),f2vr(1040),f2vr(1007)],[f2vr(1009)],[]),
           i(f2v_ld2(p),[c('3')],[f2vr(1043)],[]),
           i(f2v_N(pmul),[f2vr(1043),f2vr(1011)],[f2vr(25)],[]),
           i(f2v_st1(lo),[f2vr(25)],[ae(output,2,0)],[]),
           i(f2v_st1(hi),[f2vr(25)],[ae(output,4,0)],[]),
           i(f2v_N(xmul),[f2vr(1040),f2vr(1009)],[f2vr(23)],[]),
           i(f2v_st1(lo),[f2vr(23)],[ae(output,6,0)],[]),
           i(f2v_st1(hi),[f2vr(23)],[ae(output,0,0)],[]),
           i(f2v_ld2(p),[c('2')],[f2vr(1053)],[]),
           i(f2v_N(xmsub),[f2vr(1012),f2vr(1053),f2vr(5)],[f2vr(1010)],[]),
           i(f2v_N(xnmsub),[f2vr(5),f2vr(1053),f2vr(1012)],[f2vr(1013)],[]),
           i(f2v_N(xmul),[f2vr(1040),f2vr(1013)],[f2vr(1008)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(1042)],[]),
           i(f2v_N(pmul),[f2vr(1042),f2vr(1010)],[f2vr(18)],[]),
           i(f2v_st1(lo),[f2vr(18)],[ae(output,3,0)],[]),
           i(f2v_st1(hi),[f2vr(18)],[ae(output,5,0)],[]),
           i(f2v_st1(hi),[f2vr(1008)],[ae(output,7,0)],[]),
           i(f2v_st1(lo),[f2vr(1008)],[ae(output,1,0)],[])
          ]
).
```

Example **fcr_8**: Medium-Level Scheduling

```
procedure(fftw_hc2real_8,
          [
           array_stride_multitude_precision(real_input,real_istride,1,double),
           array_stride_multitude_precision(imag_input,imag_istride,1,double),
           array_stride_multitude_precision(output,ostride,1,double)
          ],
          [ real_input, imag_input, output, real_istride, imag_istride, ostride ],
          [
           f2_const('(0.5,-0.2e1)',double,'0.5','-0.2e1'),
           f2_const('2',double,'0.70710678118654752440084436','-0.14142135623730950488016887e1'),
           f2_const('(0.1e1,0.1e1)',double,'0.1e1','0.1e1'),
           f2_const('(0.1e1,-0.5)',double,'0.1e1','-0.5'),
           f2_const('(0.1e1,-0.1e1)',double,'0.1e1','-0.1e1'),
           f2_const('(0.1e1,-0.2e1)',double,'0.1e1','-0.2e1'),
           f2_const('0',double,'0.14142135623730950488016887e1','0.14142135623730950488016887e1'),
           f2_const('3',double,'0.2e1','0.2e1')
          ],
          [
           i(f2v_ld1,[ae(imag_input,1,0),ae(real_input,3,0)],[f2vr(34)],[]),
           i(f2v_ld1,[ae(imag_input,3,0),ae(real_input,1,0)],[f2vr(32)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.1e1)')],[f2vr(1044)],[]),
           i(f2v_ld1,[ae(real_input,0,0),ae(real_input,4,0)],[f2vr(26)],[]),
           i(f2v_ld2(p),[c('(0.1e1,0.1e1)')],[f2vr(1048)],[]),
           i(f2v_ld2(p),[c('(0.5,-0.2e1)')],[f2vr(1052)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.5)')],[f2vr(1045)],[]),
           i(f2v_N(pmadd),[f2vr(34),f2vr(1044),f2vr(32)],[f2vr(15)],[]),
           i(f2v_N(pnmsub),[f2vr(32),f2vr(1044),f2vr(34)],[f2vr(20)],[]),
           i(f2v_N(xmadd),[f2vr(26),f2vr(1048),f2vr(26)],[f2vr(1030)],[]),
           i(f2v_N(xmsub),[f2vr(26),f2vr(1048),f2vr(26)],[f2vr(1029)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.2e1)')],[f2vr(1040)],[]),
           i(f2v_ld2(p),[c('3')],[f2vr(1043)],[]),
           i(f2v_ld2(p),[c('2')],[f2vr(1053)],[]),
           i(f2v_N(xcxnsma),[f2vr(1048),f2vr(15),f2vr(15)],[f2vr(5)],[]),
           i(f2v_ld2(p),[c('0')],[f2vr(1042)],[]),
           i(f2v_ld1(lo),[f2vr(1030),ae(real_input,2,0)],[f2vr(27)],[]),
           i(f2v_N(xnmsub),[f2vr(27),f2vr(1052),f2vr(27)],[f2vr(1007)],[]),
           i(f2v_ld1(lo),[f2vr(1029),ae(imag_input,2,0)],[f2vr(9)],[]),
           i(f2v_N(xnmsub),[f2vr(9),f2vr(1052),f2vr(9)],[f2vr(1012)],[]),
           i(f2v_N(pnmadd),[f2vr(1007),f2vr(1045),f2vr(20)],[f2vr(1011)],[]),
           i(f2v_N(pnmsub),[f2vr(20),f2vr(1040),f2vr(1007)],[f2vr(1009)],[]),
           i(f2v_N(xmsub),[f2vr(1012),f2vr(1053),f2vr(5)],[f2vr(1010)],[]),
           i(f2v_N(xnmsub),[f2vr(5),f2vr(1053),f2vr(1012)],[f2vr(1013)],[]),
           i(f2v_N(pmul),[f2vr(1043),f2vr(1011)],[f2vr(25)],[]),
           i(f2v_N(xmul),[f2vr(1040),f2vr(1009)],[f2vr(23)],[]),
           i(f2v_N(xmul),[f2vr(1040),f2vr(1013)],[f2vr(1008)],[]),
           i(f2v_N(pmul),[f2vr(1042),f2vr(1010)],[f2vr(18)],[]),
           i(f2v_st1(lo),[f2vr(25)],[ae(output,2,0)],[]),
           i(f2v_st1(hi),[f2vr(25)],[ae(output,4,0)],[]),
           i(f2v_st1(lo),[f2vr(23)],[ae(output,6,0)],[]),
           i(f2v_st1(hi),[f2vr(23)],[ae(output,0,0)],[]),
           i(f2v_st1(lo),[f2vr(18)],[ae(output,3,0)],[]),
           i(f2v_st1(hi),[f2vr(18)],[ae(output,5,0)],[]),
           i(f2v_st1(hi),[f2vr(1008)],[ae(output,7,0)],[]),
           i(f2v_st1(lo),[f2vr(1008)],[ae(output,1,0)],[])
          ]
).
```

Example fcr_8: Floating-Point Register Allocation

```
procedure(fftw_hc2real_8,
          [
           array_stride_multitude_precision(real_input,real_istride,1,double),
           array_stride_multitude_precision(imag_input,imag_istride,1,double),
           array_stride_multitude_precision(output,ostride,1,double)
          ],
          [ real_input, imag_input, output, real_istride, imag_istride, ostride ],
          [
           f2_const('(0.5,-0.2e1)',double,'0.5','-0.2e1'),
           f2_const('2',double,'0.70710678118654752440084436','-0.14142135623730950488016887e1'),
           f2_const('(0.1e1,0.1e1)',double,'0.1e1','0.1e1'),
           f2_const('(0.1e1,-0.5)',double,'0.1e1','-0.5'),
           f2_const('(0.1e1,-0.1e1)',double,'0.1e1','-0.1e1'),
           f2_const('(0.1e1,-0.2e1)',double,'0.1e1','-0.2e1'),
           f2_const('0',double,'0.14142135623730950488016887e1','0.14142135623730950488016887e1'),
           f2_const('3',double,'0.2e1','0.2e1')
          ],
          [
           i(f2v_ld1(lo),[ae(imag_input,1,0)],[fp2r(0)],[]),
           i(f2v_ld1(hi),[ae(real_input,3,0),fp2r(0)],[fp2r(0)],[]),
           i(f2v_ld1(lo),[ae(imag_input,3,0)],[fp2r(1)],[]),
           i(f2v_ld1(hi),[ae(real_input,1,0),fp2r(1)],[fp2r(1)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.1e1)')],[fp2r(2)],[]),
           i(f2v_ld1(lo),[ae(real_input,0,0)],[fp2r(3)],[]),
           i(f2v_ld1(hi),[ae(real_input,4,0),fp2r(3)],[fp2r(3)],[]),
           i(f2v_ld2(p),[c('(0.1e1,0.1e1)')],[fp2r(4)],[]),
           i(f2v_ld2(p),[c('(0.5,-0.2e1)')],[fp2r(5)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.5)')],[fp2r(6)],[]),
           i(f2v_N(pmadd),[fp2r(0),fp2r(2),fp2r(1)],[fp2r(7)],[]),
           i(f2v_N(pnmsub),[fp2r(1),fp2r(2),fp2r(0)],[fp2r(1)],[]),
           i(f2v_N(xmadd),[fp2r(3),fp2r(4),fp2r(3)],[fp2r(8)],[]),
           i(f2v_N(xmsub),[fp2r(3),fp2r(4),fp2r(3)],[fp2r(3)],[]),
           i(f2v_ld2(p),[c('(0.1e1,-0.2e1)')],[fp2r(9)],[]),
           i(f2v_ld2(p),[c('3')],[fp2r(10)],[]),
           i(f2v_ld2(p),[c('2')],[fp2r(11)],[]),
           i(f2v_N(xcxnsma),[fp2r(4),fp2r(7),fp2r(7)],[fp2r(4)],[]),
           i(f2v_ld2(p),[c('0')],[fp2r(12)],[]),
           i(f2v_ld1(lo),[fp2r(8),ae(real_input,2,0)],[fp2r(8)],[]),
           i(f2v_N(xnmsub),[fp2r(8),fp2r(5),fp2r(8)],[fp2r(8)],[]),
           i(f2v_ld1(lo),[fp2r(3),ae(imag_input,2,0)],[fp2r(3)],[]),
           i(f2v_N(xnmsub),[fp2r(3),fp2r(5),fp2r(3)],[fp2r(5)],[]),
           i(f2v_N(pnmadd),[fp2r(8),fp2r(6),fp2r(1)],[fp2r(6)],[]),
           i(f2v_N(pnmsub),[fp2r(1),fp2r(9),fp2r(8)],[fp2r(1)],[]),
           i(f2v_N(xmsub),[fp2r(5),fp2r(11),fp2r(4)],[fp2r(13)],[]),
           i(f2v_N(xnmsub),[fp2r(4),fp2r(11),fp2r(5)],[fp2r(4)],[]),
           i(f2v_N(pmul),[fp2r(10),fp2r(6)],[fp2r(10)],[]),
           i(f2v_N(xmul),[fp2r(9),fp2r(1)],[fp2r(1)],[]),
           i(f2v_N(xmul),[fp2r(9),fp2r(4)],[fp2r(9)],[]),
           i(f2v_N(pmul),[fp2r(12),fp2r(13)],[fp2r(12)],[]),
           i(f2v_st1(lo),[fp2r(10)],[ae(output,2,0)],[]),
           i(f2v_st1(hi),[fp2r(10)],[ae(output,4,0)],[]),
           i(f2v_st1(lo),[fp2r(1)],[ae(output,6,0)],[]),
           i(f2v_st1(hi),[fp2r(1)],[ae(output,0,0)],[]),
           i(f2v_st1(lo),[fp2r(12)],[ae(output,3,0)],[]),
           i(f2v_st1(hi),[fp2r(12)],[ae(output,5,0)],[]),
           i(f2v_st1(hi),[fp2r(9)],[ae(output,7,0)],[]),
           i(f2v_st1(lo),[fp2r(9)],[ae(output,1,0)],[])
          ]
).
```

Example fcr_8: Effective Address Generation

```
procedure(fftw_hc2real_8,
          [
           array_stride_multitude_precision(real_input,real_istride,1,double),
           array_stride_multitude_precision(imag_input,imag_istride,1,double),
           array_stride_multitude_precision(output,ostride,1,double)
          ],
          [ real_input, imag_input, output, real_istride, imag_istride, ostride ],
          [
           f2_const(0,double,'0.1e1','0.1e1'),
           f2_const(1,double,'0.1e1','-0.1e1'),
           f2_const(2,double,'0.5','-0.2e1'),
           f2_const(3,double,'0.1e1','-0.5'),
           f2_const(4,double,'0.1e1','-0.2e1'),
           f2_const(5,double,'0.7071067811865475244008444','-0.1414213562373095048801689e1'),
           f2_const(6,double,'0.2e1','0.2e1'),
           f2_const(7,double,'0.1414213562373095048801689e1','0.1414213562373095048801689e1')
          ],
          [
           i(i_shiftLeft(3),[r(6)],[r(6)],[]),
           i(i_shiftLeft(3),[r(7)],[r(7)],[]),
           i(i_shiftLeft(3),[r(8)],[r(8)],[]),
           i(i_addIS('Consts__fftw_hc2real_8@ha'),[imm(0)],[r(12)],[]),
           i(i_addI('Consts__fftw_hc2real_8@l'),[r(12)],[r(12)],[]),
           i(f2v_ld2(double,p),[r(12)],[fp2r(4)],[s(const(0))]),
           i(f2v_ld1(double,lo),[r(3)],[fp2r(3)],[s(ae(real_input,0,0))]),
           i(i_shiftLeft(2),[r(6)],[r(2)],[]),
           i(f2v_ld1(double,hi),[fp2r(3),r(3),r(2)],[fp2r(3)],[s(ae(real_input,4,0))]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(2),r(12)],[s(const(16))]),
           i(f2v_ld1(double,lo),[r(4),r(7)],[fp2r(0)],[s(ae(imag_input,1,0))]),
           i(i_sub,[r(2),r(6)],[r(9)],[]),
           i(f2v_ld1(double,hi),[fp2r(0),r(3),r(9)],[fp2r(0)],[s(ae(real_input,3,0))]),
           i(i_add,[r(7),r(7)],[r(10)],[]),
           i(i_add,[r(10),r(7)],[r(11)],[]),
           i(f2v_ld1(double,lo),[r(4),r(11)],[fp2r(1)],[s(ae(imag_input,3,0))]),
           i(f2v_ld1(double,hi),[fp2r(1),r(3),r(6)],[fp2r(1)],[s(ae(real_input,1,0))]),
           i(f2v_N(xmadd),[fp2r(3),fp2r(4),fp2r(3)],[fp2r(8)],[]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(5),r(12)],[s(const(32))]),
           i(f2v_N(xmsub),[fp2r(3),fp2r(4),fp2r(3)],[fp2r(3)],[]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(6),r(12)],[s(const(48))]),
           i(f2v_N(pmadd),[fp2r(0),fp2r(2),fp2r(1)],[fp2r(7)],[]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(9),r(12)],[s(const(64))]),
           i(f2v_N(pnmsub),[fp2r(1),fp2r(2),fp2r(0)],[fp2r(1)],[]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(11),r(12)],[s(const(80))]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(10),r(12)],[s(const(96))]),
           i(i_add,[r(6),r(6)],[r(2)],[]),
           i(f2v_ld1(double,lo),[fp2r(8),r(3),r(2)],[fp2r(8)],[s(ae(real_input,2,0))]),
           i(f2v_N(xnmsub),[fp2r(8),fp2r(5),fp2r(8)],[fp2r(8)],[]),
           i(f2v_ld1(double,lo),[fp2r(3),r(4),r(10)],[fp2r(3)],[s(ae(imag_input,2,0))]),
           i(f2v_N(xnmsub),[fp2r(3),fp2r(5),fp2r(3)],[fp2r(5)],[]),
           i(f2v_N(xcxnsma),[fp2r(4),fp2r(7),fp2r(7)],[fp2r(4)],[]),
           i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(12),r(12)],[s(const(112))]),
           i(f2v_N(pnmadd),[fp2r(8),fp2r(6),fp2r(1)],[fp2r(6)],[]),
           i(f2v_N(pnmsub),[fp2r(1),fp2r(9),fp2r(8)],[fp2r(1)],[]),
           i(f2v_N(xmsub),[fp2r(5),fp2r(11),fp2r(4)],[fp2r(13)],[]),
           i(f2v_N(xnmsub),[fp2r(4),fp2r(11),fp2r(5)],[fp2r(4)],[]),
           i(f2v_N(pmul),[fp2r(10),fp2r(6)],[fp2r(10)],[]),
           i(f2v_N(xmul),[fp2r(9),fp2r(1)],[fp2r(1)],[]),
           i(f2v_N(pmul),[fp2r(12),fp2r(13)],[fp2r(12)],[]),
           i(f2v_N(xmul),[fp2r(9),fp2r(4)],[fp2r(9)],[]),
           i(i_add,[r(8),r(8)],[r(2)],[]),
           i(f2v_st1(double,lo),[fp2r(10),r(5),r(2)],[],[d(ae(output,2,0))]),
           i(i_shiftLeft(2),[r(8)],[r(9)],[]),
           i(f2v_st1(double,hi),[fp2r(10),r(5),r(9)],[],[d(ae(output,4,0))]),
```

```
             i(i_add,[r(5),r(9)],[r(6)],[]),
             i(f2v_st1(double,lo),[fp2r(1),r(6),r(2)],[],[d(ae(output,6,0))]),
             i(f2v_st1(double,hi),[fp2r(1),r(5)],[],[d(ae(output,0,0))]),
             i(i_sub,[r(9),r(8)],[r(3)],[]),
             i(f2v_st1(double,lo),[fp2r(12),r(5),r(3)],[],[d(ae(output,3,0))]),
             i(f2v_st1(double,hi),[fp2r(12),r(6),r(8)],[],[d(ae(output,5,0))]),
             i(f2v_st1(double,hi),[fp2r(9),r(6),r(3)],[],[d(ae(output,7,0))]),
             i(f2v_st1(double,lo),[fp2r(9),r(5),r(8)],[],[d(ae(output,1,0))])
            ]
).
```

Example `fcr_8`: List-based Instruction Scheduling

```
procedure(fftw_hc2real_8,
          [
           array_stride_multitude_precision(real_input,real_istride,1,double),
           array_stride_multitude_precision(imag_input,imag_istride,1,double),
           array_stride_multitude_precision(output,ostride,1,double)
          ],
          [ real_input, imag_input, output, real_istride, imag_istride, ostride ],
          [
           f2_const(0,double,'0.1e1','0.1e1'),
           f2_const(1,double,'0.1e1','-0.1e1'),
           f2_const(2,double,'0.5','-0.2e1'),
           f2_const(3,double,'0.1e1','-0.5'),
           f2_const(4,double,'0.1e1','-0.2e1'),
           f2_const(5,double,'0.707106781186547524400844','-0.1414213562373095048801689e1'),
           f2_const(6,double,'0.2e1','0.2e1'),
           f2_const(7,double,'0.1414213562373095048801689e1','0.1414213562373095048801689e1')
          ],
          [
  /* 0 */  i(i_shiftLeft(3),[r(6)],[r(6)],[]),
  /* 0 */  i(i_addIS('Consts__fftw_hc2real_8@ha'),[imm(0)],[r(12)],[]),
  /* 1 */  i(i_shiftLeft(3),[r(7)],[r(7)],[]),
  /* 1 */  i(i_addI('Consts__fftw_hc2real_8@l'),[r(12)],[r(12)],[]),
  /* 2 */  i(i_shiftLeft(2),[r(6)],[r(2)],[]),
  /* 2 */  i(f2v_ld1(double,lo),[r(3)],[fp2r(3)],[s(ae(real_input,0,0))]),
  /* 3 */  i(f2v_ld2(double,p),[r(12)],[fp2r(4)],[s(const(0))]),
  /* 3 */  i(i_add,[r(7),r(7)],[r(10)],[]),
  /* 4 */  i(f2v_ld1(double,hi),[fp2r(3),r(3),r(2)],[fp2r(3)],[s(ae(real_input,4,0))]),
  /* 4 */  i(i_add,[r(10),r(7)],[r(11)],[]),
  /* 5 */  i(i_copyI(16),[],[r(0)],[]),
  /* 5 */  i(i_sub,[r(2),r(6)],[r(9)],[]),
  /* 6 */  i(f2v_ld1(double,lo),[r(4),r(7)],[fp2r(0)],[s(ae(imag_input,1,0))]),
  /* 6 */  i(i_add,[r(6),r(6)],[r(2)],[]),
  /* 7 */  i(f2v_ld1(double,lo),[r(4),r(11)],[fp2r(1)],[s(ae(imag_input,3,0))]),
  /* 7 */  i(i_shiftLeft(3),[r(8)],[r(8)],[]),
  /* 8 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(2),r(12)],[s(const(16))]),
  /* 8 */  i(f2v_N(xmadd),[fp2r(3),fp2r(4),fp2r(3)],[fp2r(8)],[]),
  /* 9 */  i(f2v_ld1(double,hi),[fp2r(0),r(3),r(9)],[fp2r(0)],[s(ae(real_input,3,0))]),
  /* 9 */  i(f2v_N(xmsub),[fp2r(3),fp2r(4),fp2r(3)],[fp2r(3)],[]),
 /* 10 */  i(f2v_ld1(double,hi),[fp2r(1),r(3),r(6)],[fp2r(1)],[s(ae(real_input,1,0))]),
 /* 10 */  i(i_shiftLeft(2),[r(8)],[r(9)],[]),
 /* 11 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(5),r(12)],[s(const(32))]),
 /* 11 */  i(i_add,[r(5),r(9)],[r(6)],[]),
 /* 12 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(6),r(12)],[s(const(48))]),
 /* 13 */  i(f2v_ld1(double,lo),[fp2r(8),r(3),r(2)],[fp2r(8)],[s(ae(real_input,2,0))]),
 /* 13 */  i(i_add,[r(8),r(8)],[r(2)],[]),
 /* 14 */  i(f2v_N(pmadd),[fp2r(0),fp2r(2),fp2r(1)],[fp2r(7)],[]),
 /* 14 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(9),r(12)],[s(const(64))]),
 /* 15 */  i(f2v_N(pnmsub),[fp2r(1),fp2r(2),fp2r(0)],[fp2r(1)],[]),
 /* 15 */  i(f2v_ld1(double,lo),[fp2r(3),r(4),r(10)],[fp2r(3)],[s(ae(imag_input,2,0))]),
 /* 16 */  i(f2v_N(xnmsub),[fp2r(8),fp2r(5),fp2r(8)],[fp2r(8)],[]),
```

```
 /* 16 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(11),r(12)],[s(const(80))]),
 /* 17 */  i(f2v_N(xnmsub),[fp2r(3),fp2r(5),fp2r(3)],[fp2r(5)],[]),
 /* 17 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(10),r(12)],[s(const(96))]),
 /* 18 */  i(f2v_ld2(double,p),[r(12),r(0)],[fp2r(12)],[s(const(112))]),
 /* 18 */  i(i_sub,[r(9),r(8)],[r(3)],[]),
 /* 19 */  i(f2v_N(xcxnsma),[fp2r(4),fp2r(7),fp2r(7)],[fp2r(4)],[]),
 /* 21 */  i(f2v_N(pnmadd),[fp2r(8),fp2r(6),fp2r(1)],[fp2r(6)],[]),
 /* 22 */  i(f2v_N(pnmsub),[fp2r(1),fp2r(9),fp2r(8)],[fp2r(1)],[]),
 /* 24 */  i(f2v_N(xmsub),[fp2r(5),fp2r(11),fp2r(4)],[fp2r(13)],[]),
 /* 25 */  i(f2v_N(xnmsub),[fp2r(4),fp2r(11),fp2r(5)],[fp2r(4)],[]),
 /* 26 */  i(f2v_N(pmul),[fp2r(10),fp2r(6)],[fp2r(10)],[]),
 /* 27 */  i(f2v_N(xmul),[fp2r(9),fp2r(1)],[fp2r(1)],[]),
 /* 29 */  i(f2v_N(pmul),[fp2r(12),fp2r(13)],[fp2r(12)],[]),
 /* 30 */  i(f2v_N(xmul),[fp2r(9),fp2r(4)],[fp2r(9)],[]),
 /* 31 */  i(f2v_st1(double,lo),[fp2r(10),r(5),r(2)],[],[d(ae(output,2,0))]),
 /* 32 */  i(f2v_st1(double,hi),[fp2r(10),r(5),r(9)],[],[d(ae(output,4,0))]),
 /* 33 */  i(f2v_st1(double,lo),[fp2r(1),r(6),r(2)],[],[d(ae(output,6,0))]),
 /* 34 */  i(f2v_st1(double,hi),[fp2r(1),r(5)],[],[d(ae(output,0,0))]),
 /* 35 */  i(f2v_st1(double,lo),[fp2r(12),r(5),r(3)],[],[d(ae(output,3,0))]),
 /* 36 */  i(f2v_st1(double,hi),[fp2r(12),r(6),r(8)],[],[d(ae(output,5,0))]),
 /* 37 */  i(f2v_st1(double,hi),[fp2r(9),r(6),r(3)],[],[d(ae(output,7,0))]),
 /* 38 */  i(f2v_st1(double,lo),[fp2r(9),r(5),r(8)],[],[d(ae(output,1,0))])
          ]
).
```

Example **fcr_8**: Assembly Output

```
.data
     .align 3
Consts__fftw_hc2real_8:
     .double 1.0, 1.0, 1.0, -1.0, 0.5, -2.0, 1.0, -0.5, 1.0, -2.0
     .double 0.70710678118654752440084436, -0.14142135623730950488016887e1
     .double 2.0, 2.0
     .double 0.14142135623730950488016887e1, 0.14142135623730950488016887e1
.text
     .align 3
     .globl _fftw_hc2real_8
_fftw_hc2real_8:
     slwi      r6, r6, 3
     addis     r12, 0, Consts__fftw_hc2real_8@ha
     slwi      r7, r7, 3
     addi      r12, r12, Consts__fftw_hc2real_8@l
     slwi      r2, r6, 2
     lfdx      f3, 0, r3
     lfpdx     f4, 0, r12
     add       r10, r7, r7
     lfsdx     f3, r3, r2
     add       r11, r10, r7
     addi      r0, 0, 16
     sub       r9, r2, r6
     lfdx      f0, r4, r7
     add       r2, r6, r6
     lfdx      f1, r4, r11
     slwi      r8, r8, 3
     lfpdux    f2, r12, r0
     fxmadd    f8, f3, f4, f3
     lfsdx     f0, r3, r9
     fxmsub    f3, f3, f4, f3
     lfsdx     f1, r3, r6
     slwi      r9, r8, 2
     lfpdux    f5, r12, r0
     add       r6, r5, r9
     lfpdux    f6, r12, r0
```

```
lfdx      f8, r3, r2
add       r2, r8, r8
fpmadd    f7, f0, f2, f1
lfpdux    f9, r12, r0
fpnmsub   f1, f1, f2, f0
lfdx      f3, r4, r10
fxnmsub   f8, f8, f5, f8
lfpdux    f11, r12, r0
fxnmsub   f5, f3, f5, f3
lfpdux    f10, r12, r0
lfpdx     f12, r12, r0
sub       r3, r9, r8
fxcxnsma  f4, f4, f7, f7
fpnmadd   f6, f8, f6, f1
fpnmsub   f1, f1, f9, f8
fxmsub    f13, f5, f11, f4
fxnmsub   f4, f4, f11, f5
fpmul     f10, f10, f6
fxmul     f1, f9, f1
fpmul     f12, f12, f13
fxmul     f9, f9, f4
stfdx     f10, r5, r2
stfsdx    f10, r5, r9
stfdx     f1, r6, r2
stfsdx    f1, 0, r5
stfdx     f12, r5, r3
stfsdx    f12, r6, r8
stfsdx    f9, r6, r3
stfdx     f9, r5, r8
blr
```

7.3.4 Complex Backward 4-point FFT (`ftwi_4`)

The following example code is another complex FFT code. Unlike `fn_3`, this code has not been vectorized directly by the MAP vectorizer. Instead, knowledge about the structure of this code has been encoded in the form of some hand-coding tricks into an auxiliary module that aids the vectorizer.

Example `ftwi_4`: Vectorization

```
procedure(fftwi_twiddle_4_inner,
          [
           array_stride_multitude_precision(inout,iostride,2,double),
           array_stride_multitude_precision('W',1,2,double)
          ],
          [
           inout,
           'W',
           iostride
          ],
          [],
          [
           i(f2v_ld2(p),[ae(inout,0,0)],[f2vr(7)],[]),
           i(f2v_ld2(p),[ae(inout,2,0)],[f2vr(24)],[]),
           i(f2v_ld2(p),[ae('W',1,0)],[f2vr(25)],[]),
           i(f2v_N(xpmul),[f2vr(25),f2vr(24)],[f2vr(26)],[]),
           i(f2v_N(xcxnsma),[f2vr(25),f2vr(24),f2vr(26)],[f2vr(6)],[]),
           i(f2v_ld2(p),[ae(inout,1,0)],[f2vr(27)],[]),
           i(f2v_ld2(p),[ae('W',0,0)],[f2vr(28)],[]),
           i(f2v_N(xpmul),[f2vr(28),f2vr(27)],[f2vr(29)],[]),
           i(f2v_N(xcxnsma),[f2vr(28),f2vr(27),f2vr(29)],[f2vr(5)],[]),
           i(f2v_ld2(p),[ae(inout,3,0)],[f2vr(30)],[]),
           i(f2v_ld2(p),[ae('W',2,0)],[f2vr(31)],[]),
           i(f2v_N(xpmul),[f2vr(31),f2vr(30)],[f2vr(32)],[]),
           i(f2v_N(xcxnsma),[f2vr(31),f2vr(30),f2vr(32)],[f2vr(4)],[]),
           i(f2v_st2(p),[f2vr(3)],[ae(inout,2,0)],[]),
           i(f2v_st2(p),[f2vr(2)],[ae(inout,0,0)],[]),
           i(f2v_st2(p),[f2vr(1)],[ae(inout,1,0)],[]),
           i(f2v_st2(p),[f2vr(0)],[ae(inout,3,0)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(13),f2vr(19)],[f2vr(23)],[]),
           i(f2v_N(swap),[f2vr(23)],[f2vr(2)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(13),f2vr(19)],[f2vr(22)],[]),
           i(f2v_N(swap),[f2vr(22)],[f2vr(3)],[]),
           i(f2v_N(paddsub(add,sub)),[f2vr(21),f2vr(16)],[f2vr(0)],[]),
           i(f2v_N(swap),[f2vr(10)],[f2vr(21)],[]),
           i(f2v_N(paddsub(sub,add)),[f2vr(20),f2vr(16)],[f2vr(1)],[]),
           i(f2v_N(swap),[f2vr(10)],[f2vr(20)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(17),f2vr(18)],[f2vr(19)],[]),
           i(f2v_N(swap),[f2vr(4)],[f2vr(18)],[]),
           i(f2v_N(swap),[f2vr(5)],[f2vr(17)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(14),f2vr(15)],[f2vr(16)],[]),
           i(f2v_N(swap),[f2vr(4)],[f2vr(15)],[]),
           i(f2v_N(swap),[f2vr(5)],[f2vr(14)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(11),f2vr(12)],[f2vr(13)],[]),
           i(f2v_N(swap),[f2vr(6)],[f2vr(12)],[]),
           i(f2v_N(swap),[f2vr(7)],[f2vr(11)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(8),f2vr(9)],[f2vr(10)],[]),
           i(f2v_N(swap),[f2vr(6)],[f2vr(9)],[]),
           i(f2v_N(swap),[f2vr(7)],[f2vr(8)],[])
          ]
).
```

`genfft` produces the following stub code that invokes the procedure compiled by the MAP compiler ("fftwi_twiddle_4_inner") as a sub-procedure within a loop.

```
void fftwi_twiddle_4(fftw_complex * A, const fftw_complex * W, int iostride, int m, int dist)
{
  int i;
  fftw_complex *inout = A;

  for (i = m; i > 0; i--, inout += dist, W += 3) { fftwi_twiddle_4_inner(inout,W,iostride); }
}

static const int twiddle_order[] = {1, 2, 3};
fftw_codelet_desc fftwi_twiddle_4_desc = {
     "fftwi_twiddle_4",
     (void (*)()) fftwi_twiddle_4,
     4,
     FFTW_BACKWARD,
     FFTW_TWIDDLE,
     99,
     3,
     twiddle_order,
};
```

Example `ftwi_4`: Peephole Optimization

```
procedure(fftwi_twiddle_4_inner,
          [
           array_stride_multitude_precision(inout,iostride,2,double),
           array_stride_multitude_precision('W',1,2,double)
          ],
          [ inout, 'W', iostride ],
          [ f2_const('(0.1e1,0.1e1)',double,'0.1e1','0.1e1') ],
          [
           i(f2v_ld2(p),[ae(inout,0,0)],[f2vr(7)],[]),
           i(f2v_ld2(p),[ae(inout,2,0)],[f2vr(24)],[]),
           i(f2v_ld2(p),[ae('W',1,0)],[f2vr(25)],[]),
           i(f2v_N(xpmul),[f2vr(25),f2vr(24)],[f2vr(26)],[]),
           i(f2v_N(xcxnsma),[f2vr(25),f2vr(24),f2vr(26)],[f2vr(6)],[]),
           i(f2v_ld2(p),[ae(inout,1,0)],[f2vr(27)],[]),
           i(f2v_ld2(p),[ae('W',0,0)],[f2vr(28)],[]),
           i(f2v_N(xpmul),[f2vr(28),f2vr(27)],[f2vr(29)],[]),
           i(f2v_N(xcxnsma),[f2vr(28),f2vr(27),f2vr(29)],[f2vr(5)],[]),
           i(f2v_ld2(p),[ae(inout,3,0)],[f2vr(30)],[]),
           i(f2v_ld2(p),[ae('W',2,0)],[f2vr(31)],[]),
           i(f2v_N(xpmul),[f2vr(31),f2vr(30)],[f2vr(32)],[]),
           i(f2v_N(xcxnsma),[f2vr(31),f2vr(30),f2vr(32)],[f2vr(4)],[]),
           i(f2v_st2(p),[f2vr(1)],[ae(inout,1,0)],[]),
           i(f2v_st2(p),[f2vr(0)],[ae(inout,3,0)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(5),f2vr(4)],[f2vr(103)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(5),f2vr(4)],[f2vr(104)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(7),f2vr(6)],[f2vr(105)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(7),f2vr(6)],[f2vr(106)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(106),f2vr(104)],[f2vr(107)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(106),f2vr(104)],[f2vr(108)],[]),
           i(f2v_st2(p),[f2vr(108)],[ae(inout,2,0)],[]),
           i(f2v_st2(p),[f2vr(107)],[ae(inout,0,0)],[]),
           i(f2v_ld2(p),[c('(0.1e1,0.1e1)')],[f2vr(1034)],[]),
           i(f2v_N(xcxnsma),[f2vr(1034),f2vr(103),f2vr(105)],[f2vr(0)],[]),
           i(f2v_N(xcxnpma),[f2vr(1034),f2vr(103),f2vr(105)],[f2vr(1)],[])
          ]
).
```

Example **ftwi_4**: High-Level Scheduling

```
procedure(fftwi_twiddle_4_inner,
          [
           array_stride_multitude_precision(inout,iostride,2,double),
           array_stride_multitude_precision('W',1,2,double)
          ],
          [ inout, 'W', iostride ],
          [ f2_const('(0.1e1,0.1e1)',double,'0.1e1','0.1e1') ],
          [
           i(f2v_ld2(p),[ae('W',2,0)],[f2vr(31)],[]),
           i(f2v_ld2(p),[ae(inout,3,0)],[f2vr(30)],[]),
           i(f2v_N(xpmul),[f2vr(31),f2vr(30)],[f2vr(32)],[]),
           i(f2v_N(xcxnsma),[f2vr(31),f2vr(30),f2vr(32)],[f2vr(4)],[]),
           i(f2v_ld2(p),[ae('W',0,0)],[f2vr(28)],[]),
           i(f2v_ld2(p),[ae(inout,1,0)],[f2vr(27)],[]),
           i(f2v_N(xpmul),[f2vr(28),f2vr(27)],[f2vr(29)],[]),
           i(f2v_N(xcxnsma),[f2vr(28),f2vr(27),f2vr(29)],[f2vr(5)],[]),
           i(f2v_ld2(p),[ae('W',1,0)],[f2vr(25)],[]),
           i(f2v_ld2(p),[ae(inout,2,0)],[f2vr(24)],[]),
           i(f2v_N(xpmul),[f2vr(25),f2vr(24)],[f2vr(26)],[]),
           i(f2v_N(xcxnsma),[f2vr(25),f2vr(24),f2vr(26)],[f2vr(6)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(5),f2vr(4)],[f2vr(104)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(5),f2vr(4)],[f2vr(103)],[]),
           i(f2v_ld2(p),[ae(inout,0,0)],[f2vr(7)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(7),f2vr(6)],[f2vr(106)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(7),f2vr(6)],[f2vr(105)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(106),f2vr(104)],[f2vr(107)],[]),
           i(f2v_st2(p),[f2vr(107)],[ae(inout,0,0)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(106),f2vr(104)],[f2vr(108)],[]),
           i(f2v_st2(p),[f2vr(108)],[ae(inout,2,0)],[]),
           i(f2v_ld2(p),[c('(0.1e1,0.1e1)')],[f2vr(1034)],[]),
           i(f2v_N(xcxnsma),[f2vr(1034),f2vr(103),f2vr(105)],[f2vr(0)],[]),
           i(f2v_st2(p),[f2vr(0)],[ae(inout,3,0)],[]),
           i(f2v_N(xcxnpma),[f2vr(1034),f2vr(103),f2vr(105)],[f2vr(1)],[]),
           i(f2v_st2(p),[f2vr(1)],[ae(inout,1,0)],[])
          ]
).
```

Example ftwi_4: Medium-Level Scheduling

```
procedure(fftwi_twiddle_4_inner,
          [
           array_stride_multitude_precision(inout,iostride,2,double),
           array_stride_multitude_precision('W',1,2,double)
          ],
          [
           inout,
           'W',
           iostride
          ],
          [
           f2_const('(0.1e1,0.1e1)',double,'0.1e1','0.1e1')
          ],
          [
           i(f2v_ld2(p),[ae('W',2,0)],[f2vr(31)],[]),
           i(f2v_ld2(p),[ae(inout,3,0)],[f2vr(30)],[]),
           i(f2v_ld2(p),[ae('W',0,0)],[f2vr(28)],[]),
           i(f2v_ld2(p),[ae(inout,1,0)],[f2vr(27)],[]),
           i(f2v_ld2(p),[ae('W',1,0)],[f2vr(25)],[]),
           i(f2v_N(xpmul),[f2vr(31),f2vr(30)],[f2vr(32)],[]),
           i(f2v_ld2(p),[ae(inout,2,0)],[f2vr(24)],[]),
           i(f2v_ld2(p),[ae(inout,0,0)],[f2vr(7)],[]),
           i(f2v_N(xpmul),[f2vr(28),f2vr(27)],[f2vr(29)],[]),
           i(f2v_ld2(p),[c('(0.1e1,0.1e1)')],[f2vr(1034)],[]),
           i(f2v_N(xpmul),[f2vr(25),f2vr(24)],[f2vr(26)],[]),
           i(f2v_N(xcxnsma),[f2vr(31),f2vr(30),f2vr(32)],[f2vr(4)],[]),
           i(f2v_N(xcxnsma),[f2vr(28),f2vr(27),f2vr(29)],[f2vr(5)],[]),
           i(f2v_N(xcxnsma),[f2vr(25),f2vr(24),f2vr(26)],[f2vr(6)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(5),f2vr(4)],[f2vr(104)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(5),f2vr(4)],[f2vr(103)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(7),f2vr(6)],[f2vr(106)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(7),f2vr(6)],[f2vr(105)],[]),
           i(f2v_N(paddsub(add,add)),[f2vr(106),f2vr(104)],[f2vr(107)],[]),
           i(f2v_N(paddsub(sub,sub)),[f2vr(106),f2vr(104)],[f2vr(108)],[]),
           i(f2v_N(xcxnsma),[f2vr(1034),f2vr(103),f2vr(105)],[f2vr(0)],[]),
           i(f2v_N(xcxnpma),[f2vr(1034),f2vr(103),f2vr(105)],[f2vr(1)],[]),
           i(f2v_st2(p),[f2vr(107)],[ae(inout,0,0)],[]),
           i(f2v_st2(p),[f2vr(108)],[ae(inout,2,0)],[]),
           i(f2v_st2(p),[f2vr(0)],[ae(inout,3,0)],[]),
           i(f2v_st2(p),[f2vr(1)],[ae(inout,1,0)],[])
          ]
 ).
```

Example `ftwi_4`: Floating-Point Register Allocation

```
procedure(fftwi_twiddle_4_inner,
          [
           array_stride_multitude_precision(inout,iostride,2,double),
           array_stride_multitude_precision('W',1,2,double)
          ],
          [
           inout,
           'W',
           iostride
          ],
          [
           f2_const('(0.1e1,0.1e1)',double,'0.1e1','0.1e1')
          ],
          [
           i(f2v_ld2(p),[ae('W',2,0)],[fp2r(0)],[]),
           i(f2v_ld2(p),[ae(inout,3,0)],[fp2r(1)],[]),
           i(f2v_ld2(p),[ae('W',0,0)],[fp2r(2)],[]),
           i(f2v_ld2(p),[ae(inout,1,0)],[fp2r(3)],[]),
           i(f2v_ld2(p),[ae('W',1,0)],[fp2r(4)],[]),
           i(f2v_N(xpmul),[fp2r(0),fp2r(1)],[fp2r(5)],[]),
           i(f2v_ld2(p),[ae(inout,2,0)],[fp2r(6)],[]),
           i(f2v_ld2(p),[ae(inout,0,0)],[fp2r(7)],[]),
           i(f2v_N(xpmul),[fp2r(2),fp2r(3)],[fp2r(8)],[]),
           i(f2v_ld2(p),[c('(0.1e1,0.1e1)')],[fp2r(9)],[]),
           i(f2v_N(xpmul),[fp2r(4),fp2r(6)],[fp2r(10)],[]),
           i(f2v_N(xcxnsma),[fp2r(0),fp2r(1),fp2r(5)],[fp2r(0)],[]),
           i(f2v_N(xcxnsma),[fp2r(2),fp2r(3),fp2r(8)],[fp2r(2)],[]),
           i(f2v_N(xcxnsma),[fp2r(4),fp2r(6),fp2r(10)],[fp2r(4)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(2),fp2r(0)],[fp2r(11)],[]),
           i(f2v_N(paddsub(sub,sub)),[fp2r(2),fp2r(0)],[fp2r(2)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(7),fp2r(4)],[fp2r(12)],[]),
           i(f2v_N(paddsub(sub,sub)),[fp2r(7),fp2r(4)],[fp2r(7)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(12),fp2r(11)],[fp2r(13)],[]),
           i(f2v_N(paddsub(sub,sub)),[fp2r(12),fp2r(11)],[fp2r(12)],[]),
           i(f2v_N(xcxnsma),[fp2r(9),fp2r(2),fp2r(7)],[fp2r(1)],[]),
           i(f2v_N(xcxnpma),[fp2r(9),fp2r(2),fp2r(7)],[fp2r(9)],[]),
           i(f2v_st2(p),[fp2r(13)],[ae(inout,0,0)],[]),
           i(f2v_st2(p),[fp2r(12)],[ae(inout,2,0)],[]),
           i(f2v_st2(p),[fp2r(1)],[ae(inout,3,0)],[]),
           i(f2v_st2(p),[fp2r(9)],[ae(inout,1,0)],[])
          ]
).
```

Example `ftwi_4`: Effective Address Generation

```
procedure(fftwi_twiddle_4_inner,
          [
           array_stride_multitude_precision(inout,iostride,2,double),
           array_stride_multitude_precision('W',1,2,double)
          ],
          [
           inout,
           'W',
           iostride
          ],
          [
           f2_const(0,double,'0.1e1','0.1e1')
          ],
          [
           i(i_shiftLeft(4),[r(5)],[r(5)],[]),
           i(i_addIS('Consts__fftwi_twiddle_4_inner@ha'),[imm(0)],[r(12)],[]),
           i(i_addI('Consts__fftwi_twiddle_4_inner@l'),[r(12)],[r(12)],[]),
           i(i_copyI(32),[],[r(2)],[]),
           i(f2v_ld2(double,p),[r(4),r(2)],[fp2r(0),r(4)],[s(ae('W',2,0))]),
           i(i_add,[r(5),r(5)],[r(6)],[]),
           i(i_add,[r(6),r(5)],[r(7)],[]),
           i(f2v_ld2(double,p),[r(3),r(7)],[fp2r(1)],[s(ae(inout,3,0))]),
           i(i_copyI(-32),[],[r(8)],[]),
           i(f2v_ld2(double,p),[r(4),r(8)],[fp2r(2)],[s(ae('W',0,0))]),
           i(f2v_ld2(double,p),[r(3),r(5)],[fp2r(3)],[s(ae(inout,1,0))]),
           i(i_copyI(-16),[],[r(9)],[]),
           i(f2v_ld2(double,p),[r(4),r(9)],[fp2r(4),r(4)],[s(ae('W',1,0))]),
           i(f2v_ld2(double,p),[r(3),r(6)],[fp2r(6)],[s(ae(inout,2,0))]),
           i(f2v_N(xpmul),[fp2r(0),fp2r(1)],[fp2r(5)],[]),
           i(f2v_ld2(double,p),[r(3)],[fp2r(7)],[s(ae(inout,0,0))]),
           i(f2v_N(xpmul),[fp2r(2),fp2r(3)],[fp2r(8)],[]),
           i(f2v_ld2(double,p),[r(12)],[fp2r(9)],[s(const(0))]),
           i(f2v_N(xpmul),[fp2r(4),fp2r(6)],[fp2r(10)],[]),
           i(f2v_N(xcxnsma),[fp2r(0),fp2r(1),fp2r(5)],[fp2r(0)],[]),
           i(f2v_N(xcxnsma),[fp2r(2),fp2r(3),fp2r(8)],[fp2r(2)],[]),
           i(f2v_N(xcxnsma),[fp2r(4),fp2r(6),fp2r(10)],[fp2r(4)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(2),fp2r(0)],[fp2r(11)],[]),
           i(f2v_N(paddsub(sub,sub)),[fp2r(2),fp2r(0)],[fp2r(2)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(7),fp2r(4)],[fp2r(12)],[]),
           i(f2v_N(paddsub(sub,sub)),[fp2r(7),fp2r(4)],[fp2r(7)],[]),
           i(f2v_N(paddsub(add,add)),[fp2r(12),fp2r(11)],[fp2r(13)],[]),
           i(f2v_N(paddsub(sub,sub)),[fp2r(12),fp2r(11)],[fp2r(12)],[]),
           i(f2v_N(xcxnsma),[fp2r(9),fp2r(2),fp2r(7)],[fp2r(1)],[]),
           i(f2v_N(xcxnpma),[fp2r(9),fp2r(2),fp2r(7)],[fp2r(9)],[]),
           i(f2v_st2(double,p),[fp2r(13),r(3)],[],[d(ae(inout,0,0))]),
           i(f2v_st2(double,p),[fp2r(12),r(3),r(6)],[],[d(ae(inout,2,0))]),
           i(f2v_st2(double,p),[fp2r(1),r(3),r(7)],[],[d(ae(inout,3,0))]),
           i(f2v_st2(double,p),[fp2r(9),r(3),r(5)],[],[d(ae(inout,1,0))])
          ]
).
```

Example ftwi_4: List-based Instruction Scheduling

```
procedure(fftwi_twiddle_4_inner,
          [
           array_stride_multitude_precision(inout,iostride,2,double),
           array_stride_multitude_precision('W',1,2,double)
          ],
          [
           inout,
           'W',
           iostride
          ],
          [
           f2_const(0,double,'0.1e1','0.1e1')
          ],
          [
  /* 0 */  i(i_shiftLeft(4),[r(5)],[r(5)],[]),
  /* 0 */  i(i_copyI(32),[],[r(2)],[]),
  /* 1 */  i(i_add,[r(5),r(5)],[r(6)],[]),
  /* 1 */  i(i_copyI(-32),[],[r(8)],[]),
  /* 2 */  i(i_copyI(-16),[],[r(9)],[]),
  /* 2 */  i(i_add,[r(6),r(5)],[r(7)],[]),
  /* 3 */  i(f2v_ld2(double,p),[r(4),r(2)],[fp2r(0),r(4)],[s(ae('W',2,0))]),
  /* 3 */  i(i_addIS('Consts__fftwi_twiddle_4_inner@ha'),[imm(0)],[r(12)],[]),
  /* 4 */  i(f2v_ld2(double,p),[r(3),r(5)],[fp2r(3)],[s(ae(inout,1,0))]),
  /* 4 */  i(i_addI('Consts__fftwi_twiddle_4_inner@l'),[r(12)],[r(12)],[]),
  /* 5 */  i(f2v_ld2(double,p),[r(3),r(6)],[fp2r(6)],[s(ae(inout,2,0))]),
  /* 6 */  i(f2v_ld2(double,p),[r(3),r(7)],[fp2r(1)],[s(ae(inout,3,0))]),
  /* 7 */  i(f2v_ld2(double,p),[r(4),r(8)],[fp2r(2)],[s(ae('W',0,0))]),
  /* 8 */  i(f2v_ld2(double,p),[r(4),r(9)],[fp2r(4)],[s(ae('W',1,0))]),
  /* 9 */  i(f2v_ld2(double,p),[r(3)],[fp2r(7)],[s(ae(inout,0,0))]),
 /* 10 */  i(f2v_N(xpmul),[fp2r(0),fp2r(1)],[fp2r(5)],[]),
 /* 10 */  i(f2v_ld2(double,p),[r(12)],[fp2r(9)],[s(const(0))]),
 /* 11 */  i(f2v_N(xpmul),[fp2r(2),fp2r(3)],[fp2r(8)],[]),
 /* 12 */  i(f2v_N(xpmul),[fp2r(4),fp2r(6)],[fp2r(10)],[]),
 /* 15 */  i(f2v_N(xcxnsma),[fp2r(0),fp2r(1),fp2r(5)],[fp2r(0)],[]),
 /* 16 */  i(f2v_N(xcxnsma),[fp2r(2),fp2r(3),fp2r(8)],[fp2r(2)],[]),
 /* 17 */  i(f2v_N(xcxnsma),[fp2r(4),fp2r(6),fp2r(10)],[fp2r(4)],[]),
 /* 21 */  i(f2v_N(paddsub(add,add)),[fp2r(2),fp2r(0)],[fp2r(11)],[]),
 /* 22 */  i(f2v_N(paddsub(add,add)),[fp2r(7),fp2r(4)],[fp2r(12)],[]),
 /* 23 */  i(f2v_N(paddsub(sub,sub)),[fp2r(2),fp2r(0)],[fp2r(2)],[]),
 /* 24 */  i(f2v_N(paddsub(sub,sub)),[fp2r(7),fp2r(4)],[fp2r(7)],[]),
 /* 27 */  i(f2v_N(paddsub(add,add)),[fp2r(12),fp2r(11)],[fp2r(13)],[]),
 /* 28 */  i(f2v_N(paddsub(sub,sub)),[fp2r(12),fp2r(11)],[fp2r(12)],[]),
 /* 29 */  i(f2v_N(xcxnsma),[fp2r(9),fp2r(2),fp2r(7)],[fp2r(1)],[]),
 /* 30 */  i(f2v_N(xcxnpma),[fp2r(9),fp2r(2),fp2r(7)],[fp2r(9)],[]),
 /* 32 */  i(f2v_st2(double,p),[fp2r(13),r(3)],[],[d(ae(inout,0,0))]),
 /* 33 */  i(f2v_st2(double,p),[fp2r(12),r(3),r(6)],[],[d(ae(inout,2,0))]),
 /* 34 */  i(f2v_st2(double,p),[fp2r(1),r(3),r(7)],[],[d(ae(inout,3,0))]),
 /* 35 */  i(f2v_st2(double,p),[fp2r(9),r(3),r(5)],[],[d(ae(inout,1,0))])
          ]
).
```

Example ftwi_4: Assembly Output

```
.data
     .align 3
Consts__fftwi_twiddle_4_inner:
     .double 0.1e1, 0.1e1
.text
     .align 3
     .globl _fftwi_twiddle_4_inner
_fftwi_twiddle_4_inner:
     slwi      r5, r5, 4
     addi      r2, 0, 32
     add       r6, r5, r5
     addi      r8, 0, -32
     addi      r9, 0, -16
     add       r7, r6, r5
     lfpdux    f0, r4, r2
     addis     r12, 0, Consts__fftwi_twiddle_4_inner@ha
     lfpdx     f3, r3, r5
     addi      r12, r12, Consts__fftwi_twiddle_4_inner@l
     lfpdx     f6, r3, r6
     lfpdx     f1, r3, r7
     lfpdx     f2, r4, r8
     lfpdx     f4, r4, r9
     lfpdx     f7, 0, r3
     fxpmul    f5, f0, f1
     lfpdx     f9, 0, r12
     fxpmul    f8, f2, f3
     fxpmul    f10, f4, f6
     fxcxnsma  f0, f0, f1, f5
     fxcxnsma  f2, f2, f3, f8
     fxcxnsma  f4, f4, f6, f10
     fpadd     f11, f2, f0
     fpadd     f12, f7, f4
     fpsub     f2, f2, f0
     fpsub     f7, f7, f4
     fpadd     f13, f12, f11
     fpsub     f12, f12, f11
     fxcxnsma  f1, f9, f2, f7
     fxcxnpma  f9, f9, f2, f7
     stfpdx    f13, 0, r3
     stfpdx    f12, r3, r6
     stfpdx    f1, r3, r7
     stfpdx    f9, r3, r5
     blr
```

Chapter 8

Results

To assess the performance impact of the presented techniques on real Blue Gene systems, the compute-intensive numerical kernels of FFTW 2.1.5 were compiled using the following setups. (*i*) *xlc_scalar* uses the XL C compiler without automatic vectorization. (*ii*) *xlc_vect* uses XL C with automatic vectorization. (*iii*) *xlc_mapvect* uses the MAP vectorizer and optimizer, producing C code with SIMD intrinsics compiled by XL C. (*iv*) *map_vect* uses the MAP vectorizer, optimizer, and backend.

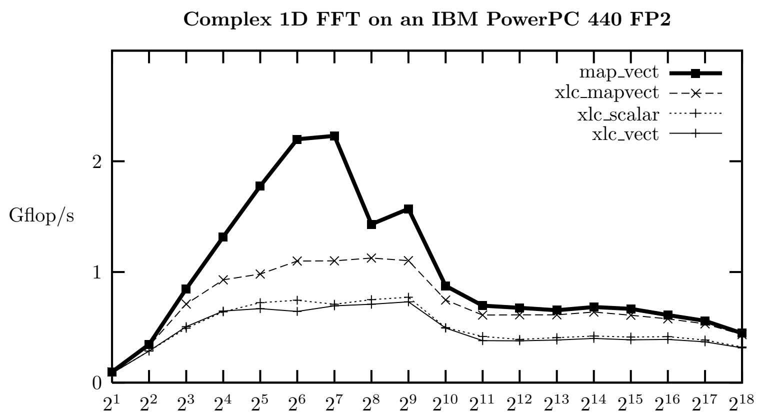

Figure 8.1: Complex Power-of-two 1D FFT Performance on Blue Gene.

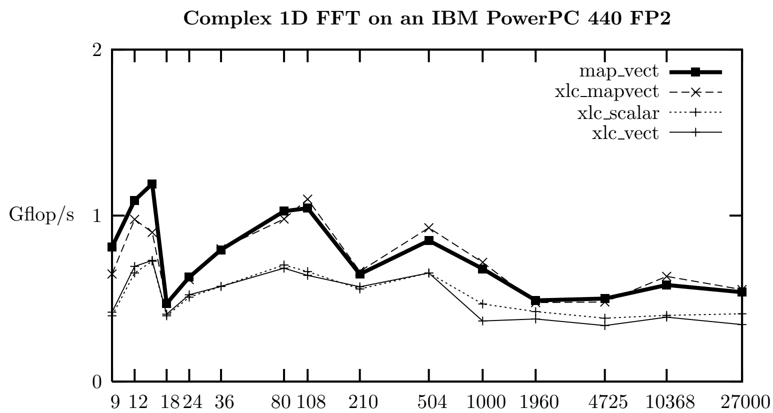

Figure 8.2: Complex Non-power-of-two 1D FFT Performance on Blue Gene.

Figs. 8.1 and 8.2 show the single-processor complex FFT performance achieved on Blue Gene systems by using various compilers and settings. In both cases, the MAP vectorizer and optimizer improve the floating-point performance significantly. For complex power-of-two length FFTs, the MAP backend gives an additional performance boost.

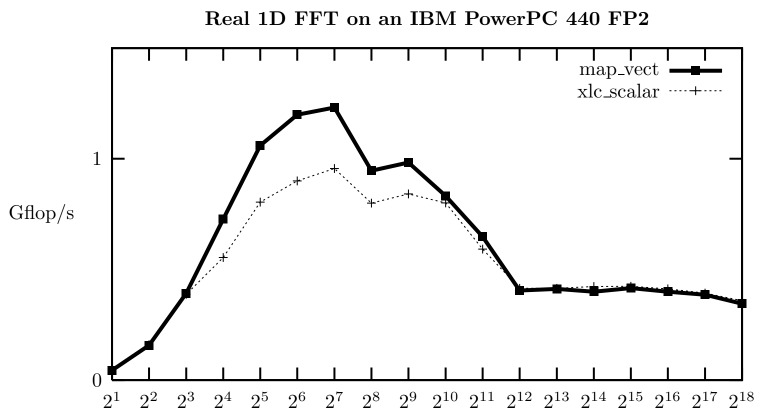

Figure 8.3: Real Power-of-two 1D FFT Performance on Blue Gene.

Fig. 8.3 shows performance of real 1D FFTs of vectors with a power-of-two length. The MAP tool chain improves performance, but the effect is noticably smaller than for complex FFT codes.

Discussion

Performance of Large Transforms. In all figures shown, it is noticable that the performance falls sharply as soon as cache capacity boundaries are hit, i. e., as soon as the size of data sets no longer fits into the cache.

Performance of Complex Non-power-of-two Sizes. It is noticeable that the backend does not yield significant performances gain in the non-power-of-two case (Fig. 8.2). Experiments show that a comparable performance level as in the power-of-two case could be obtained, if a large number of additional numerical kernels was integrated into the library. Because of pragmatic reasons—many FFTW users primarily calculate power-of-two complex FFTs— the library was kept slim, and the additionally needed kernels were not included.

Performance of Real Transforms. Real transforms have a larger number of memory access instructions compared to complex transforms operating on interleaved complex data. Also, the optimal SIMD vectorization of real FFT kernels is not always possible. Thus, the speedup for real FFTs is lower than for complex FFTs.

Performance of Multi-Dimensional Transforms. Internally, FFTW reduces any multi-dimensional transform to multiple one-dimensional transforms done in a row-column style. As early tests indicated, it would make sense to generate specialized codes optimized for the case when all array data do not reside in L1 cache but in L2 cache, future work will focus on specifically optimizing these cases as well.

Instruction Count. For all codes investigated, the MAP vectorizer and optimizer for Blue Gene systems significantly reduced the instruction count by utilizing FP2 SIMD instructions. While the biggest part of the gain can be attributed to vectorization, the optimizer also contributed to code quality by utilizing FP2 specific instructions, thus eliminating many SIMD swaps and multiplications.

For SIMD codes the address generation part of the backend improves the code quality by minimizing the number of integer instructions.

As FFTW kernels can be very large, minimizing the instruction count helps avoid hitting L1 instruction cache capacity limits.

Effect of the Backend. Experiments to find out the performance contribution of the compiler backend (*xlc_mapvect* vs. *map_vect*) show that the MAP backend produces much better code for compilation units consisting of one large basic block, while XL C profits from being able to perform its optimizations on units larger than one basic block by utilizing techniques like loop unrolling.

Summary

The MAP compiler tool chain covers *all* stages of compilation that are important for achieving high performance in numerical software for linear signal processing transforms.

First, the code produced by a special purpose program generator, like FFTW's `genfft`, is vectorized, seeking an optimal utilization of the 2-way SIMD floating-point unit of IBM's PowerPC 440 FP2 processors.

Next, the MAP optimizer tries to minimize SIMD data reordering overhead and maximize utilization of FMAs and other FP2 specific idioms.

Finally, the code is compiled down to assembly, using (*i*) an optimal algorithm for register allocation for basic blocks, (*ii*) several levels of scheduling, and (*iii*) a clever instruction selection method for dealing with effective address generation on a processor with DSP-like addressing mode restrictions.

Superior Performance Level. In the best cases, code produced by the MAP compiler runs at 80% of the performance that the best algorithm known in the literature could theoretically achieve on the target hardware.

MAP compiled FFTW codelets enabled the material science code Qbox [39] to run with a sustained performance of 60 Tflop/s on BlueGene/L, thus reaching the second highest performance ever achieved by an application code. Recent advancements have allowed to reach significantly higher performance values.

Appendix A

MAP Backend Source Codes

The appendix presents a selection of OCAML source codes of two generic components of the MAP backend, the flexible list-based instruction scheduler `LL1` and the register allocator—along with the auxiliary modules they require.

A.1 Generic Auxiliary Modules

Commonly used functions that are not adequately covered by the OCAML standard library are implemented in the following modules.

A.1.1 `basics.ml`: Basic Functions

```
open List
open Num

type precision = Single | Double

let string_of_precision = function Single -> "single" | Double -> "double"

let precision_of_string = function
  | "single" -> Single
  | "double" -> Double
  | s -> failwith ("precision_of_string(" ^ s ^")")

module IntMap    = Map.Make(struct type t = int let compare = compare end)
module StringMap = Map.Make(struct type t = string let compare = compare end)
module StringSet = Set.Make(struct type t = string let compare = compare end)

let hdTl' = function [] -> failwith "hdTl'" | [x] | _::x::_ -> x

let subst y z x = if x = y then z else x

let overlap (xs,ys) = exists (fun x -> mem x ys) xs

let uniq xs = match xs with
  | [] | [_] -> xs
  | a::([b] as ys) -> if a=b then ys else xs
  | _ -> fold_right (fun x xs -> if mem x xs then xs else x::xs) xs []

let identity x = x

let option_is_some = function Some _ -> true | None -> false
let value_of_option = function Some x -> x | None -> failwith "value_of_option"

(* incr' is like the built-in function incr, but returns the old value *)
let incr' a_ref = let a_val = !a_ref in incr a_ref; a_val

let rec zs_from_to' i j zs = if i>j then zs else zs_from_to' i (j-1) (j::zs)
let ints_from_to z0 z = zs_from_to' z0 z []
```

```
let sel3and4of4uniq _ _ x y = (uniq x, uniq y)
let sel3of4uniq _ _ x _ = uniq x
let get3of3 (_,_,x) = x

let pick_maximum eval = function
  | [] -> failwith "pick_maximum([])"
  | x::xs ->
      let x_val = eval x in
        let rec loop max_val max_el = function
          | [] -> (max_val,max_el)
          | x::xs ->
              let x_val = eval x in
                if x_val > max_val then loop x_val x xs
                else loop max_val max_el xs
        in loop x_val x xs

let optionalCons x xs = match x with Some x -> x::xs | None -> xs

let list_of_optionList xs = fold_right optionalCons xs []

let msb x =
  let rec msb_internal msb0 = function
    | 0 -> msb0
    | n -> msb_internal (msb0+1) (n lsr 1) in
  msb_internal (-1) x

let find_elem p xs = try Some (List.find p xs) with Not_found -> None

let toFalse _ = false
let toTrue  _ = true

let get1of3 (x,_,_) = x
let get2of3 (_,x,_) = x
let get3of3 (_,_,x) = x

let get1of4 (x,_,_,_) = x
let get2of4 (_,x,_,_) = x
let get3of4 (_,_,x,_) = x
let get4of4 (_,_,_,x) = x

let queue_of_list xs =
  let q = Queue.create ()
  in List.iter (fun x -> Queue.add x q) xs; q

let rec mapOptionWithN n f = function
  | [] -> []
  | x::xs ->
      match f n x with
        | Some v -> (v,x)::(mapOptionWithN (n+1) f xs)
        | None -> mapOptionWithN (n+1) f xs

let swapPair (a,b) = (b,a)

let one  = num_of_int 1
let zero = num_of_int 0
let mone = minus_num one
let ten  = num_of_int 10
let negative x = x </ zero

(* decimal digits of precision to maintain internally, and to print out *)
let precision = 50
let print_precision = 25

let inveps = ten **/ (Int precision)
let epsilon = one // inveps
```

```
let pinveps = ten **/ (Int print_precision)
let pepsilon = one // pinveps

let round x = epsilon */ (round_num (x */ inveps))

(* comparison predicate for real numbers *)
let num_equal x y = (* use both relative and absolute error *)
  let absdiff = abs_num (x -/ y) in
  absdiff <=/ pepsilon or
  absdiff <=/ pepsilon */ (abs_num x +/ abs_num y)

let takeNth0_of n =
  let rec takeNth0_of' bag n = function
    | [] -> failwith "takeNth0_of"
    | x::xs when n = 0 -> (x,List.rev_append bag xs)
    | x::xs -> takeNth0_of' (x::bag) (n-1) xs
  in takeNth0_of' [] n

let sliceNth0_of n =
  let rec sliceNth0_of' bag n = function
    | [] -> failwith "sliceNth0_of"
    | x::xs when n = 0 -> (bag,x,xs)
    | x::xs -> sliceNth0_of' (x::bag) (n-1) xs
  in sliceNth0_of' [] n

let slices_of p ys =
  let rec slices_of' bag = function
    | [] -> raise Not_found
    | x::xs -> if p x then (bag,x,xs) else slices_of' (x::bag) xs
  in slices_of' [] ys
```

A.1.2 `id.ml`: Identifiers

```
open List

module Id : sig
  type t

  val makeNew  : unit -> t
  val equal    : t -> t -> bool
  val compare  : t -> t -> int
  val toString : t -> string
end = struct
  type t = ID of int

  let makeNew =
    let currentId = ref 0 in
      fun () ->
        incr currentId;
        ID !currentId

  let toString (ID i) = Printf.sprintf "id(%d)" i
  let equal = (=)
  let compare = compare
end

module IdSet = Set.Make(Id)
module IdMap = Map.Make(Id)

let idmapToXs keymap_addE valueToXs =
  IdMap.fold
    (fun id value ->
       fold_right (fun key -> keymap_addE key id) (valueToXs value))
```

```
let listToIdmap xs =
  fold_right (fun x -> IdMap.add (Id.makeNew ()) x) xs IdMap.empty

let idmap_exists k m = try IdMap.find k m; true with Not_found -> false
let idmap_inc key map = IdMap.add key (IdMap.find key map + 1) map

(* add to list of existing entries *)
let idmap_findE k m = try IdMap.find k m with Not_found -> []
let idmap_addE k v m = IdMap.add k (v::idmap_findE k m) m
let idmap_addE' value key = idmap_addE key value
```

A.1.3 `nonDetMonad.ml`: Simulating Backtracking

```
open Util

let identityM a state = Some(a,state)

let unitM a state cont = cont a state

let (>>=) = fun f1 f2 -> fun state cont ->
  f1 state (fun a state' -> f2 a state' cont)

let fetchStateM state cont = cont state state
let storeStateM state _oldstate cont = cont () state

let catch1M exn handler state cont =
  try cont () state with exn -> handler () state cont

let (|||) = fun f1 f2 ->
  fun state cont ->
    match f1 () state cont with
      | None           -> f2 () state cont
      | Some(a',state') -> Some(a',state')

let failM _state _cont = None

let difM a b = if a <> b then unitM () else failM

let runM f x state = f x state identityM
let runP f x state = optionIsSome (runM f x state)

let posAssertM = function true  -> unitM () | false -> failM
let negAssertM = function false -> unitM () | true  -> failM

(****************************************************************************)

let oneOf2M x y state cont =
  match cont x state with
    | Some x -> Some x
    | None -> cont y state

(* avoid creation of choicepoint for last member of a list *)
let rec memberM' el xs state cont = match xs with
  | []    -> cont el state
  | x::xs ->
      match cont el state with
        | None -> memberM' x xs state cont
        | Some x -> Some x

let memberM xs state cont = match xs with
  | []    -> None
  | x::xs -> memberM' x xs state cont
```

```
let rec enumNth0MemberM' el n0 = function
  | []    -> unitM (n0,el)
  | x::xs -> (fun _ -> unitM (n0,el)) |||
             (fun _ -> enumNth0MemberM' x (n0+1) xs)

let enumNth0MemberM = function
  | []    -> failM
  | x::xs -> enumNth0MemberM' x 0 xs

let mapPairM f (a,b) =
  f a >>= fun a' ->
    f b >>= fun b' ->
      unitM (a',b')

let mapTripleM f (a,b,c) =
  f a >>= fun a' ->
    f b >>= fun b' ->
      f c >>= fun c' ->
        unitM (a',b',c')

let consM x xs = unitM (x::xs)

let rec mapM f = function
  | []    -> unitM []
  | x::xs -> f x >>= fun x' ->
               mapM f xs >>= fun xs' ->
                 unitM (x'::xs')

let optionToValueM = function
  | None   -> failM
  | Some x -> unitM x

let rec iterM f = function
  | []    -> unitM ()
  | x::xs -> f x >>= fun _ -> iterM f xs

let rec iterirevM f i = function
  | []    -> unitM ()
  | x::xs -> iterirevM f (succ i) xs >>= fun _ -> f i x

(* aux. predicate to avoid creation of choicepoint for the last element *)
let rec selectM' el = function
  | [] -> unitM (el,[])
  | x::xs as xxs ->
      (fun _ -> unitM (el,xxs)) |||
      (fun _ -> selectM' x xs >>= fun (z,zs) -> unitM (z,el::zs))

let selectM = function
  | []    -> failM
  | x::xs -> selectM' x xs

let rec selectFirstM p = function
  | []             -> failM
  | x::xs when p x -> unitM (x,xs)
  | x::xs          -> selectFirstM p xs >>= fun (z,zs) -> unitM (z,x::zs)

let deleteFirstM p xs = selectFirstM p xs >>= fun (_,zs) -> unitM zs

let rec permutationM = function
  | [] -> unitM []
  | xs -> selectM xs >>= fun (z,zs) -> permutationM zs >>= consM z

let rec forallM p = function
  | []    -> unitM ()
  | x::xs -> p x >>= fun _ -> forallM p xs
```

```
let existsM p xs = memberM xs >>= p

(* auxiliary function to avoid creation of choicepoint *)
let rec betweenM' i0 i1 i =
  if i1 <= i then (fun _ -> unitM i0) ||| (fun _ -> betweenM' i1 (succ i1) i)
  else unitM i0

let betweenM i0 i =
  if i0 <= i then betweenM' i0 (succ i0) i
  else failM

let disjM xs = memberM xs >>= fun f -> f ()
```

A.1.4 `unparsing.ml`: Generic Unparsing

```
open Basics
open Buffer

let string_of_precision = function Single -> "single" | Double -> "double"
let char_of_precision = function Single -> 's' | Double -> 'd'
let shortstring_of_precision = function Single -> "s" | Double -> ""

let add_precision b p = add_char b (char_of_precision p)

let add_int b i = add_string b (string_of_int i)

let add_list' s_lbracket s_comma s_rbracket addOne b = function
  | [] ->
      add_string b s_lbracket;
      add_string b s_rbracket
  | x::xs ->
      add_string b s_lbracket;
      addOne     b x;
      let rec add_list'' = function
        | []    -> add_string b s_rbracket
        | x::xs -> add_string b s_comma; addOne b x; add_list'' xs
      in add_list'' xs

let add_list addOne b xs   = add_list' "[\n\t " ",\n\t " "\n\t]" addOne b xs
let add_listC addOne b xs  = add_list' "[" "," "]" addOne b xs
let add_listC' addOne b xs = add_list' "" "," "" addOne b xs

let add_decimal = add_int
let add_decimals b xs  = add_list add_decimal b xs
let add_decimalsC b xs = add_listC add_decimal b xs

let add_lit b lit  = add_char b '\''; add_string b lit; add_char b '\''
let add_lits b xs  = add_list add_lit b xs
let add_litsC b xs = add_listC add_lit b xs

let add_litpair b (l1,l2) = add_lit  b l1; add_char b '-'; add_lit b l2
let add_litpairs b xs     = add_list add_litpair b xs
let add_litpairsC b xs    = add_listC add_litpair b xs

let add_someString string_of_one b x = add_string b (string_of_one x)

let string_of_something add_something x =
  let buf = Buffer.create 16 in add_something buf x; Buffer.contents buf

let string_of_listC' string_of_one xs =
  string_of_something (add_listC' (add_someString string_of_one)) xs
```

A.1.5 `util.ml`: Other Utilities

```
open List
open Unix

(****************************************
 * Integer operations
 ****************************************)
(* fint the inverse of n modulo m *)
let invmod n m =
    let rec loop i =
        if ((i * n) mod m == 1) then i
        else loop (i + 1)
    in
        loop 1

(* Yooklid's algorithm *)
let rec gcd n m =
    if (n > m)
      then gcd m n
    else
      let r = m mod n
      in
          if (r == 0) then n
          else gcd r n

(* reduce the fraction m/n to lowest terms, modulo factors of n/n *)
let lowest_terms n m =
    if (m mod n == 0) then
      (1,0)
    else
      let nn = (abs n) in let mm = m * (n / nn)
      in let mpos =
          if (mm > 0) then (mm mod nn)
          else (mm + (1 + (abs mm) / nn) * nn) mod nn
      and d = gcd nn (abs mm)
      in (nn / d, mpos / d)

(* find a generator for the multiplicative group mod p
   (where p must be prime for a generator to exist!!) *)

exception No_Generator

let find_generator p =
    let rec period x prod =
        if (prod == 1) then 1
        else 1 + (period x (prod * x mod p))
    in let rec findgen x =
        if (x == 0) then raise No_Generator
        else if ((period x x) == (p - 1)) then x
        else findgen ((x + 1) mod p)
    in findgen 1

(* raise x to a power n modulo p (requires n > 0) (in principle,
   negative powers would be fine, provided that x and p are relatively
   prime...we don't need this functionality, though) *)

exception Negative_Power

let rec pow_mod x n p =
    if (n == 0) then 1
    else if (n < 0) then raise Negative_Power
    else if (n mod 2 == 0) then pow_mod (x * x mod p) (n / 2) p
    else x * (pow_mod x (n - 1) p) mod p
```

```
(****************************************
 * auxiliary functions
 ****************************************)
let rec forall combiner a b f =
    if (a >= b) then []
    else combiner (f a) (forall combiner (a + 1) b f)

let sum_list l = fold_right (+) l 0
let max_list l = fold_right (max) l (-999999)
let min_list l = fold_right (min) l 999999
let count pred = fold_left (fun a elem -> if (pred elem) then 1 + a else a) 0

let remove elem = filter ((!=) elem)
let cons a b = a::b

let null = function [] -> true | _ -> false

(* functional composition *)
let (@@) f g x = f (g x)

(* Hmm... CAML won't allow second-order polymorphism.  Oh well.. *)
(* let forall_flat = forall (@);; *)
let rec forall_flat a b f =
    if (a >= b) then []
    else (f a) @ (forall_flat (a + 1) b f)

let identity x = x

let find_elem p xs = try Some (List.find p xs) with Not_found -> None

(* find x, x >= a, such that (p x) is true *)
let rec suchthat a pred =
  if (pred a) then a else suchthat (a + 1) pred


let selectFirst p xs =
  let rec selectFirst' = function
    | [] -> raise Not_found
    | x::xs when p x -> (x,xs)
    | x::xs -> let (x',xs') = selectFirst' xs in (x',x::xs')
  in try Some(selectFirst' xs) with Not_found -> None

(* used for inserting an element into a sorted list *)
let insertList stop el xs =
  let rec insert' = function
    | [] -> [el]
    | x::xs as xxs -> if stop el x then el::xxs else x::(insert' xs)
  in insert' xs

(* used for inserting an element into a sorted list *)
let insert_list p el xs =
  let rec insert' = function
    | [] -> [el]
    | x::xs as xxs -> if p el x < 0 then el::xxs else x::(insert' xs)
  in insert' xs

let zip xs =
  let rec zip' ls rs = function
    | []    -> (ls,rs)
    | x::xs -> zip' (x::rs) ls xs
  in zip' [] [] xs

let rec intertwine xs zs = match (xs,zs) with
  | ([],zs) -> zs
  | (x::xs,zs) -> x::(intertwine zs xs)
```

```
let (@.) (a,b) (c,d) = (a@c,b@d)

let listAssoc key assoclist =
  try Some (List.assoc key assoclist) with Not_found -> None

let identity x = x
let return   x = x

let listToString toString separator =
  let rec listToString_internal = function
    | [] -> ""
    | [x] -> toString x
    | x::xs -> (toString x) ^ separator ^ (listToString_internal xs) in
  listToString_internal

let stringlistToString = listToString identity

let intToString = string_of_int
let floatToString = string_of_float

let same_length xs zs =
  let rec same_length_internal = function
    | [],[]  -> true
    | [], _  -> false
    | _, []  -> false
    | _::xs,_::zs -> same_length_internal (xs,zs)
  in same_length_internal (xs,zs)

let optionIsSome = function None -> false | Some _ -> true
let optionIsNone = function None -> true  | Some _ -> false
let optionToValue' exn = function None -> raise exn | Some x -> x
let optionToValue v = optionToValue' (Failure "optionToValue") v
let optionToList = function None -> [] | Some a -> [a]

let optionToListAndConcat xs = function
  | None   -> xs
  | Some x -> x::xs

let option_to_boolvaluepair oldvalue = function
  | None          -> (false, oldvalue)
  | Some newvalue -> (true,  newvalue)

let minimize f xs =
  let rec minimize' z z' = function
    | []    -> Some z
    | x::xs ->
        let x' = f x in
          if x' < z' then minimize' x x' xs else minimize' z z' xs
  in match xs with
    | []    -> None
    | [x]   -> Some x
    | x::xs -> minimize' x (f x) xs

let list_removefirst p =
  let rec remove_internal = function
    | [] -> []
    | x::xs -> if p x then xs else x::(remove_internal xs)
  in remove_internal

let cons a b = a::b

let mapOption f = function
  | Some x -> Some (f x)
  | None   -> None
```

```
let get1of3 (x,_,_) = x
let get2of3 (_,x,_) = x
let get3of3 (_,_,x) = x

let get1of4 (x,_,_,_) = x
let get2of4 (_,x,_,_) = x
let get3of4 (_,_,x,_) = x
let get4of4 (_,_,_,x) = x

let get1of5 (x,_,_,_,_) = x
let get2of5 (_,x,_,_,_) = x
let get3of5 (_,_,x,_,_) = x
let get4of5 (_,_,_,x,_) = x
let get5of5 (_,_,_,_,x) = x

let get1of6 (x,_,_,_,_,_) = x
let get2of6 (_,x,_,_,_,_) = x
let get3of6 (_,_,x,_,_,_) = x
let get4of6 (_,_,_,x,_,_) = x
let get5of6 (_,_,_,_,x,_) = x
let get6of6 (_,_,_,_,_,x) = x

let repl1of2 x (_,a) = (x,a)
let repl2of2 x (a,_) = (a,x)

let repl1of3 x (_,a,b) = (x,a,b)
let repl2of3 x (a,_,b) = (a,x,b)
let repl3of3 x (a,b,_) = (a,b,x)

let repl1of4 x (_,a,b,c) = (x,a,b,c)
let repl2of4 x (a,_,b,c) = (a,x,b,c)
let repl3of4 x (a,b,_,c) = (a,b,x,c)
let repl4of4 x (a,b,c,_) = (a,b,c,x)

let repl1of5 x (_,a,b,c,d) = (x,a,b,c,d)
let repl2of5 x (a,_,b,c,d) = (a,x,b,c,d)
let repl3of5 x (a,b,_,c,d) = (a,b,x,c,d)
let repl4of5 x (a,b,c,_,d) = (a,b,c,x,d)
let repl5of5 x (a,b,c,d,_) = (a,b,c,d,x)

let repl1of6 x (_,a,b,c,d,e) = (x,a,b,c,d,e)
let repl2of6 x (a,_,b,c,d,e) = (a,x,b,c,d,e)
let repl3of6 x (a,b,_,c,d,e) = (a,b,x,c,d,e)
let repl4of6 x (a,b,c,_,d,e) = (a,b,c,x,d,e)
let repl5of6 x (a,b,c,d,_,e) = (a,b,c,d,x,e)
let repl6of6 x (a,b,c,d,e,_) = (a,b,c,d,e,x)

let rec fixpoint f a = match f a with
  | (false, b) -> b
  | (true, b') -> fixpoint f b'

let diff a b = filter (fun x -> not (List.mem x b)) a

let addelem a set = if not (List.mem a set) then a :: set else set

let union l =
  let f x = addelem x   (* let is source of polymorphism *)
  in List.fold_right f l

let uniq l =
  List.fold_right (fun a b -> if List.mem a b then b else a :: b) l []

let toNil _  = []
let toNone _ = None
let toZero _ = 0
```

```
let lists_overlap xs zs = List.exists (fun i -> List.mem i xs) zs
let lists_intersection xs zs = List.filter (fun i -> List.mem i xs) zs

let msb x =
  let rec msb_internal msb0 = function
    | 0 -> msb0
    | n -> msb_internal (msb0+1) (n lsr 1) in
  msb_internal (-1) x

let rec list_last = function
  | [] -> failwith "list_last"
  | [x] -> x
  | x::xs -> list_last xs
```

A.2 Input and Output of Prolog Terms

All components of the MAP tool chain operate on the same intermediate representation, which is a subset of ground Prolog terms. Unlike Prolog and Mercury, Ocaml does not natively support Prolog terms. This necessitated the implementation of the following modules.

A.2.1 gil_basics.ml: Basic Definitions

This module includes the data-type definitions of Prolog-style terms and implements common opererations on these terms like testing for groundness, checking for term equality, term comparison, and term unification.

```
open List
open Num

type gilTerm =                       (* GIL TERM *************************)
  | GT_Atom of                       (*   atom (=constant)               *)
        string                       (*      <1> : txt repr. (wo/quotes) *)
  | GT_Any                           (*   anonymous variable             *)
  | GT_Var of                        (*   variable (named)               *)
        string                       (*      <1> : textual repr.         *)
  | GT_Int of                        (*   integer (of arbitrary size)    *)
        num                          (*      <1> : number                *)
  | GT_Float of                      (*   floating-point number          *)
        string ref *                 (*      <1> : cached txt. repr.     *)
                                     (*            "" = not yet cached.  *)
        num                          (*      <2> : number                *)
  | GT_List of                       (*   (proper) list                  *)
        gilTerm list                 (*      <1> : list of subterms      *)
  | GT_Struct of                     (*   structure (w/quoted functor)   *)
        string *                     (*      <1> : functor (wo/quotes)   *)
        gilTerm array                (*      <2> : arguments (fields).   *)

let num_print_precision = ref 25

(* TBD: integrity constraints for this datatype *)
(* NOTE: GT_Float can retain the original textual representation of the
 * input. This can help improve the resemblance between input and output,
 * which increases user-friendliness. *)

module StringMap = Map.Make(struct type t = string let compare = compare end)
module StringSet = Set.Make(struct type t = string let compare = compare end)
```

```
let rec eq_gilTerm u v = match (u,v) with
  | (GT_Atom s1, GT_Atom s2) -> s1 = s2
  | (GT_Var v1, GT_Var v2) -> v1 = v2
  | (GT_Int i, GT_Int j)
  | (GT_Float(_,i), GT_Float(_,j)) -> i =/ j
  | (GT_List xs, GT_List ys) ->
      length xs = length ys && for_all2 eq_gilTerm xs ys
  | (GT_Struct(f1,a1), GT_Struct(f2,a2)) ->
      let l = Array.length a1
      in Array.length a2 = l && f1 = f2 && eq_gilTermArray' (l - 1) a1 a2
  | _ -> false
and eq_gilTermArray' i a1 a2 =
  i < 0 || (eq_gilTerm a1.(i) a2.(i) && eq_gilTermArray' (i - 1) a1 a2)

let rec compare_gilTerm u v = match (u,v) with
  | (GT_Int i, GT_Int j)
  | (GT_Float(_,i), GT_Float(_,j)) -> compare_num i j
  | (GT_List xs, GT_List ys) -> compare_gilTerms xs ys
  | (GT_Struct(f1,a1), GT_Struct(f2,a2)) ->
      let c = compare f1 f2 in
      if c <> 0 then c
      else
        let l1 = Array.length a1
        and l2 = Array.length a2
        in if l1 <> l2 then compare l1 l2
           else compare_gilTermArray' 0 (l1 - 1) a1 a2
  | (u,v) -> compare u v
and compare_gilTerms xs ys = match (xs,ys) with
  | ([], [])       ->  0
  | (_::_, [])     ->  1
  | ([], _::_)     -> -1
  | (x::xs, y::ys) ->
      let c = compare_gilTerm x y in
        if c <> 0 then c else compare_gilTerms xs ys
and compare_gilTermArray' i n a1 a2 =
  if i > n then 0
  else
    let c = compare_gilTerm a1.(i) a2.(i) in
      if c <> 0 then c else compare_gilTermArray' (i + 1) n a1 a2


module GilTermSet = Set.Make(struct
                              type t = gilTerm
                              let compare = compare_gilTerm
                            end)

let gilTermSet_mem' set el = GilTermSet.mem el set

module GilTermMap = Map.Make(struct
                              type t = gilTerm
                              let compare = compare_gilTerm
                            end)

let gilTermMap_find k m = try Some(GilTermMap.find k m) with Not_found -> None
let gilTermMap_findE k m = try GilTermMap.find k m with Not_found -> []
let gilTermMap_addE k v m  = GilTermMap.add k (v::(gilTermMap_findE k m)) m
let gilTermMap_addE' v k m = GilTermMap.add k (v::(gilTermMap_findE k m)) m

let rec subst_in_gilTerm map t =
  try GilTermMap.find t map with Not_found -> subst_in_gilTerm' map t
and subst_in_gilTerm' map t = match t with
  | GT_Any           -> t
  | GT_Atom _        -> t
  | GT_Var _         -> t
  | GT_Int _         -> t
```

```
  | GT_Float(_,_)     -> t
  | GT_List xs        -> GT_List (List.map (subst_in_gilTerm map) xs)
  | GT_Struct(f,args) -> GT_Struct(f, Array.map (subst_in_gilTerm map) args)

(* returns true if term u is a subterm of v or u=v. *)
let rec occurs_in_gilTerm u v = eq_gilTerm u v || occurs_in_gilTerm' u v
and occurs_in_gilTerm' u = function
  | GT_Any            -> false
  | GT_Atom _         -> false
  | GT_Var _          -> false
  | GT_Int _          -> false
  | GT_Float(_,_)     -> false
  | GT_List xs        -> exists (occurs_in_gilTerm u) xs
  | GT_Struct(_,args) -> occurs_in_gilTermArray' u (Array.length args - 1) args
and occurs_in_gilTermArray' u i xs =
  i >= 0 && (occurs_in_gilTerm u xs.(i) || occurs_in_gilTermArray' u (i-1) xs)

let rec gilTerm_is_ground = function
  | GT_Any            -> false
  | GT_Atom _         -> true
  | GT_Var _          -> false
  | GT_Int _          -> true
  | GT_Float _        -> true
  | GT_List xs        -> for_all gilTerm_is_ground xs
  | GT_Struct(_,args) -> gilTermArray_is_ground' (Array.length args - 1) args
and gilTermArray_is_ground' i args =
  i < 0 || (gilTerm_is_ground args.(i) && gilTermArray_is_ground' (i - 1) args)

exception Not_unifyable

(* note: In most cases, dereferencing is _not_ necessary.
 * Try to make the default path fast. *)
let rec unify_gilTerms' map u v = match (u, v) with
  | (GT_Atom s1, GT_Atom s2) when s1 = s2 -> map
  | (GT_Int n1, GT_Int n2) when n1 =/ n2 -> map
  | (GT_Float(_,i), GT_Float(_,j)) when i =/ j -> map
  | (GT_List xs, GT_List ys) -> unify_gilTermsList map (xs,ys)
  | (GT_Struct(f1,a1), GT_Struct(f2,a2))
    when f1 = f2 && Array.length a1 = Array.length a2 ->
      unify_gilTermsArray (Array.length a1 - 1) map a1 a2
  | (GT_Any, _)
  | (_, GT_Any) -> map
  | (GT_Var v1, GT_Var v2) when v1 = v2 -> map
  | (GT_Var _ as v, t)
  | (t, (GT_Var _ as v)) -> unify_var map t v   (* maybe deref *)
  | _ -> raise Not_unifyable
and unify_var map t v =
  try (unify_gilTerms' map t (GilTermMap.find v map))
  with Not_found ->
    (if occurs_in_gilTerm' v t then raise Not_unifyable
     else GilTermMap.add v t map)
and unify_gilTermsList map = function
  | ([], []) -> map
  | (x::xs, y::ys) -> unify_gilTermsList (unify_gilTerms' map x y) (xs,ys)
  | _ -> raise Not_unifyable
and unify_gilTermsArray i map a1 a2 =
  if i < 0 then map
  else unify_gilTermsArray (i - 1) (unify_gilTerms' map a1.(i) a2.(i)) a1 a2

let unifyable_gilTerms u v =
  try ignore (unify_gilTerms' GilTermMap.empty u v); true
  with Not_unifyable -> false

let unify_gilTerms u v =
  try Some (unify_gilTerms' GilTermMap.empty u v) with Not_unifyable -> None
```

```
let memberChk_gilTermList  t xs = exists (unifyable_gilTerms t) xs
let nonMember_gilTermList  t xs = not (exists (unifyable_gilTerms t) xs)
let nonMember_gilTermList' xs t = not (exists (unifyable_gilTerms t) xs)
let mem_gilTermList        t xs = exists (eq_gilTerm t) xs

let assoc_gilTermList key xs =
  snd (find (fun (pat,_) -> unifyable_gilTerms pat key) xs)

let assoc_gilTermList__pred key pred xs =
  snd (find (fun (pat,stuff) ->
              unifyable_gilTerms pat key &&
              pred key stuff) xs)

let assoc_gilTermList' key default xs =
  try assoc_gilTermList key xs with Not_found -> default

let value_of_option = function
  | Some s -> s
  | None -> failwith "value_of_option(None)"

let gilTermLists_overlap (xs,zs) = exists (fun x -> mem_gilTermList x zs) xs

let uniq_gilTermList xs =
  fold_right (fun x xs -> if mem_gilTermList x xs then xs else x::xs) xs []

let diff_gilTermList xs ys =
  fold_right (fun x xs -> if mem_gilTermList x ys then xs else x::xs) xs []

let gilTerm_of_int i = GT_Int (Num.num_of_int i)

let functor_is_in_map s_map = function
  | GT_Atom f | GT_Struct(f,_) -> StringMap.mem f s_map
  | _ -> false
```

A.2.2 `gil_util.ml`: Basic Operations

This module provides low-level routines on Prolog-style terms.

```
open Array
open List
open String
open Num
open Gil_basics

let range_includes range c = match range with
  | `T1(pt) -> c = pt
  | `T2(fromPt,toPt) -> c >= fromPt && c <= toPt

let lowercase_ranges = [`T2('a','z')]
and uppercase_ranges = [`T2('A','Z'); `T1('_')]
and identchar_ranges = [`T2('A','Z'); `T2('a','z'); `T1('_'); `T2('0','9')]

let lut_of_ranges rs =  (* slow, used once during program initialization *)
  init 256 (fun i -> exists (fun r -> range_includes r (char_of_int i)) rs)

let lowercase_lut = lut_of_ranges lowercase_ranges
and uppercase_lut = lut_of_ranges uppercase_ranges
and identchar_lut = lut_of_ranges identchar_ranges

let atom_must_be_quoted s =
  let rec loop i =
    i > 0 && (not identchar_lut.(int_of_char s.[i]) || loop (i - 1))
  in s = "" || not (lowercase_lut.(int_of_char s.[0])) || loop (length s - 1)
```

```
(* strip away <nL> on the left and <nR> chars on the right side of <s>. *)
(* precondition: nL and nR are non-negative integers *)
let strip nL nR s = if nL + nR = 0 then s else sub s nL (length s - nL - nR)

let rindex2 str c1 c2 = try rindex str c1 with Not_found -> rindex str c2

let num_of_floatstring str =
  let idx_dot = index str '.' in
  let len = length str - 1
  and ten = num_of_int 10 in
  let str' = create len in
  begin
    blit str 0 str' 0 idx_dot;
    blit str (idx_dot + 1) str' idx_dot (len - idx_dot);
    try
      let idx_e = rindex2 str' 'e' 'E' in
      let ab = num_of_string (strip 0 (len - idx_e) str')
      and e  = num_of_string (strip (idx_e + 1) 0 str') in
        ab // (ten **/ (num_of_int (idx_e - idx_dot) -/ e))
    with Not_found ->
      num_of_string str' // (ten **/ num_of_int (len - idx_dot))
  end

let number_of_trailing_chars s c =
  let rec loop i =
    if i < 0 then (i, `EndOfString)
    else let c' = s.[i] in if c' = c then loop (i-1) else (i, `DifChar(c')) in
  let l_mone = length s - 1 in let (p,why) = loop l_mone in (l_mone - p, why)

let floatstring_of_num n =
  let s = approx_num_exp !num_print_precision n in
  let l = length s in
  let (sR, eOpt) =
    if l > 2 && s.[l - 1] = '0' &&      (* min. len. = 2, last char = '0' *)
       (let c = s.[l - 2] in c = 'e' || c = 'E') then (2, None)
    else let i = rindex2 s 'e' 'E' in (l - i, Some (strip i 0 s)) in
  let sL = if s.[0] = '+' then 1 else 0 in
  let mantissa = strip sL sR s in
  let trailing_zeroes = match number_of_trailing_chars mantissa '0' with
    | (tz, `EndOfString) -> tz
    | (tz, `DifChar('.')) when tz > 0 -> tz-1
    | (tz, `DifChar(_)) -> tz in
  let mantissa = strip 0 trailing_zeroes mantissa in
    match eOpt with
      | Some exponent -> mantissa ^ exponent
      | None -> mantissa

(* "float atom" = arbitrary precision floating point number. Prolog format. *)
let floatAtom_of_num n = GT_Atom (floatstring_of_num n)

let num_of_floatAtom = function
  | GT_Atom s -> num_of_floatstring s
  | _ -> failwith "num_of_floatAtom"

let string_of_numPair (n,m) =
  "(" ^ (floatstring_of_num n) ^ "," ^ (floatstring_of_num m) ^ ")"
```

A.2.3 gil_IO.ml: Input/Output Functions

This module includes functions for handling input and output of Prolog-style terms.

```
open Buffer
```

```
open String
open Gil_basics
open Gil_util

let add_unquoted b s = add_string b s
let add_quoted   b s = add_char b '\''; add_string b s; add_char b '\''

let cachedstring_of_num tRep n =              (* uses 'tRep' as a cache *)
  let s = !tRep
  in if s = "" then let s = floatstring_of_num n in tRep := s; s else s

let rec add_list b s_term c_sep = function
  | [] -> add_string b s_term
  | x::xs -> add_char b c_sep; add_term b x; add_list b "]" ',' xs
and add_term b = function
  | GT_Atom s when atom_must_be_quoted s -> add_quoted b s
  | GT_Atom s -> add_unquoted b s
  | GT_Var s -> add_unquoted b s
  | GT_Any -> add_char b '_'
  | GT_Int n -> add_unquoted b (Num.string_of_num n)
  | GT_Float(tRep, n) -> add_unquoted b (cachedstring_of_num tRep n)
  | GT_List xs -> add_list b "[]" '[' xs
  | GT_Struct(f, args) ->
      let l = Array.length args in
        assert (l > 0);
        if atom_must_be_quoted f then add_quoted b f else add_unquoted b f;
        add_char b '(';
          add_term b args.(0);
          for i = 1 to l - 1 do add_char b ','; add_term b args.(i) done;
        add_char b ')'


(* EXPORTED FUNCTIONS ******************************************************)

(* Limitation: at the moment 'read_file' only works with regular files. *)
let read_file_as_string f =
  let in' = open_in f
  in let length = in_channel_length in'
  in let s = String.create length
  in really_input in' s 0 length; close_in in'; s

let write_buffer_as_file f b =
  let oc = open_out f in output_buffer oc b; close_out oc

let write_string_as_file f s =
  let oc = open_out f in output_string oc s; close_out oc

let append_mode = [Open_append; Open_creat; Open_binary; Open_nonblock]

let append_string_to_file f s =
  let oc = open_out_gen append_mode 0o644 f
  in output_string oc s; close_out oc

(* output (unparsing) *)
let add_gilTerm buf t = add_term buf t
let add_gilTermDotNl buf t = add_term buf t; add_string buf ".\n"

let string_of_gilTerm t = let b = Buffer.create 16 in add_term b t; contents b

let print_gilTerm eol t = print_string (string_of_gilTerm t); print_string eol

let print_gilTermDotNl t = print_gilTerm ".\n" t
let print_gilTermDotNlFlush t = print_gilTerm ".\n" t; flush stdout

(* input (parsing) *)
```

```
(* XXX 'lexbuf' should be created from a string (bug?) *)
let gilTermOption_from_lexbuf l = Gil_parser.p_OneTerm Gil_lexer.scan l

let gilTerms_of_string s =
  Gil_parser.p_AllTerms Gil_lexer.scan (Lexing.from_string s)

let gilTerms_of_file f = gilTerms_of_string (read_file_as_string f)

let exactlyOneTerm_of_string str = match gilTerms_of_string str with
  | [ t ] when gilTerm_is_ground t -> t
  | _ -> failwith "exactlyOneTerm_of_string"

let exactlyOneTerm_of_file f = exactlyOneTerm_of_string (read_file_as_string f)

let io_behaviour = ref (None : (gilTerm * gilTerm * gilTerm * gilTerm) option)

let invoke_gilFilter read_term f =
  let (termIn,input,output,termOut) = value_of_option !io_behaviour
  in let substis = unify_gilTerms' GilTermMap.empty termIn read_term
  in print_gilTermDotNl
       (subst_in_gilTerm
          (unify_gilTerms'
             substis (f (subst_in_gilTerm substis input)) output) termOut)

let verbose = ref false
let info s = if !verbose then prerr_endline (Sys.argv.(0) ^ ": " ^ s)

(* TBD: support increasing indentation (enhances readability) *)
let debug_output = ref true
let outputDebugString s = if !debug_output then print_endline ("% " ^ s)
let outputDebugString' s t =
  if !debug_output then print_endline ("% " ^ s ^ (string_of_gilTerm t))

let process_command fn = function
  | GT_Struct("debugOutput", [| GT_Atom b |]) ->
      debug_output := bool_of_string b
  | GT_Struct("verbose", [| GT_Atom b |]) ->
      verbose := bool_of_string b
  | GT_Struct("io_behaviour", [| in'; i; o; out' |]) ->
      io_behaviour := Some(in',i,o,out')
  | GT_Struct(f, args) ->
      fn (f, args)
  | _ -> ()

let usage2 = "Usage: " ^ Sys.argv.(0) ^ " <cmd-file> <input-file>"

let assert_and_get_two_args () =
  if Array.length Sys.argv <> 3 then failwith usage2
  else (Sys.argv.(1), Sys.argv.(2))

let gilFilter2Args process_argument f (cmd_file, data_file) =
  List.iter (process_command process_argument) (gilTerms_of_file cmd_file);
  invoke_gilFilter (exactlyOneTerm_of_file data_file) f

let failwith s = failwith (Sys.argv.(0) ^ ": " ^ s)

exception IllegalTerm of string * string

let illTerm where t = raise (IllegalTerm(where, string_of_gilTerm t))
let illTerm_s where s = raise (IllegalTerm(where, s))

let gilTermMap_find' id k m =
  try GilTermMap.find k m
  with Not_found -> illTerm ("GilTermMap.find(" ^ id ^ ")") k
```

```
let assoc_gilTermList__exn k xs =
  try assoc_gilTermList k xs
  with Not_found -> illTerm "assoc_gilTermList" k

let assoc_gilTermList__pred__exn k pred xs =
  try assoc_gilTermList__pred k pred xs
  with Not_found -> illTerm "assoc_gilTermList__pred" k

let functor_of_gilTerm = function
  | GT_Struct(f,_) -> f
  | GT_Atom a -> a
  | t -> illTerm "functor_of_gilTermStruct" t

let list_of_gilTerm = function
  | GT_List xs -> xs
  | t -> illTerm "list_of_gilTerm" t

let int_of_gilTerm = function
  | GT_Int n -> Num.int_of_num n
  | t -> illTerm "int_of_gilTerm" t
```

A.2.4 `gil_lexer.mll`: Tokenizing

The lexical analysis of input terms uses `ocamllex`, a lexical analyzer generator. Out of the following input, `ocamllex` produces a lexical analyzer for Prolog-style terms.

```
{
open Lexing
open Gil_basics
open Gil_util
open Gil_parser

exception IllegalChar_atPos of string * int

let floatToken_of_string s = T_Float(s, num_of_floatstring s)
let intToken_of_string s   = T_Int(Num.num_of_string s)

let strip_lexeme s0 s1 lexbuf =
  let str = lexeme lexbuf
  in String.sub str s0 (String.length str - s0 - s1)
}

let lowercase = ['a'-'z']
let uppercase = ['A'-'Z' '_']
let identchar = ['A'-'Z' 'a'-'z' '_' '0'-'9']
let digit = ['0'-'9']
let sign = ['+' '-']
let e = ['e' 'E']

rule scan = parse
  | '_'
        { Tunderline }
  | uppercase+ identchar*
        { T_Var(lexeme lexbuf) }

  | lowercase identchar* '('
        { T_Functor(strip_lexeme 0 1 lexbuf) }
  | lowercase identchar*
        { T_Atom(lexeme lexbuf) }

  |  "'" [^ '\'']* "'("
```

```
      { T_Functor(strip_lexeme 1 2 lexbuf) }
  |  "'" [^ '\'']* "'"
      { T_Atom(strip_lexeme 1 1 lexbuf) }

  |  sign? digit+ '.' digit+ (e sign? digit+)?
      { floatToken_of_string (lexeme lexbuf) }
  |  sign? digit+
      { intToken_of_string (lexeme lexbuf) }

  | '['                        { Tlbracket }
  | ']'                        { Trbracket }
  | ')'                        { Trparen }
  | ','                        { Tc }
  | '.'                        { Tperiod }
  | eof                        { Teof }

  | [' ' '\t' '\n' '\r']+
  | '%' [^'\n']*               { scan lexbuf }
  | _   { raise (IllegalChar_atPos(lexeme lexbuf, lexeme_start lexbuf)) }
```

A.2.5 gil_parser.mly: Parsing

Syntax analysis of input terms uses the parser generator `ocamlyacc`. Out of the following input, `ocamlyacc` produces a parser for Prolog-style terms.

```
%{
open Gil_basics
%}

%token Tunderline Tlbracket Trbracket Trparen Tc Tperiod Teof

%token <string> T_Var
%token <Num.num> T_Int
%token <string * Num.num> T_Float
%token <string> T_Atom
%token <string> T_Functor

%start p_AllTerms p_OneTerm

%type <Gil_basics.gilTerm list>   p_AllTerms
%type <Gil_basics.gilTerm option> p_OneTerm

%%

p_AllTerms:
  | p_Term Tperiod p_AllTerms   { $1::$3 }
  | p_Term Teof                 { [$1] }
  | Teof                        { [] }
;

p_OneTerm:
  | p_Term Tperiod              { Some $1 }
  | p_Term Teof                 { Some $1 }
  | Teof                        { None }
;


p_Term:
  | Tunderline                  { GT_Any }
  | T_Var                       { GT_Var($1) }
  | T_Atom                      { GT_Atom($1) }
  | T_Int                       { GT_Int($1) }
  | T_Float                     { GT_Float(ref (fst $1), snd $1) }
```

```
  | Tlbracket Trbracket         { GT_List [] }
  | Tlbracket p_List            { GT_List $2 }
  | T_Functor p_StructArgs      { GT_Struct($1, Array.of_list $2) }
;

p_StructArgs:
  | p_Term Trparen              { [$1] }
  | p_Term Tc p_StructArgs      { $1::$3 }
;

p_List:
  | p_Term Trbracket            { [$1] }
  | p_Term Tc p_List            { $1::$3 }
;

%%
```

A.3 Instruction Definitions

As all modules operate on instruction DAGs, they also share the definition of instructions and basic operations on them.

A.3.1 instr_def.ml: Instr Definition

```
open Basics
open Gil_basics
open Num

type instr =
  | I of                                (* ANNOTATED INSTRUCTION ************)
        gilTerm *                       (*   operation                      *)
        gilTerm list *                  (*   list of source operands        *)
        gilTerm list *                  (*   list of destination operands   *)
        gilTerm list                    (*   annotations                    *)

type cDecl =                            (* CONSTANT DECLARATION *************)
  | F_Const of                          (*   scalar FP constant             *)
        string * precision * num
  | F2_Const of                         (*   2-way SIMD FP constant         *)
        string * precision * num * num
  | F4_Const of                         (*   4-way SIMD FP constant         *)
        string * num * num * num * num
  | I4_Const of                         (*   4-way SIMD integer constant    *)
        string * num * num * num * num

let srcs_of_instr (I(_,srcs,_,_)) = srcs
let dsts_of_instr (I(_,_,dsts,_)) = dsts
let annos_of_instr (I(_,_,_,annos)) = annos

let id_of_cDecl = function      (* get identifier of some const-declaration *)
  | F_Const(id,_,_)       -> id
  | F2_Const(id,_,_,_)    -> id
  | F4_Const(id,_,_,_,_) -> id
  | I4_Const(id,_,_,_,_) -> id
```

A.3.2 instr_IO.ml: Input/Output of Instructions

```
open Basics
open Gil_basics
```

```
open Gil_util
open Gil_IO
open Instr_def

let instr_of_gilTerm = function
  | GT_Struct("i", [| op; GT_List srcs; GT_List dsts; GT_List annos |]) ->
      I(op,srcs,dsts,annos)
  | t -> illTerm "instr_of_gilTerm" t

let gilTerm_of_instr (I(op,srcs,dsts,annos)) =
  GT_Struct("i", [| op; GT_List srcs; GT_List dsts; GT_List annos |])

let cDecl_of_gilTerm = function
  | GT_Struct("f_const", [| GT_Atom id; GT_Atom p; c |]) ->
      F_Const(id, precision_of_string p, num_of_floatAtom c)
  | GT_Struct("f2_const", [| GT_Atom id; GT_Atom p; cL; cH |]) ->
      F2_Const(id, precision_of_string p,
               num_of_floatAtom cL, num_of_floatAtom cH)
  | GT_Struct("f4_const", [| GT_Atom id; c1; c2; c3; c4 |]) ->
      F4_Const(id, num_of_floatAtom c1, num_of_floatAtom c2,
                   num_of_floatAtom c3, num_of_floatAtom c4)
  | GT_Struct("i4_const", [| GT_Atom id;
                             GT_Int i1; GT_Int i2; GT_Int i3; GT_Int i4 |]) ->
      I4_Const(id, i1, i2, i3, i4)
  | t -> illTerm "cDecl_of_gilTerm" t

let gilTerm_of_cDecl = function
  | F_Const(id, p, c) ->
      GT_Struct("f_const",
                [| GT_Atom id; GT_Atom (string_of_precision p);
                   floatAtom_of_num c |])
  | F2_Const(id, p, cL, cH) ->
      GT_Struct("f2_const",
                [| GT_Atom id; GT_Atom (string_of_precision p);
                   floatAtom_of_num cL; floatAtom_of_num cH |])
  | F4_Const(id, c1, c2, c3, c4) ->
      GT_Struct("f4_const",
                [| GT_Atom id;
                   floatAtom_of_num c1; floatAtom_of_num c2;
                   floatAtom_of_num c3; floatAtom_of_num c4 |])
  | I4_Const(id, i1, i2, i3, i4) ->
      GT_Struct("i4_const",
                [| GT_Atom id; GT_Int i1; GT_Int i2; GT_Int i3; GT_Int i4 |])
```

A.4 List-based Scheduling

The list-based instruction scheduler LL1 comprises five program modules shown in the following subsections.

A.4.1 `inssched_def.ml`: Basic Definitions

Instruction scheduling builds upon the following definitions of dependencies, dependency types, and earliest starting times.

```
open Id

type depType =                        (* DEPENDENCY TYPE *****************)
  | RAW of                            (*   Read After Write              *)
      int                             (*     <1> : instruction latency   *)
```

```
  | WAR                            (*   Write After Read             *)
  | WAW                            (*   Write After Write            *)

type dependency =                  (* INSTRUCTION DEPENDENCY ***********)
  depType *                        (*   <1> : dependency type        *)
  Id.t *                           (*   <2> : id of succeeding instr.  *)
  Id.t *                           (*   <3> : id of preceeding instr.  *)
  int                              (*   <4> : crit. path contribution  *)

type bwdepscnt_est =               (* WAITING INSTRUCTION **************)
                                   (* an instr waiting on other instrs *)
  int *                            (*   <1> : bw. dependency count     *)
                                   (*       = # of instrs to be issued *)
                                   (*         before this instr        *)
  int                              (*   <2> : earliest starting time   *)

type est_id =                      (* PRE-READY INSTRUCTION ************)
  | EST_Id of                      (* an instr that can be issued once *)
                                   (* its inputs have been written     *)
        int *                      (*   <1> : earliest starting time   *)
        Id.t                       (*   <2> : instruction id           *)

type cp_id =                       (* READY INSTRUCTION ****************)
  | CP_Id of                       (* an instr ready for issue         *)
        int *                      (*   <1> : critical path length     *)
        Id.t                       (*   <2> : instruction id           *)

(* set of pre-ready instructions *)
module ESTIdSet = Set.Make(struct type t = est_id let compare = compare end)

let depToSucc (_,succ,pred,_) = succ

(* map a dependency to the latency from the preceeding to the succeeding
 * instruction. if the dependency is of type 'read after write',
 * return the latency of the producer. otherwise return 0. *)
let depToLatency (t,_,_,_) = match t with
  | RAW n -> n
  | WAW   -> 0
  | WAR   -> 0

(* map a dependency to its effective contribution to the critical path length.
 * the effective contribution may be different from its latency. *)
let depToCPLen (_,_,_,x) = x

(* compare two ready instructions (used for sorting in descending order) *)
let cmp_cpid (CP_Id(x,_)) (CP_Id(y,_)) = compare y x
```

A.4.2 `inssched_basics.ml`: Basic Operations

The basic operations of the instruction scheduler include calculating forward and backward dependencies and critical path lengths, all implemented in the following module.

```
open Id
open Util
open List
open Gil_basics
open Gil_IO
open Instr_def
open Instr_IO
open Inssched_def
```

```
(* look up the effective latency for the instruction pair ('s','d'), for
 * the resource of type 'res' in the  list of scheduling clauses
 * 'scheduling'. if that pair can not be found, return 'default_latency'. *)
let get_effective_latency scheduling s d res default_latency =
  try
    let t = (GT_Struct("s_d", [| s; d |]))
    in assoc_gilTermList res (get2of5 (assoc_gilTermList t scheduling))
  with Not_found -> default_latency

(* lookup a scheduling clause in 'scheduling' matching 'instr'. *)
let lookup_instr instr scheduling =
  let test i (annoReq,_,_,_,_) =
    let annos = annos_of_instr (instr_of_gilTerm i)
    in for_all (fun req -> memberChk_gilTermList req annos) annoReq
  in assoc_gilTermList__pred__exn instr test scheduling

let calc_forward_dependencies favor favorReleasers scheduling instrs =
  let calc_fw_deps (idx,orig_order,idmap,deps,lastreads,lastwrite) instr =
    let id = Id.makeNew () in
    let addDepsR r_res = match gilTermMap_find r_res lastwrite with  (* RAW *)
      | None -> return
      | Some (w_id,w_latency) ->
          let producer = IdMap.find w_id idmap
          in let eLat  = get_effective_latency
                           scheduling producer instr r_res w_latency
          in idmap_addE w_id ((RAW eLat), id, w_id, eLat + 1)
    and addDepsW (w_res,_) deps =                             (* WAW and WAR *)
      fold_right (fun r_id -> idmap_addE r_id (WAR, id, r_id, 1))
                 (gilTermMap_findE w_res lastreads)
                 (match gilTermMap_find w_res lastwrite with
                    | Some (w_id,_) -> idmap_addE w_id (WAW, id, w_id, 1) deps
                    | None -> deps)
    in let (I(_,rs,ws,_)) = instr_of_gilTerm instr
    in let (_,lats,xsrcs,xdsts,_) = lookup_instr instr scheduling
    in let rs = xsrcs @ rs
    in let ws = map (fun d -> (d, assoc_gilTermList' d 0 lats)) (xdsts @ ws)
    in let lastreads' = fold_right (gilTermMap_addE' id) rs lastreads
    in (idx + 1,
        IdMap.add id idx orig_order,
        IdMap.add id instr idmap,
        fold_right addDepsW ws
                   (fold_right addDepsR rs (IdMap.add id [] deps)),
        fold_right (fun (res,_) -> GilTermMap.add res []) ws lastreads',
        fold_right
          (fun (res,lat) -> GilTermMap.add res (id,lat)) ws lastwrite) in
  let (idm0,gtm0) = (IdMap.empty, GilTermMap.empty)
  in let s0 = (0, idm0, idm0, idm0, gtm0, gtm0)
  in let (_,orig_order,idmap,fw_deps,_,_) = fold_left calc_fw_deps s0 instrs
  in (orig_order, (fun id -> IdMap.find id idmap), fw_deps)

(* given the forward dependencies of some DAG,
 * return the corresponding backward dependencies. *)
let calc_backward_dependencies_count fw_deps =
  IdMap.fold (fun _ -> fold_right (fun dep -> idmap_inc (depToSucc dep)))
             fw_deps (IdMap.map toZero fw_deps)

(* map the forward and backward-dependencies to critical-path lengths *)
let calc_critical_path_lengths fw_deps bw_deps_cnt =
  let fold_start_instrs id cnt = if cnt = 0 then cons id else return in
  let startinstrs = IdMap.fold fold_start_instrs bw_deps_cnt [] in
  let rec calc_cplen' id cplen =
    if idmap_exists id cplen then cplen else calc_cplen'' id cplen
  and calc_cplen'' id cplen = match IdMap.find id fw_deps with
    | [] -> IdMap.add id 0 cplen
    | deps ->
```

```
      let cplen' = fold_right calc_cplen' (map depToSucc deps) cplen in
      let depToCPLen' d = depToCPLen d + IdMap.find (depToSucc d) cplen'
      in IdMap.add id (max_list (map depToCPLen' deps)) cplen' in
  let cplens = fold_right calc_cplen' startinstrs IdMap.empty in
    fun id -> IdMap.find id cplens
```

A.4.3 `execution_modelling.ml`: Execution Modeling

The parametrizable simulation of a super-scalar in-order approximation of some target architecture is shown in the following module.

```
open Id
open Util
open List
open Gil_basics
open Gil_IO
open Inssched_def
open Inssched_basics

type requiredResources =              (* REQUIRED RESOURCES ***************)
  | True                              (*   None (succeeds always)        *)
  | False                             (*   (fails always)                *)
  | Unit_blocked of gilTerm * int     (*   Unit <1> for <2> cycles       *)
  | Or of requiredResources list      (*   One of the resources in <1>   *)
  | And of requiredResources list     (*   All of the resources in <1>   *)

let rec reqRes_of_gilTerm = function
  | GT_Atom "true"                      -> True
  | GT_Atom "false"                     -> False
  | GT_Struct("and", [| GT_List xs |])    -> And (map reqRes_of_gilTerm xs)
  | GT_Struct("or", [| GT_List xs |])     -> Or (map reqRes_of_gilTerm xs)
  | GT_Struct("unit_blocked", [| u; n |]) -> Unit_blocked(u, int_of_gilTerm n)
  | t -> illTerm "reqRes_of_gilTerm" t

type allocationStatus =               (* UNIT ALLOCATION STATUS ***********)
  | Free of gilTerm                   (*   <1> is available for use      *)
  | Busy_until of gilTerm * int       (*   <1> is blocked until t >= <2>  *)

let initialStatus_of_resources xs = map (fun x -> Free x) xs

let maybe_release_resource t = function
  | Busy_until(r,until) when t > until -> Free r
  | x -> x

let suitable_free_resource unit = function
  | Busy_until _ -> false
  | Free r -> unifyable_gilTerms r unit

open NonDetMonad

let rec mayIssueInSameCycleM'' = function
  | True   -> unitM ()
  | False  -> failM
  | And xs -> iterM mayIssueInSameCycleM'' xs
  | Or xs  -> memberM xs >>= mayIssueInSameCycleM''
  | Unit_blocked(u,n) ->
      fetchStateM >>= fun (t,res) ->
        selectFirstM (suitable_free_resource u) res >>= function
          | (Free r,other) -> storeStateM (t, (Busy_until(r,t+n-1))::other)
          | _ -> failwith "mayIssueInSameCycleM"

let mayIssueInSameCycleM' scheduling (_,_,instr) =
  try  mayIssueInSameCycleM'' (get5of5 (lookup_instr instr scheduling))
```

```
  with _ -> unitM ()                  (* ignore unknown instructions *)

let mayIssueInSameCycleM scheduling idmap instrs ready =
  iterM (mayIssueInSameCycleM' scheduling) instrs >>= fun _ ->
    selectM ready >>= fun ((CP_Id(cplen,id),rs) as p) ->
      mayIssueInSameCycleM' scheduling (-1,-1,idmap id) >>= fun _ ->
        unitM p
```

A.4.4 `inssched.ml`: List Scheduling Algorithm

The actual list-based scheduling algorithm is implemented in the following module.

```
open Id
open Util
open List
open Basics
open Gil_basics
open Gil_IO
open Instr_def
open Instr_IO
open Inssched_def
open Inssched_basics
open Execution_modelling
open NonDetMonad

type picking_heuristics = OrigOrder | CritPath

(* adapt earliest starting time. decrement the number of bw deps of an
 * instruction. if it is zero, move the instr from waiting to preready. *)
let adapt_est t dep (preready,waiting) =
  let id = depToSucc dep
  in let (r_bwDepsLen,r_est) = IdMap.find id waiting
  in let r_est' = max (t + (depToLatency dep)) r_est
  in if r_bwDepsLen > 1 then         (* # of other bw_deps of the successor *)
       (preready, IdMap.add id (r_bwDepsLen-1, r_est') waiting)
     else
       (ESTIdSet.add (EST_Id(r_est',id)) preready, IdMap.remove id waiting)

let flatinssched maxPerCycle favor favorReleasers resources scheduling
                 idmap fw_deps cplens =
  (* move instructions from 'preready' to 'ready' *)
  let rec add_new_ready t ((ready, preready) as ready_preready) =
    if ESTIdSet.is_empty preready then ready_preready
    else
      let EST_Id(est,id) as x = ESTIdSet.min_elt preready
      in if t < est then ready_preready
         else
           let instr = idmap id
           in let (I(_,rs,ws,_)) = instr_of_gilTerm instr
           in let rws = lists_intersection rs ws
           in let rws' =
                List.map (fun x -> assoc_gilTermList' x 0 favorReleasers) rws
           in let i_favor = assoc_gilTermList' instr 0 favor
           in let i_favorReleasers = sum_list rws'
           in let cpl' = cplens id + i_favor + i_favorReleasers
           in let ready' = insert_list cmp_cpid (CP_Id(cpl',id)) ready
           in add_new_ready t (ready', ESTIdSet.remove x preready)
  and loop avail_res t nr_freeslots instrs_in_t instrs
           ((ready, preready) as ready_preready) waiting =
    let complete () = rev (instrs_in_t @ instrs)
    and start_new_cycle () =
      let (_,(_,avail_res')) =
```

```
        value_of_option (runM (iterM (mayIssueInSameCycleM' scheduling))
                              instrs_in_t (t,avail_res))
      in let t' = t+1
      in loop (map (maybe_release_resource t') avail_res')
              t' maxPerCycle [] (instrs_in_t @ instrs)
              (add_new_ready t' ready_preready) waiting in
    match (ready,preready,waiting) with
      | ([],p,w) when ESTIdSet.is_empty p && w = IdMap.empty -> complete ()
      | ([],p,_) when not (ESTIdSet.is_empty p) -> start_new_cycle ()
      | ([],_,_) -> failwith "flatinssched"
      | (_::_,_,_) when nr_freeslots = 0 -> start_new_cycle ()
      | (_::_ as ready,_,_) ->
          (* select an instr with the longest critical path that may be issued
           * simultaneously with the ones already selected earlier. *)
          match runM (mayIssueInSameCycleM scheduling idmap instrs_in_t)
                     ready (t,avail_res) with
            | None -> start_new_cycle ()
            | Some((CP_Id(cplen,id),ready'),_) ->
                let (preready',waiting') =
                        fold_right (adapt_est t) (IdMap.find id fw_deps)
                                   (preready,waiting)
                in let instrs_in_t' = (t,cplen,idmap id)::instrs_in_t
                in loop avail_res t (nr_freeslots-1) instrs_in_t' instrs
                        (add_new_ready t (ready',preready')) waiting'
 in loop (initialStatus_of_resources resources) 0 maxPerCycle [] []

let instrsToInstructionscheduled
        pickHeur
        maxPerCycle favor favorReleasers resources scheduling instrs =
  let (orig_order,idmap,fw_deps) =
    calc_forward_dependencies favor favorReleasers scheduling instrs
  in let bw_deps_cnt = calc_backward_dependencies_count fw_deps
  in let toCP = match pickHeur with
             | OrigOrder -> fun id -> -(IdMap.find id orig_order)
             | CritPath  -> calc_critical_path_lengths fw_deps bw_deps_cnt
  in let foldReady id (n,_) = if n=0 then cons (CP_Id(toCP id,id)) else return
  in let all = IdMap.map (fun cnt -> (cnt, 0)) bw_deps_cnt
  in let ready = sort cmp_cpid (IdMap.fold foldReady all [])
  in let waiting = fold_right (fun (CP_Id(_,i)) -> IdMap.remove i) ready all
  in flatinssched maxPerCycle favor favorReleasers resources scheduling
                  idmap fw_deps toCP (ready, ESTIdSet.empty) waiting
```

A.4.5 Main Program Entry Point

The following module implements the program entry point of the scheduler.

```
open List
open Basics
open Gil_basics
open Gil_IO
open Instr_def
open Instr_IO
open Execution_modelling
open Inssched


(* GLOBAL VARIABLES ********************************************************)

let maxInstrsPerCycle = ref 1
and resources         = ref []
and scheduling        = ref []
and favor             = ref []
and favorReleasers    = ref []
```

```
and pickHeur          = ref CritPath

(* PREPARATION ***********************************************************)

let gilTerm_of_iTuple (t,cplen,i) =
  GT_Struct("t_cplen_i", [| gilTerm_of_int t; gilTerm_of_int cplen; i |])

let do_scheduling instrs =
  let instrs =
        instrsToInstructionscheduled
          !pickHeur !maxInstrsPerCycle !favor !favorReleasers
          !resources !scheduling (list_of_gilTerm instrs)
  in let instrs' = map gilTerm_of_iTuple instrs
  in GT_Struct("t", [| GT_List (map get3of3 instrs); GT_List instrs' |])


(* PROGRAM START-UP ******************************************************)

let latency_of_gilTerm = function
  | GT_Struct("dst_lat", [| d; lat |]) -> (d, int_of_gilTerm lat)
  | t -> illTerm "latency_of_gilTerm" t

let schedInfo_of_gilTerm term =
  let mkSchedInfo' annoReq ls xS xD reqRes =
    (annoReq, map latency_of_gilTerm ls, xS, xD, reqRes_of_gilTerm reqRes)
  in match term with
       | GT_Struct("i_lats_res", [| i; GT_List ls; reqRes |]) ->
           (i, mkSchedInfo' [] ls [] [] reqRes)
       | GT_Struct("i_lats_xsrcs_xdsts_res",
                   [| i; GT_List ls; GT_List xS; GT_List xD; reqRes |]) ->
           (i, mkSchedInfo' [] ls xS xD reqRes)
       | GT_Struct("i_annoreq_lats_xsrcs_xdsts_res",
                   [| i; GT_List annoReq; GT_List ls; GT_List xS; GT_List xD;
                      reqRes |]) ->
           (i, mkSchedInfo' annoReq ls xS xD reqRes)
       | t -> illTerm "schedulingInfo_of_gilTerm" t

let favor_of_gilTerm = function
  | GT_Struct("i_howmuch", [| i; howmuch |]) -> (i, int_of_gilTerm howmuch)
  | t -> illTerm "favor_of_gilTerm" t

let favorReleasers_of_gilTerm = function
  | GT_Struct("d_howmuch", [| d; howmuch |]) -> (d, int_of_gilTerm howmuch)
  | t -> illTerm "favorReleasers_of_gilTerm" t

let pickHeur_of_gilTerm = function
  | GT_Atom "original_order" -> OrigOrder
  | GT_Atom "critical_path"  -> CritPath
  | t -> illTerm "pickHeur_of_gilTerm" t

(***)

let process_argument = function
  | ("pickingHeuristics", [| p |])      -> pickHeur := pickHeur_of_gilTerm p
  | ("maxInstrsPerCycle", [| m |])      -> maxInstrsPerCycle :=
                                             int_of_gilTerm m
  | ("resources", [| GT_List r |])      -> resources := r
  | ("scheduling", [| GT_List x |])     -> scheduling :=
                                             map schedInfo_of_gilTerm x
  | ("favor", [| GT_List f |])          -> favor := map favor_of_gilTerm f
  | ("favorReleasers", [| GT_List x |]) -> favorReleasers :=
                                             map favorReleasers_of_gilTerm x
  | _ -> ()

let main =
```

```
  gilFilter2Args process_argument do_scheduling (assert_and_get_two_args ());
  exit 0
```

A.5 Register Allocation

The register allocator comprises two modules, listed in the following.

A.5.1 regalloc_basics.ml: Basic Definitions

```
open Num
open Basics
open Gil_basics
open Gil_IO
open Queue

class ['a] finiteSetLIFO xs0 =
  object
    val mutable xs : 'a list ref = ref xs0
    method draw  = match !xs with [] -> raise Empty | x::xs1 -> xs := xs1; x
    method add x = xs := x::!xs
  end

class ['a] finiteSetFIFO els0 =
  object
    val q : 'a Queue.t = queue_of_list els0
    method draw   = Queue.take q
    method add el = Queue.add el q
  end

class finiteIntSetNoReuse from' to' =          (* for use with large sets *)
  object
    val mutable next = ref (from' - 1)
    method draw = if !next >= to' then raise Empty; incr next; !next
    method add (_ : int) = ()
  end

class infiniteIntSetNoReuse from' =
  object
    val mutable next = ref (from' - 1)
    method draw = incr next; !next
    method add (_ : int) = ()
  end

class ['a] twoSetsPlusSet s1 s2 s3 =
  object
    method draw          = ((try s1#draw with Empty -> s2#draw) : 'a)
    method add (el : 'a) = ((s3#add el) : unit)
  end

class ['a] encapsulateSet f_draw f_add s =
  object
    method draw = ((f_draw (s#draw)) : 'a)
    method add (el : 'a) = ((s#add (f_add el)) : unit)
  end

class ['a, 'b, 'c] finiteMap (eq_keys   : ('a -> 'a -> bool))
                             (eq_values : ('b -> 'b -> bool)) =
  object (self)
    val mutable bindings : ('a * 'b) list ref = ref []

    method add r v = bindings := (r,v)::!bindings       (* young *)
```

```
    method assoc r =                            (* quite defensive *)
      match List.partition (fun (r',_) -> eq_keys r r') !bindings with
        | ([el], rest) -> (el, rest)
        | _ -> failwith "finiteMap#assoc"

    method remove r = bindings := snd (self#assoc r)
    method touch r  = let (el, rest) = self#assoc r in bindings := el::rest

    method findMaximum (f : int -> ('a * 'b) -> int option) =
      snd (snd (pick_maximum fst (mapOptionWithN 1 f !bindings)))
  end

type ('a, 'b) allocationResult = UseNow of 'a | Spill_ThenUse of 'b * 'a

class ['a, 'b] allocation free bindings force_reuse =
  object (self)
    val mutable released : 'a list ref = ref []

    method allocate (v : 'b) (eval : (int -> ('a * 'b) -> int option)) =
      let ((r : 'a), action) =
        match !released with
          | r::rs ->
              released := rs;
              (r, UseNow r)
          | [] ->
              try let r = free#draw in (r, UseNow r)
              with Empty ->
              begin
                let (r,(v' : 'b)) = bindings#findMaximum eval
                in let _ = bindings#remove r
                in (r, Spill_ThenUse(v',r))
              end
      in bindings#add r v; action

    method free (r : 'a) = (bindings#remove r; free#add r : unit)

    method mark (r : 'a) = (bindings#touch r : unit)

    method release (r : 'a) =
      (bindings#remove r;
       if force_reuse then released := r::!released else free#add r : unit)

    method freeReleased () =
      List.iter (fun x -> free#add x) !released;
      released := []

    method replace (r : 'a) (v : 'b) =
      (bindings#remove r;
       bindings#add r v : unit)
  end

(***)

let create_buffered_set set (modestly,fifo) =
  let (b_modestly, b_fifo) = (bool_of_string modestly, bool_of_string fifo)
  in let buf = if b_fifo then new finiteSetFIFO [] else new finiteSetLIFO []
  in let (s1,s2) = if b_modestly then (buf,set) else (set,buf)
  in new twoSetsPlusSet s1 s2 buf

let buffer_of_creationGilTerm = function
  | GT_Struct("finite_from_fifo_modestly",
              [| GT_Int amount; GT_Int start; GT_Atom fifo; GT_Atom m |]) ->
      let (s0,n) = (int_of_num start, int_of_num amount)
      in let unbuffered = new finiteIntSetNoReuse s0 (s0 + n - 1)
      in create_buffered_set unbuffered (m, fifo)
```

```
  | GT_Struct("infiniteFrom", [| GT_Int n |]) ->
      new infiniteIntSetNoReuse (int_of_num n)
  | t -> illTerm "buffer_of_creationGilTerm" t

let set_of_gilTerm id t =
  let int_of_gilTerm = function
    | GT_Struct(id', [| GT_Int n |]) when id = id' -> int_of_num n
    | t -> illTerm "intTerm_of_gilTerm" t
  and gilTerm_of_int i = GT_Struct(id, [| GT_Int (num_of_int i) |])
  in match t with
       | GT_Struct("first_then", [| first_set; then_set |]) ->
           let first' = buffer_of_creationGilTerm first_set
           and then'  = buffer_of_creationGilTerm then_set
           in let buffered = new twoSetsPlusSet first' then' then'
           in new encapsulateSet gilTerm_of_int int_of_gilTerm buffered
       | GT_Struct("modestly_fifo_creation",
                   [| GT_Atom modestly; GT_Atom fifo; creat |]) ->
           let unbuffered = buffer_of_creationGilTerm creat
           in let buffered = create_buffered_set unbuffered (modestly, fifo)
           in new encapsulateSet gilTerm_of_int int_of_gilTerm buffered
       | t -> illTerm "set_of_gilTerm" t
```

A.5.2 `regalloc.ml`: Main Program Entry Point

```
open Basics
open List
open Num
open Gil_basics
open Gil_IO
open Instr_def
open Instr_IO
open Queue
open List
open Regalloc_basics


(* GLOBAL VARIABLES ********************************************************)

let srcToInfo     = ref StringMap.empty        (* WO @ program start up    *)
let constantLoads = ref []                     (* promised to load consts  *)


(* REGISTER ALLOCATION *****************************************************)

let srcInfo_of_gilTerm v =
  try StringMap.find (functor_of_gilTerm v) !srcToInfo
  with Not_found -> illTerm "srcInfo_of_gilTerm" v

type vregfileentry =                           (* REG. FILE ENTRY **********)
  | Fresh                                      (*   Unmapped               *)
  | In of gilTerm                              (*   Resides in Register    *)
  | Out of gilTerm                             (*   Lies on the Stack      *)
  | InOut of gilTerm * gilTerm                 (*   In Reg and On Stack    *)
  | Const of instr                             (*   Creator                *)
  | InConst of gilTerm * instr                 (*   In Reg. + Creator      *)

let vrfe_is_unmapped = function
  | Fresh | Out _   | Const _   -> true
  | In _  | InOut _ | InConst _ -> false

let vrfe_to_rreg = function
  | In r  | InOut(r,_) | InConst(r,_) -> r
  | Fresh | Out _ | Const _ -> failwith "vrfe_to_rreg: register not mapped"
```

```
let dump_reg r v' = function
  | In(r) ->
      let (_,_,_,stack,spillOp,_) = srcInfo_of_gilTerm v'
      in let eLoc = try stack#draw with Empty -> failwith "dump_reg(In _)"
      in ([I(spillOp,[r],[eLoc],[])], Out(eLoc))
  | InOut(_,eLoc) -> ([], Out(eLoc))
  | InConst(_,c)  -> ([], Const(c))
  | Fresh | Out _ | Const _ -> failwith "dump_reg"

let reload_reg r v = function
  | In _ | InConst _ | InOut _ -> failwith "reload_reg: already mapped"
  | Fresh -> ([], In(r))
  | Const (I(op,srcs,dsts,annos) as i) ->
      ([I(op,srcs,[r],annos)], InConst(r,i))
  | Out eLoc ->
      let (_,_,_,_,_,reloadOp) = srcInfo_of_gilTerm v
      in ([I(reloadOp,[eLoc],[r],[])], InOut(r,eLoc))

let release_dead_reg vRF v =
  let (_,rRF,_,stack,_,_) = srcInfo_of_gilTerm v
  in match GilTermMap.find v vRF with
       | Fresh | Out _ | Const _ -> illTerm "release_dead_reg" v
       | In r  | InConst(r, _)   -> rRF#release r; rRF
       | InOut(r, eLoc) -> rRF#release r; stack#add eLoc; rRF

let add_substis vRF =
  fold_left
    (fun m k -> GilTermMap.add k (vrfe_to_rreg (GilTermMap.find k vRF)) m)

let update_future_refs (refs,dead) v = match tl (GilTermMap.find v refs) with
  | []         -> (GilTermMap.remove v refs, v::dead)
  | _::_ as xs -> (GilTermMap.add v xs refs, dead)

let spillGain refs keep index (r,v) =
  if mem_gilTermList v keep then None
  else try Some (hd (GilTermMap.find v refs)) with _ -> illTerm "spillGain" v

let allocate_reg' vRF instrs v = function
  | UseNow r -> (r,vRF,instrs)
  | Spill_ThenUse(v',r) ->
      let (spill_code,vrfe) = dump_reg r v' (GilTermMap.find v' vRF)
      in (r,(GilTermMap.add v' vrfe vRF),spill_code @ instrs)

let allocate_reg refs keep vRF instrs v =
  let (_,rRF,_,_,_,_) = srcInfo_of_gilTerm v
  in allocate_reg' vRF instrs v (try rRF#allocate v (spillGain refs keep)
                              with _ -> failwith "allocate_reg")

let spillNmap refs keep to_vrfe (vRF,instrs) v =
  let (r,vRF,instrs) = allocate_reg refs keep vRF instrs v
  in (GilTermMap.add v (to_vrfe r) vRF,instrs)

let spillNreload refs keep (vRF,instrs) (v,_) =
  let (r,vRF,instrs) = allocate_reg refs keep vRF instrs v
  in try
       let (reload_instrs,vrfe) = reload_reg r v (GilTermMap.find v vRF)
       in (GilTermMap.add v vrfe vRF, reload_instrs @ instrs)
     with Not_found -> (GilTermMap.add v (In(r)) vRF,instrs)

let add_to_history vRF =
  fold_right (fun v xs -> (v, vrfe_to_rreg (GilTermMap.find v vRF))::xs)

let instr_is_constLoad instr =
  memberChk_gilTermList (gilTerm_of_instr instr) !constantLoads
```

```
let update_references refs vRF regs =
  let (refs,dead) = fold_left update_future_refs (refs,[]) regs
  in let rRFs_rs = List.map (release_dead_reg vRF) dead
  in (refs, rRFs_rs, fold_right GilTermMap.remove dead vRF)

(* ideally, when reusing a register, we want to take to pick the one
 * that has been dead the longest, *unless* we could reuse one of the
 * source operands of the instruction being processed -- no matter
 * what other allocation options are enabled. also reusing (in this way)
 * should be even preferred to using a fresh register. *)
let regalloc1 (refs,vRF0,history,instrs) (I(op,srcs0,dsts0,annos) as i) =
  let srcs = uniq_gilTermList (filter (functor_is_in_map !srcToInfo) srcs0)
  and dsts = uniq_gilTermList (filter (functor_is_in_map !srcToInfo) dsts0)
  in let dstsMinusSrcs = filter (nonMember_gilTermList' srcs) dsts
  in let all = map (fun v -> (v, try GilTermMap.find v vRF0
                              with Not_found -> Fresh)) srcs
  in let unmapped = filter (fun (_,vrfe) -> vrfe_is_unmapped vrfe) all
  in let (vRF1,instrs) =
        fold_left (spillNreload refs (srcs @ dsts)) (vRF0,instrs) unmapped
  in let substis1 = add_substis vRF1 GilTermMap.empty srcs
  in let (refs,rRFs_rs1,vRF2) = update_references refs vRF1 srcs
  in let to_vrfe r = if instr_is_constLoad i then InConst(r,i) else In r
  in let (vRF3,instrs) =
        fold_left (spillNmap refs dsts to_vrfe) (vRF2,instrs) dstsMinusSrcs
  in let substis   = add_substis vRF3 substis1 dsts
  in let history'  = add_to_history vRF1 (map fst unmapped) history
  in let history'' = add_to_history vRF3 dsts history'
  in let instr'    = I(op, map (subst_in_gilTerm substis) srcs0,
                        map (subst_in_gilTerm substis) dsts0, annos)
  in let (refs,rRFs_rs2,vRF) = update_references refs vRF3 dstsMinusSrcs
  in let _ = iter (fun rRF -> rRF#freeReleased ()) (rRFs_rs1 @ rRFs_rs2)
  in (refs, vRF, history'', instr'::instrs)

let defsUses_for_instrs xs =
  let defUse (n,rs) (I(_,s,d,_)) =
    let sd_uniq = uniq_gilTermList (rev_append d s)
    in let relevant = filter (functor_is_in_map !srcToInfo) sd_uniq
    in (n + 1, fold_right (gilTermMap_addE' n) relevant rs)
  in GilTermMap.map rev (snd (fold_left defUse (0, GilTermMap.empty) xs))

let regAllocN instrs =
  let st0   = (defsUses_for_instrs instrs, GilTermMap.empty, [], [])
  in let st = fold_left regalloc1 st0 instrs
  in let (_,_,history',instrs') = st in (rev history', rev instrs')


(* PREPARATION *************************************************************)

let do_allocation ts =
  let instrs0 = map instr_of_gilTerm (list_of_gilTerm ts)
  in let (history, instrs) = regAllocN instrs0
  in let instrs'  = map gilTerm_of_instr instrs
  in let history' = map (fun (f,t) -> GT_Struct("f_t", [| f; t |])) history
  in GT_Struct("t", [| GT_List instrs'; GT_List history' |])

(* PROGRAM START-UP ********************************************************)

let dstToInfo   = ref StringMap.empty
and stackToInfo = ref StringMap.empty

let process_argument = function
  | ("dst_forceReuse_set", [| GT_Atom dst; GT_Atom force_reuse; set |]) ->
      let free = set_of_gilTerm dst set
      and fmap = new finiteMap eq_gilTerm eq_gilTerm
      and force_reuseB = bool_of_string force_reuse
```

```
      in let rRF = new allocation free fmap force_reuseB
      in dstToInfo := StringMap.add dst rRF !dstToInfo
  | ("src_dst_stack", [| GT_Atom src; GT_Atom dst; GT_Atom stack; |]) ->
      let dst_obj = StringMap.find dst !dstToInfo
      and (spillOp,reloadOp,stack_obj) = StringMap.find stack !stackToInfo
      in let t = (dst,dst_obj,stack,stack_obj,spillOp,reloadOp)
      in srcToInfo := StringMap.add src t !srcToInfo
  | ("stack_spill_reload_set", [| GT_Atom id; spill; reload; set |]) ->
      let t = (spill, reload, (set_of_gilTerm id set))
      in stackToInfo := StringMap.add id t !stackToInfo
  | ("constantLoads", [| GT_List ls |]) -> constantLoads := ls
  | _ -> ()

let main =
  gilFilter2Args process_argument do_allocation (assert_and_get_two_args ());
  exit 0
```

Bibliography

[1] Advanced Micro Devices Inc., *AMD Core Math Library (ACML)*, 2006, `http://developer.amd.com/acml`.

[2] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers, 2002.

[3] G. Almasi, R. Bellofatto, J. Brunheroto, C. Causcaval, J. G. Castanos, L. Ceze, P. Crumley, C. Ch. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss, *An Overview of the BlueGene/L System Software Organization*, Proceedings of Euro-Par 2003 – 9th International Conference on Parallel and Distributed Computing, LNCS, vol. 2790, Springer-Verlag, 2003, pp. 543–555.

[4] American National Standards Institute (ANSI), *ISO/IEC 9899 – Programming languages – C*, 1999.

[5] Apple Computer, *vDSP Library*, 2001, `http://developer.apple.com/`.

[6] J. W. Backus, *The history of FORTRAN I, II and III*, Proceedings of the first ACM SIGPLAN conference on the History of programming languages, 1978.

[7] D. H. Bailey, *FFTs in External or Hierarchical Memory*, Supercomputing Journal **4** (1990), pp. 23–35.

[8] L. A. Belady, *A study of replacement algorithms for virtual storage computers*, IBM Systems Journal **5** (1966), no. 2, pp. 78–101.

[9] J. Bilmes, K. Asanović, C. Chin, and J. Demmel, *Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology*, Proceedings of ICS 1997 – International Conference on Supercomputing, ACM Press, 1997, pp. 340–347.

[10] S. Chatterjee, L. R. Bachega, P. Bergner, K. A. Dockser, M. Gupta, F. G. Gustavson, C. A. Lapkowski, G. K. Liu, M. Mendell, R. Nair, C. D. Wait, T. J. C. Ward, and P. Wu, *Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L*, IBM Journal for Research and Development **49** (2005), no. 2/3, pp. 377–392.

[11] D. Chaver, C. Tenllado, L. Pinjuel, M. Prieto, and F. Tirado, *Wavelet transform for large scale image processing on modern microprocessors*, Proceedings of VECPAR 2002 – 5th International Conference on High Performance

Computing for Computational Science, LNCS, vol. 2565, Springer-Verlag, 2002, pp. 549–562.

[12] CodePlay Ltd., *Vector C Compiler*, 2005, `http://www.codeplay.com`.

[13] J. W. Cooley and J. W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Mathematics of Computation **19** (1965), pp. 297–301.

[14] R. Crandall and J. Klivington, *Supercomputer-Style FFT Library for the Apple G4*, Advanced Computation Group, Apple Computer, 2002.

[15] Institut National de Recherche en Informatique et Automatique (INRIA), *The Caml Language*, 2006, `http://caml.inria.fr`.

[16] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, *Self-Adapting Linear Algebra Algorithms and Software*, Proceedings of the IEEE **93** (2005), no. 2, pp. 293–312.

[17] K. Dockser, *Oedipus Architecture: Extensions to PowerPC BookE for Hummer*^{2}, Tech. report, IBM, August 2001.

[18] J. Dongarra and F. Sullivan, *Top Ten Algorithms of the Century*, Computing in Science and Engineering **2** (2000), pp. 22–23.

[19] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*, SIAM Press, Philadelphia, 1998.

[20] J. J. Dongarra, F. G. Gustavson, and A. Karp, *Implementing linear algebra algorithms for dense matrices on a vector pipeline machine*, SIAM Review **26** (1984), pp. 91–112.

[21] S. Egner and M. Püschel, *The AREP Home Page*, 2000, `http://www.ece.cmu.edu/~smart/arep/arep.html`.

[22] ———, *Symmetry-Based Matrix Factorization*, Journal of Symbolic Computation **37** (2004), no. 2, pp. 157–186.

[23] R. J. Fisher and H. G. Dietz, *Compiling for SIMD Within A Register*, Proceedings of LCPC 1998 – 11th Workshop on Languages and Compilers for Parallel Computing, 1998, pp. 290–304.

[24] R. J. Fisher and H. G. Dietz, *The* SCC *Compiler: SWARing at MMX and 3DNow*, Proceedings of LCPC 2000 – 12th Annual Workshop on Languages and Compilers for Parallel Computing, LNCS, vol. 1863, Springer-Verlag, 2000, pp. 399–414.

[25] F. Franchetti, J. Lorenz, and C. W. Ueberhuber, *Low Communication FFTs*, Aurora Technical Report TR2002-27, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2002.

[26] F. Franchetti and M. Püschel, *A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms*, Proceedings of IPDPS 2002 – 16th International Parallel and Distributed Processing Symposium, 2002.

[27] F. Franchetti and M. Püschel, *Short vector code generation for the discrete Fourier transform*, Proceedings of IPDPS 2003 – 17th International Parallel and Distributed Processing Symposium, 2003, pp. 22–26.

[28] M. Frigo, *A fast Fourier transform compiler*, Proceedings of PLDI 1999 – ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 169–180.

[29] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, Proceedings of ICASSP 1998 – IEEE International Conference on Acoustics Speech and Signal Processing, vol. 3, 1998, pp. 1381–1384.

[30] ———, *The design and implementation of FFTW3*, Proceedings of the IEEE **93** (2005), no. 2, pp. 216–231.

[31] M. Frigo and S. Kral, *The Advanced FFT Program Generator GENFFT*, Aurora Technical Report TR2001-03, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2001.

[32] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, Proceedings of FOCS 1999 – 40th Annual Symposium on Foundations of Computer Science, 1999, pp. 285–297.

[33] K. S. Gatlin and L. Carter, *Faster FFTs via Architecture-Cognizance*, Proceedings of PACT 2000 – 9th International Conference on Parallel Architectures and Compilation Techniques, 2000, pp. 249–260.

[34] S. W. Golomb and L. D. Baumert, *Backtrack programming*, Journal of the ACM **12** (1965), pp. 516–524.

[35] G. H. Golub and C. F. Van Loan, *Matrix computations*, 3nd ed., Johns Hopkins University Press, Baltimore, 1996.

[36] J. R. Goodman and W. C. Hsu, *Code scheduling and register allocation in large basic blocks*, Proceedings of ICS 1988 – 2nd International Conference on Supercomputing, 1988, pp. 442–452.

[37] The GAP Group, *The GAP (groups, algorithms and programming) Web Page*, `http://www.gap-system.org`.

[38] J. Guo, M. Garzaran, and D. Padua, *The power of Belady's algorithm in register allocation for long basic blocks*, Languages and Compilers for Parallel Computing, LNCS, vol. 2958, Springer-Verlag, 2004, pp. 374–390.

[39] F. Gygi, E. Draeger, B. R. de Supinski, R. K. Yates, F. Franchetti, S. Kral, J. Lorenz, C. W. Ueberhuber, J. Gunnels, and J. Sexton, *Large-Scale First-Principles Molecular Dynamics Simulations on the BlueGene/L Platform using the Qbox Code*, Proceedings of the ACM/IEEE Conference on Supercomputing, 2005, Gordon Bell Prize runner-up.

[40] S. Hoxey, F. Karim, B. Hay, and H. Warren (editors), *The PowerPC Compiler Writer's Guide*, Warthman Associates, 1996.

[41] Intel Corporation, *AP-808 split radix fast Fourier transform using streaming SIMD extensions*, 1999.

[42] ______, *Intel C/C++ Compiler*, 2005, `http://www.intel.com/software/products/compilers`.

[43] ______, *Math Kernel Library (MKL)*, 2005, `http://www.intel.com/software/products/mkl`.

[44] *International Technology Roadmap for Semiconductors (ITRS)*, 2006, `http://public.itrs.net`.

[45] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, *A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures*, Circuits, Systems and Signal Processing **9** (1990), pp. 449–500.

[46] J. R. Johnson, R. W. Johnson, C. P. Marshall, J. E. Mertz, D. Pryor, and J. H. Weckel, *Data flow, the FFT, and the CRAY T3E*, Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, 1999.

[47] H. Karner and C. W. Ueberhuber, *Overlapped Four-Step FFT Computation*, Parallel Computation, LNCS, vol. 1557, Springer-Verlag, 1999, pp. 590–591.

[48] R. Korf, *Depth-First Iterative Deepening: An optimal admissible tree search*, Artificial Intelligence **27** (1985), no. 1, pp. 97–109.

[49] S. Kral, *FFT Compiler Techniques for 2-way SIMD Architectures*, Master's thesis, Institute for Applied Mathematics and Numerical Analysis, Vienna University of Technology, 2004.

[50] ______, *The* FFTW-GEL *Web Page*, 2005, `http://www.complang.tuwien.ac.at/skral/fftwgel.html`.

[51] S. Kral, F. Franchetti, J. Lorenz, and C. W. Ueberhuber, *SIMD Vectorization of Straight Line FFT Code*, Proceedings of Euro-Par 2003 – 9th International Conference on Parallel and Distributed Computing, LNCS, vol. 2790, Springer-Verlag, 2003, pp. 251–260.

[52] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber, and P. Wurzinger, *FFT Compiler Techniques*, Proceedings of CC 2004 – 13th International Conference on Compiler Construction, LNCS, vol. 2985, Springer-Verlag, 2004, pp. 217–231.

[53] S. H. Lamson, Sciport, 1995, `http://www.netlib.org/scilib`.

[54] S. Larsen and S. Amarasinghe, *Exploiting superword level parallelism with multimedia instruction sets*, ACM SIGPLAN Notices **35** (2000), no. 5, pp. 145–156.

[55] R. Leupers and S. Bashford, *Graph-based code selection techniques for embedded processors*, ACM Transactions on Design Automation of Electronic Systems **5** (2000), no. 4, pp. 794–814.

[56] J. Lorenz, S. Kral, F. Franchetti, and C. W. Ueberhuber, *Vectorization techniques for the Blue Gene/L double FPU*, IBM Journal of Research and Development **49** (2005), no. 2/3, pp. 437–446.

[57] M. Lorenz, D. Koffmann, S. Bashford, R. Leupers, and P. Marwedel, *Optimized address assignment for DSPs with SIMD memory accesses*, Proceedings of ASP-DAC 2001 – Asia and South Pacific Design Automation Conference, 2001, pp. 415–420.

[58] M. Lorenz, L. Wehmeyer, T. Dräger, and R. Leupers, *Energy aware Compilation for DSPs with SIMD instructions*, Proceedings of LCTES/SCOPES 2002 – Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems, 2002, pp. 94–101.

[59] D. Mirkovic, *Automatic Performance Tuning in the UHFFT Library*, Proceedings of ICCS 2001 – International Conference on Computational Science, LNCS, vol. 2073, Springer-Verlag, 2001, pp. 71–80.

[60] J. E. Moreira, G. Almasi, C. Archer, R. Bellofatto, P. Bergner, J. R. Brunheroto, M. Brutman, J. G. Castanos, P. G. Crumley, M. Gupta, T. Inglett, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mendell, M. Mundy, D. Reed, R. K. Sahoo, A. Sanomiya, R. Shok, B. Smith, and G. G. Stewart, *Blue Gene/L Programming and Operating Environment*, IBM Journal for Research and Development **49** (2005), no. 2/3, pp. 367–376.

[61] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso, SPIRAL*: Portable Library of Optimized Signal Processing Algorithms*, 1998.

[62] J. M. F. Moura, M. Püschel, D. Padua, and J. Dongarra, *Scanning the Issue: Special Issue on Program Generation, Optimization, and Platform Adaptation*, Proceedings of the IEEE **93** (2005), no. 2, pp. 211–215.

[63] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997.

[64] K. Nadehara, T. Miyazaki, and I. Kuroda, *Radix-4 FFT implementation using SIMD multi-media instructions*, Proceedings of ICASSP 1999 – IEEE International Conference on Acoustics Speech and Signal Processing, 1999, pp. 2131–2135.

[65] Iain Nicholson, LIBSIMD, 2005, `http://libsimd.sourceforge.net/`.

[66] *Performance Application Programming Interface (PAPI)*, 2005, `http://icl.cs.utk.edu/papi`.

[67] D. A. Patterson and J. L. Hennessy, *Computer Organisation and Design: The Hardware/Software Interface*, 2nd ed., Morgan Kaufmann Publishers, 1998.

[68] M. Püschel, *Decomposing Monimial Representations of Solvable Groups*, Symbolic Computation **34** (2002), no. 6, pp. 561–596.

[69] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, *SPIRAL: Code Generation for DSP Transforms*, Proceedings of the IEEE **93** (2005), no. 2, pp. 232–275.

[70] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura, *Fast Automatic Generation of DSP Algorithms*, Proceedings of ICCS 2001 – International Conference on Computational Science, LNCS, vol. 2073, Springer-Verlag, 2001, pp. 97–106.

[71] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, SPIRAL*: A generator for platform-adapted libraries of signal processing algorithms*, Journal of High Performance Computing and Applications **18** (2004), no. 1, pp. 21–45.

[72] P. Rodriguez, *A Radix-2 FFT Algorithm for Modern Single Instruction Multiple Data (SIMD) Architectures*, Proceedings of ICASSP 2002 – IEEE International Conference on Acoustics Speech and Signal Processing, 2002.

[73] E. Sikha and R. Simpson, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, 2nd ed., Morgan Kaufmann Publishers, 1995.

[74] B. Singer and M. Veloso, *Stochastic Search for Signal Processing Algorithm Optimization*, Proceedings of Supercomputing 2001, 2001.

[75] Z. Somogyi, F. Henderson, and T. Conway, *Mercury: an efficient purely declarative logic programming language*, Proceedings of the Australian Computer Science Conference, 1995, pp. 499–512.

[76] ———, *The execution algorithm of Mercury: an efficient purely declarative logic programming language*, Journal of Logic Programming **29** (1996), no. 1-3, pp. 17–64.

[77] N. Sreraman and R. Govindarajan, *A vectorizing compiler for multimedia extensions*, International Journal of Parallel Programming **28** (2000), pp. 363–400.

[78] Y. N. Srikant and P. Shankar (editors), *The Compiler Design Handbook: Optimizations and Machine Code Generation*, CRC Press, 2003.

[79] L. S. Sterling and E. Y. Shapiro, *The Art of Prolog*, MIT Press, 1986.

[80] P. N. Swarztrauber, *FFT Algorithms for Vector Computers*, Parallel Computing **1** (1984), pp. 45–63.

[81] P. van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[82] C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Frontiers in Applied Mathematics, vol. 10, SIAM, Philadelphia, 1992.

[83] Y. Voronenko and M. Püschel, *Multiplierless multiple constant multiplication*, 2006, ACM Transactions on Algorithms, to appear.

[84] R. J. Walker, *An enumerative technique for a class of combinatorial problems*, Combinatorial Analysis (Proceedings of the Symposium on Applied Mathematics) **10** (1960), pp. 91–94.

[85] Z. Wang, *Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform*, IEEE Transactions on Acoustics, Speech, and Signal Processing **32** (1984), no. 4, pp. 803–816.

[86] B. Wess and T. Zeitlhofer, *On the Phase Coupling Problem Between Data Memory Layout Generation and Address Pointer Assignment*, In Proceedings of SCOPES 2004 – Software and Compilers for Embedded Systems, 8th

International Workshop, LNCS, vol. 3199, Springer-Verlag, 2004, pp. 152–166.

[87] R. C. Whaley, *User contribution to* ATLAS, 2003, `http://www.cs.utk.edu/~rwhaley/papers/atlas_contrib.ps`.

[88] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated Empirical Optimizations of Software and the* ATLAS *Project*, Parallel Computing **27** (2001), pp. 3–35.

[89] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.

[90] J. Xiong, J. Johnson, R. Johnson, and D. Padua, *SPL: A Language and Compiler for DSP Algorithms*, Proceedings of PLDI 2001 – ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001, pp. 298–308.

[91] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, 1991.