Application-Specific Instruction-Set Architectures for Embedded DSP Applications

by

Mazen A. R. Saghir

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
CANADA

Application-Specific Instruction-Set Architectures for Embedded DSP Applications

Doctor of Philosophy, 1998

Mazen A. R. Saghir

Graduate Department of Electrical and Computer Engineering

University of Toronto

Abstract

Programmable digital signal processors (DSPs) are microprocessors with specialized architectural features for the efficient execution of digital signal processing algorithms at relatively low cost. These features also make DSPs difficult targets for high-level language (HLL) compilers, and require that assembly language programming be used to fully exploit their capabilities.

As applications become larger and more complex, and as design cycles are required to be shorter, it is important to move towards using more HLL programming. To achieve this, more compiler-friendly DSP architectures and better DSP compiler technology are required. As DSP cores become more pervasive in embedded system-on-a-chip designs, a need is also emerging for application-specific DSP cores that can be customized to the functional, performance, and cost requirements of a target application, or group of applications.

This dissertation examines the use of VLIW architectures to achieve a suitable compiler target while being able to express the forms of parallelism found in most DSP applications. Performance improvements by factors of 1.8 – 2.8 are shown to be achievable simply by using a VLIW architecture compared to more traditional architectures. A method for reducing the instruction bandwidth and storage requirements of VLIW architectures is also proposed, and its impact on performance and cost is examined.

To handle some of the DSP-specific architectural features, an optimizing C compiler is developed. In particular, two algorithms that enable the compiler to allocate data automatically across dual data-memory banks are developed, and their impact on cost and performance are examined.

Finally, a set of tools that enable a designer to customize the architecture and its instruction set to the requirements of a target application is also presented. This includes an area-estimation model and an instruction-set simulator for measuring cost and execution performance.

بسم الله الرحمن الرحيم

In the name of God, Most Beneficent, Most Merciful.

O my Lord! advance me in knowledge.

Quran 20:114

To my grandmother, Jihan Noueiri Saghir,

and

the memory of my grandfather, Abdul-Karim Mahmoud Al-Ta'i.

Acknowledgments

This dissertation could not have been written without the help of many individuals and organizations, each of whom contributed to it in their own way, and to whom I am truly grateful.

To my supervisor, Professor Paul Chow, for his technical feedback, friendly advice, financial assistance, and for giving me the space to think on my own. Most of all, however, for his patience and his confidence in me. To the members of my Ph.D. committee, Professors Dimitris Hatzinakos, Corinna G. Lee, David M. Lewis, Thomas C. Wilson, and David B. Wortman, for their careful reading of my dissertation and their valuable suggestions and comments. To Sarah Cherian and Jennifer Rodrigues for their administrative and logistical support. Finally, to the Information Technology Research Centre — now Communications and Information Technology Ontario — for their financial support, and to Nortel Networks for allowing me to finish writing my dissertation after working hours.

To my family: Abdur-Rahman, Rajiha, Dima, and Bachir, for their continued love, support, and encouragement. To my aunt Maha, for all her support through the difficult times. Finally, to Ziad, Amira, Tarek, Dima, and Rania, for once again sharing the warmth of their family with me.

To my friends: Muhammad Jaseemuddin, Mohammed Khalid, Amer Shalaby, and Halim Yanikomeroglu for their wonderful companionship through this long journey. To Rami Rahal and Zahi Anthony Suidan for the countless L-walks up and down Younge Street, and for the hours of stimulating discussions from which I have learned much about *id*, *ego*, and *superego*. To Iyad Al-Attar, for all the fun times over tea and coffee, and for explaining the intricacies of IMAX projection systems. To all my friends, past and present, in LP392: Ravi Ananth, Jason Anderson, Vaughn Betz, Ken Chalmers, Keith Farkas, Vincent Gaudet, Warren Gross, Jeff Jacob, Wai Ming Louie, Javad Omidi, Bryn Owen, Khoman Phang, Jason Podaima, Yaska Sankar, Ken Schultz, Ali Sheikholeslami, Nirmal Sohi, Mark Stoodley (thanks for the technical help!), Jordan Swartz, Marcus Van Ierssel, Qiang Wang, and Andy Gean Ye. Although those rowdy Euchre games made it difficult to concentrate on writing at times, thanks for all the fun times in the lab and for making my years at U of T truly memorable. Last, but certainly not least, to Walid Abu-Salman, Houssam Ali-Hassan, Habib Chahine, Muhammad Dajani, Souheil El-Hakim, Salim Farah, Mazen Haddad, Antoine Haroun, George Nasr, and Sami Tarazi for being great friends.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The high performance and low cost of programmablel digital signal processors (DSPs) have made them popular choices for implementing embedded DSP systems. However, their specialized architectural features, which include dual data-memory banks; specialized addressing modes; accumulator-based datapaths with few data registers; and tightly-encoded instruction sets that only support limited instances of parallelism, have also made DSPs difficult targets for high-level language (HLL) compilers. In fact, DSPs must still be programmed in assembly language to fully exploit specialized architectural features and ensure code compactness.

As applications become larger and more complex, and as design cycles are required to be shorter, it no longer becomes feasible to rely on assembly language programming to achieve the desired levels of performance and code compactness. Instead, more compiler-friendly DSP architectures and better DSP compiler technology are needed. In addition, as DSP cores become more pervasive in embedded system-on-a-chip (SoC) designs, there is a need for application-specific DSP cores that can be customized to the functional, performance, and cost requirements of a particular application, or group of applications.

One architectural style that is particularly well-suited for embedded DSP systems is the VLIW architecture. VLIW architectures support the exploitation of high levels of fine-grained parallelism, which are commonly found in DSP applications. They are also easy targets for HLL compilers, and their datapaths can easily be customized to the requirements of a target application. However, a serious impediment to the widespread use of VLIW architectures in cost-concious embedded systems is their high instruction bandwidth and storage requirements.

This dissertation addresses some of the issues facing current DSP compilers and VLIW architectures aimed at embedded DSP applications. It also introduces an iterative, application-driven design methodology for customizing a datapath and its corresponding instruction-set architecture to the functional, performance, and cost requirements of a target application, or set of applications.

The research infrastructure includes a suite of DSP benchmarks consisting of 12 kernels and 10 applications; a VLIW-based model architecture; an optimizing C compiler capable of exploiting the DSP-specific features of the architecture; an instruction-set simulator to measure the execution performance of the benchmarks on the architecture; and an area-estimation model to measure the cost of the datapath. Two algorithms that enable the compiler to allocate data automatically across the dual data-memory banks of the model architecture are presented, and their impact on performance and cost are examined. A new approach to encoding and decoding long instructions to reduce the instruction bandwidth and storage requirements of the model architecture is also proposed, and its impact on performance and cost is examined. Finally, a set of tools that enable a designer to customize the architecture and its instruction set to the requirements of a target application, or group of applications, is also presented.

1.1 Main Contributions

- Making the case for using VLIW-based architectures in embedded DSP systems.
- Developing an iterative, application-driven methodology for customizing the datapath and instruction-set of a VLIW-based architecture to the functional, performance, and cost requirements of a target application, or group of applications.
- Developing a new set of tools for supporting the design methodology. These include an area-estimation model for measuring the cost associated with a datapath; a long-instruction encoding pass for measuring the instruction-storage requirements of an application; and an architectural template generator for specifying a datapath with a minimal set of hardware resources that meets the functional requirements of a target application.
- Developing two new compiler algorithms for exploiting the dual data-memory banks of a model DSP architecture.
- Developing a new technique for encoding and decoding long instructions that minimizes instruction-storage and bandwidth requirements, while preserving the flexibility of the long-instruction format.

1.2 Organization

The dissertation is organized into eight chapters and two appendixes. Chapter 2 describes the iterative design methodology used to customize VLIW-based architectures to the functional, performance, and cost requirements of a target application, or group of applications. It also describes the set of tools used to support the design methodology, including the DSP benchmark suite, the VLIW-based model architecture, the optimizing C compiler, the instruction-set simulator, and the datapath cost model. Chapter 3 describes two compiler algorithms for exploiting the dual data-memory banks of the model architecture, and discusses their impact on performance and cost. Chapter 4 makes the case for using VLIW-based architectures, and compares their advantages and disadvantages to more traditional, tightly-encoded DSPs. Chapter 5 describes a new technique for encoding, storing, and decoding long instructions. Using the design methodology and tools described in Chapter 2 and the long-instruction encoding and storage techniques described in Chapter 5, Chapter 6 shows how the datapath and instruction set of the model architecture can be customized to the functional, performance, and cost requirements of the DSP benchmarks. It also discusses the impact of the instruction word length on performance and cost. Finally, Chapter 7 summarizes the main ideas and results of this dissertation, and suggests future work.

The two appendixes provide useful reference material. Appendix A describes the set of machine operations that can be executed by the model architecture. It also describes the way operations can be encoded using different instruction word lengths. Appendix B explains how the bit-area estimates used in the datapath cost model were derived.

Chapter 2

Design Methodology and Tools

The phenomenal growth in the consumer electronics market has increased the demand for high-performance, low-cost processors for use in embedded systems. To meet these demands, DSP vendors and semiconductor manufacturers are offering a wide range of processors with varying levels of performance and cost. Since these processors offer good performance and cost they are typically used in first-generation products. Increasingly, however, a new class of processors, called application-specific programmable processors (ASPPs), are used in later-generation products because they can be designed with specific applications in mind, and can therefore be tuned to their specific performance and cost requirements [1].

This chapter describes the methodology and the tools used to design the University of Toronto Digital Signal Processor (UTDSP), an ASPP aimed at embedded DSP applications. Section 2.1 explains the general design methodology, and describes the function of each tool. Sections 2.2 – 2.8 describe the different tools, which include the benchmark suite used to validate the design methodology; the architectural model; the C compiler; the post-optimizer and its role in tuning the architecture to an application; the instruction set simulator and the way it is used to measure performance; the way instructions are encoded to preserve flexibility and minimize cost; and the datapath area model and the way it can be used to measure cost. Section 2.9 then describes how performance and cost are evaluated, and how these can be used to modify the architecture. Finally, Section 2.10 summarizes the main ideas of this chapter.

2.1 Application-Driven Design Methodology

Figure 2.1 shows the framework used to design the UTDSP. The basic idea is to start with a target application, or a small set of applications, developed in a high-level language, then iterate through several design cycles until an ASPP architecture that meets the functional, performance, and cost constraints of the applications is reached. Since the design is guided by the constraints of the application, this design methodology is called *application-driven*.

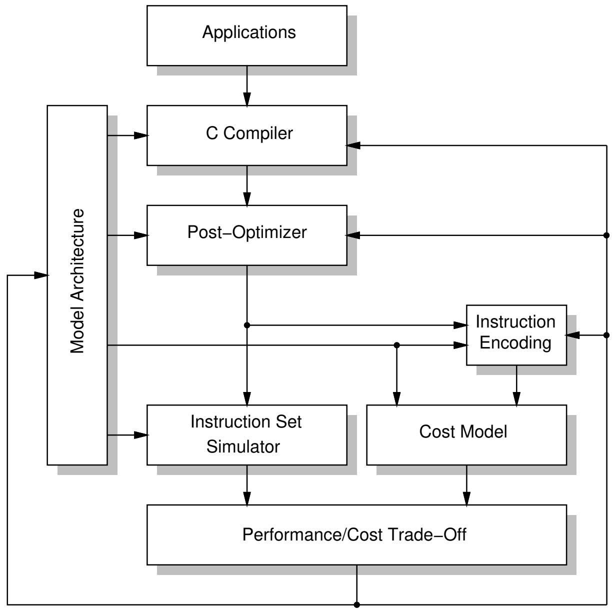


Figure 2.1: Application-driven design methodology.

A key component in the design framework is a flexible model architecture that is easy to configure, and that supports the exploitation of parallelism. The architecture must also have a flexible instruction set that is an easy target for HLL compilers, and that also facilitates the exploitation of parallelism. Initially, the architecture can be configured in an arbitrary manner. After that, it can be modified to improve performance or reduce costs.

Application programs developed in the C programming language are also used as a starting point. C is popular among DSP and embedded systems programmers because it is a portable language, and because programs written in C are easier to debug, upgrade, and maintain than assembly code.

To translate an application program into the simple machine operations that can be executed by the model architecture, a C compiler is used. The C compiler uses the set of machine operations and the number of registers specified for the architecture to generate sequential code and perform register allocation. It also applies standard scalar optimizations to improve the quality of the generated code and enhance execution performance. After generating machine operations, a post-optimizer is used to exploit the DSP-specific features of the model architecture. It also uses the long instruction format specified for the architecture to exploit parallelism among machine operations and bind operations to specific functional units. By using instruction formats that support application-specific instances of parallelism, and encoding them as executable instructions, the architecture can be tuned to the application. However, to minimize storage requirements and reduce cost, instructions must also be encoded in an efficient manner. An instruction encoding pass is therefore used to encode the instructions generated by the post-optimizer.

To simulate the execution of a program on the model architecture and measure its execution performance, an instruction-set simulator is used. A cost model is also used to estimate the area occupied by the datapath of the model architecture. Using this information, the performance and cost of the model architecture can be evaluated to ensure they meet application requirements. If the performance requirements are not met, the architecture can be reconfigured, and the compiler and post-optimizer modified, to exploit more parallelism. On the other hand, if the cost requirements are not met, the area of the datapath can be reduced by removing under-utilized hardware resources. Upon reconfiguring the architecture, the application programs have to be recompiled, and new measurements have to be made for performance and cost. This process is repeated until an architecture that meets the requirements of an application is found.

2.2 Benchmarks

To validate the design methodology and the tools, and to study their impact on the performance and cost of representative workloads, a suite of DSP benchmarks were developed in the C programming language [2]. The benchmark suite consists of six kernels and ten applications.

Table 2.1 shows the kernel benchmarks, which consist of simple algorithms commonly used in DSP applications. For each kernel, two different implementations, corresponding to different input sizes, are used. Although DSP kernels are no substitute for real applications when assessing performance or cost, they nonetheless represent portions of the code that account for significant amounts of execution time in DSP applications. That is why it is important for DSP compilers to be able to generate efficient code for such kernels, and to maximize the utilization of the underlying architecture.

Table 2.2 shows the DSP application benchmarks, which are taken from the areas of speech processing, image processing, and data communications. Such applications are commonly used in embedded systems, and provide a means for assessing the impact of different design tools on performance and cost, as well as validating the design methodology.

To study the effect of different coding styles on the quality of the code generated by the compiler, most benchmarks were implemented in four, different programming styles [3]. The four styles include: pointers (`ptrs`), pointers with software pipelining (`ptrs_SWP`), `arrays`, and arrays with software pipelining (`arrays_SWP`).

In the `ptrs` style, array elements are accessed using pointers. Since the model instruction set architecture uses specific addressing operations, this style enables the compiler to identify addressing operations. In the `ptrs_SWP` style, in addition to using pointers to access array elements, *software pipelining* [4] is used in the source code to expose parallelism within program loops. Software pipelining is frequently used

Kernels		Description
k1	fft_1024	Radix-2, in-place, decimation-in-time fast Fourier transform (FFT)
k2	fft_256	
k3	fir_256_64	Finite impulse response (FIR) filter
k4	fir_32_1	
k5	iir_4_64	Infinite impulse response (IIR) filter
k6	iir_1_1	
k7	latnrm_32_64	Normalized lattice filter
k8	latnrm_8_1	
k9	lmsfir_32_64	Least-mean-squared (LMS) adaptive FIR filter
k10	lmsfir_8_1	
k11	mult_10_10	Matrix Multiplication
k12	mult_4_4	

Table 2.1: DSP kernel benchmarks.

Applications		Description
a1	G721_A	Two implementations of the ITU G.721 ADPCM speech transcoder
a2	G721_B	
a3	V32.modem	V.32 modem encoder/decoder
a4	adpcm	Adaptive differential pulse-coded modulation speech encoder
a5	compress	Image compression using discrete cosine transform (DCT)
a6	edge_detect	Edge detection using 2D convolution and Sobel operators
a7	histogram	Image enhancement using histogram equalization
a8	lpc	Linear predictive coding speech encoder
a9	spectral	Spectral analysis using periodogram averaging
a10	trellis	Trellis decoder

Table 2.2: DSP application benchmarks.

by DSP programmers to expose parallelism within the bodies of program loops. In the `arrays` style, array elements are accessed in a more natural style using standard array notation. To facilitate the identification of addressing operations in later stages of the compiler, special compiler *annotations* are used to preserve information from the source code about where such operations occur. Finally, in the `arrays_SWP` style, software pipelining is used in conjunction with the `arrays` coding style.

For the remainder of this thesis, all results will be presented for benchmarks coded in the `ptrs` style only. Since all benchmarks are coded in the `ptrs` style — not always the case for other coding styles — the are the most complete set. Although the actual results differ for benchmarks coded in different styles, the general trends and overall conclusions remain the same across all programming styles.

2.3 Architectural Model

Within the design framework, a key requirement for the architectural model is flexibility. Flexibility is the ability of the architecture and its instruction set to be configured to meet the functional requirements of an application, as well as its performance and cost constraints. Such flexibility can be achieved by using a model that acts like a template of key datapath components, such as functional units, registers, and memory banks, that are easy to configure. It can also be achieved by using an instruction set that supports different architectural configurations.

Another requirement for the architectural model is that it should be an easy target for HLL compilers. This means that, independent of how the architecture is configured, the code-generation model should remain fixed. This can be achieved, in part, by using simple, RISC-like operations that can be mapped onto different functional units, allowing a large number of these operations to be executed in parallel, and encoding them into the same instruction.

To meet the requirements of flexibility and compiler programmability, an architectural model that is based on a very long instruction word (VLIW) architecture [5] was chosen. A VLIW architecture consists of several functional units that are each controlled by a corresponding field in a long instruction word. Since each field can specify a single operation, several operations can be executed in parallel in a manner similar to horizontal microcode. Typically, an optimizing compiler is used to schedule parallel operations into the fields of a long instruction [6]. The functional units of a VLIW architecture can be configured to meet the requirements of a target application. Furthermore, the number of fields in a long instruction can be adjusted to match the underlying architecture. Finally, using simple, RISC-like operations to control the different functional units makes it easy for the compiler to generate code for the target architecture. It also enables the compiler to detect and exploit parallelism among these operations, and schedule them into long instructions.

The drawbacks of using a VLIW architecture include its high instruction-bandwidth requirements and low-density instructions. On every clock cycle, a long instruction must be fetched from memory. If memory is stored off-chip, many pins are needed just to interface with memory. This may not be practical or economical because it requires very expensive packaging and increases power consumption. Even if memory is stored on-chip and instruction bandwidth is no longer a problem, the fields of a long instruction may not always be fully utilized. Storing such low-density instructions in memory is very wasteful, and results in increased system costs.

For these reasons, and until very recently, VLIW architectures were considered too impractical and expensive to implement as single-chip processors, especially for embedded applications. However, recent media processors and DSPs, such as the Philips TriMedia [7],[8], Chromatic Mpact [9], IBM Mfast [10],

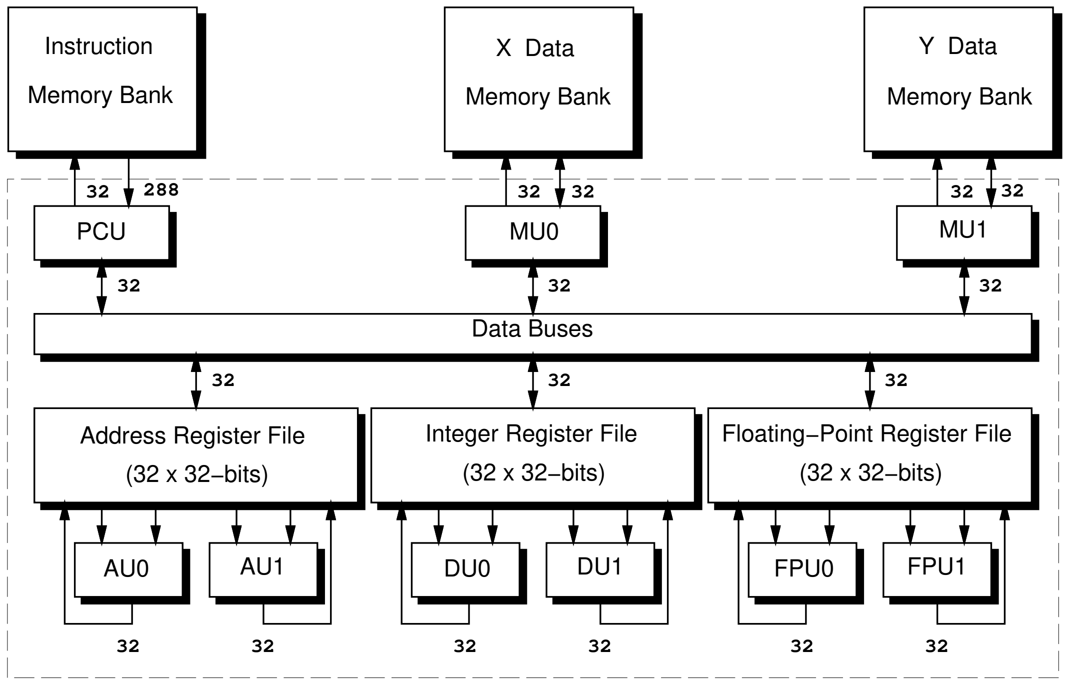


Figure 2.2: VLIW model architecture.

and Texas Instruments TMS320C6201 [11],[12], have all been designed around VLIW architectures. A common characteristic of these processors is that they use techniques to reduce the instruction bandwidth requirements and make more efficient use of memory resources. Chapter 5 examines these techniques in more detail. It also describes a new technique for encoding the long instructions of the UTDSP model architecture, and reducing its instruction bandwidth requirements. This technique is briefly described in Section 2.7.

Figure 2.2 shows the UTDSP model architecture configured with nine functional units. These include two memory units (MU0 and MU1) that execute memory operations, and that are each connected to a single-ported, data-memory bank; two address units (AU0 and AU1) that execute address operations; two integer units (DU0 and DU1) that execute integer operations; two floating-point units (FU0 and FU1) that execute floating-point operations; and a program control unit (PCU) that executes control operations. The PCU also contains a number of control registers that support low-overhead looping, and a system stack that is used to save the program counter and other control registers during a function call, or while executing nested loops. Since the model uses nine functional units, long instructions can specify up to nine parallel operations. UTDSP machine operations are simple and RISC-like, and are described in more detail in Appendix A.

To increase bandwidth and enable the simultaneous fetching of instructions and data, the model is also built around a Harvard memory architecture where instruction memory is separated from data memory. On each clock cycle, a single long-instruction can be fetched from instruction memory. Data memory is divided into two, separate, single-ported banks that can each be accessed by a single memory unit. This increases the bandwidth of data memory and enables simultaneous data accesses to be made without using more costly dual-ported memory. However, to ensure that the data-memory banks are used effectively, program data must be judiciously distributed among them. This task is best handled by the compiler, and is described in Section 2.5.

The model also uses three register files that are used to store address, integer, and floating-point operands, respectively. To facilitate code generation, each register file consists of 32, 32-bit registers. Each register file is also associated with a class of functional units. For example, the integer functional units can only operate on registers in the integer register file, while the floating-point units can only operate on registers in the floating-point register file. Having separate register files simplifies the design of the register files and minimizes the number of read and write ports used. All register files are also connected to the memory units to allow data transfers to any data-memory bank. Moreover, all register files are connected to the program-control unit, which can execute special operations to move and convert data between different register files.

Finally, the model uses a simple, four-stage pipeline whose stages include: instruction fetch, instruction decode and operand fetch, instruction execute, and result write back. All operations, including memory accesses and floating-point operations, are assumed to execute in a single clock cycle, a common attribute of DSP operations. Taken branches incur a latency of one clock cycle.

2.4 C Compiler

The C compiler is used to translate a program written in C into a functionally equivalent program written in the machine language of the target architecture. In doing so, the compiler also tries to enhance the run-time performance of the code by applying standard, machine-independent, scalar optimizations such as loop unrolling, common sub-expression elimination, strength reduction, and constant propagation [13]. The compiler can also enhance run-time performance by applying specific machine-dependent optimizations. These optimizations vary from one architecture to the next, and examples of such optimizations for general-purpose RISC processors include instruction scheduling, software pipelining, register renaming, data prefetching, and branch prediction. One reason for the poor quality of compiler-generated DSP code is the poor, and often non-existent, level of machine-dependent optimizations.

The C compiler for the model architecture was initially based on the GNU C compiler (Gcc) from the Free Software Foundation [14]. The main motivation for using Gcc was that it is public-domain software, it uses a good suite of scalar optimizations, and it is easy to retarget to different architectures. However,

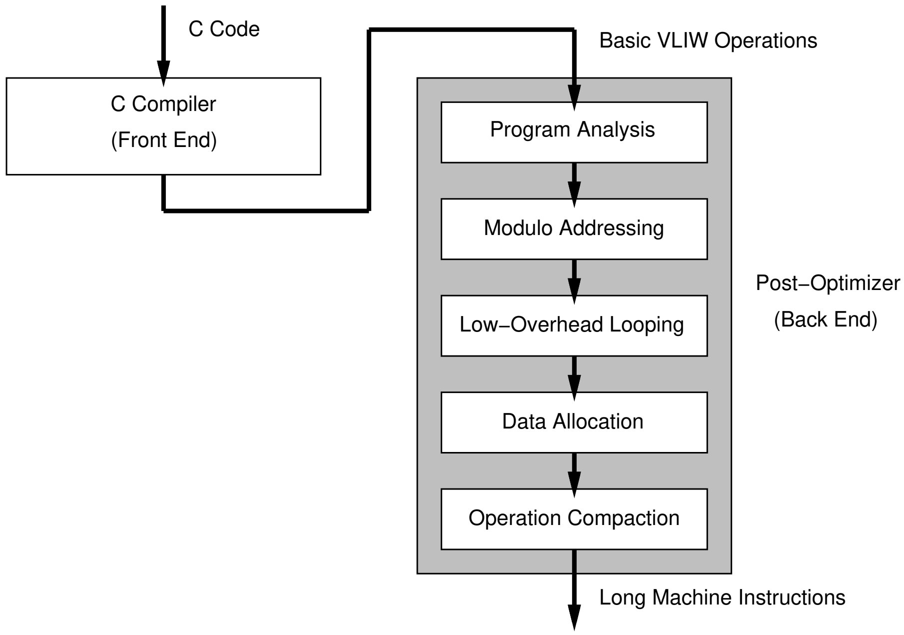


Figure 2.3: Two-phase compilation: C compiler front end and post-optimizer back end.

a drawback of using Gcc is that the register-transfer language (RTL) it uses as an intermediate form does not, on its own, provide enough information about the program to enable the implementation of more sophisticated optimizations.

To enable the implementation of DSP-specific machine optimizations without modifying Gcc, a post-optimizing pass had to be developed. Figure 2.3 shows the resulting two-phase compilation process. In the first phase, the compiler front-end translates C programs into sequential assembly operations that can be executed on the model architecture. In the second phase, the post-optimizer back-end processes the compiler-generated code and optimizes it for execution on the model architecture. The initial work on Gcc and the post-optimizer was done by Vijaya Singh [15]. The post-optimizer was later augmented and enhanced by both the author [16] and Mark Stoodley [17]. Section 2.5 describes the post-optimizer in greater detail.

The difficulty in modifying Gcc and the need for communicating information about the source code to the post-optimizer made it necessary for application programs to be coded in the special styles. For example, when accessing arrays, pointer notation (`ptrs`) had to be used to ensure that the proper addressing operations were generated. Moreover, loops had to be software pipelined (`_SWP`) to expose any parallelism within their bodies to the post-optimizer. To enable the use of a more natural coding style, the Gcc front-end was replaced by the SUIF compiler [18].

SUIF was developed at Stanford University to provide a framework for research in computer architecture and optimizing compilers. It consists of several optimizing and code-analysis passes, and is very easy to modify and augment due to its object-oriented design. SUIF also uses annotations to pass information between different optimizing passes. Such annotations are typically used to pass high-level information about the source code between different optimizing passes, making it possible, for example, to use a more natural coding style. The modular design of SUIF also makes it easy to implement the different DSP-specific optimizations within the SUIF framework. However, due to time limitations, it was not possible to implement all optimizations within SUIF. Instead, it was decided to preserve the post-optimizer and maintain the two-phase compilation process. The SUIF compiler was ported to the model architecture by Sanjay Pujare [19], and replaced Gcc as a compiler front-end. It also enabled the benchmark suite to be rewritten in a more natural style without sacrificing performance [3]. SUIF is currently being used by other researchers developing HLL DSP compilers [20],[21].

2.5 Post-Optimizer

The post-optimizer is used to enhance the execution performance of a program by exploiting the DSP-specific features of the UTDSP model architecture. It processes the sequential machine operations generated by the C compiler and generates long instructions that can execute on the model architecture. Figure 2.3 shows the five passes of the post-optimizer. These include the program analysis pass, the modulo addressing pass, the low-overhead looping pass, the data allocation pass, and the operation compaction pass.

The program analysis pass divides the code generated by the compiler into basic blocks and constructs its control-flow graph. It then analyzes the code and extracts information about its data-flow, control-flow, and aliasing characteristics for use in subsequent optimizing passes.

The modulo addressing pass enables one-dimensional arrays to be accessed in a modulo fashion. This is generally difficult to support in HLL compilers because modulo addressing is not normally specified in the syntax of the C programming language. To overcome this problem, the semantic meaning of arrays in the C language was changed so that, whenever the modulo addressing pass is invoked, all arrays are treated like circular buffers and their elements are accessed in a modulo fashion. This way, regular arrays can still be correctly accessed in a linear fashion, while arrays intended to be used as circular buffers can be addressed in a modulo fashion[1] without explicit testing for the modulus. Although the modulo addressing pass may introduce errors into an otherwise correct program — for example, using the address immediately following the end of an array as the upper bound in a loop, when doing so would actually map the

1. When modulo addressing is used, addressing operations that increment or decrement pointers (`inc`, `dec`) are replaced by special operations that perform modulo increment or decrement (`incmod`, `decmod`).

address to the beginning of the array and prevent the loop from exiting — such errors can be avoided if programmers are made aware of the way modulo addressing is implemented.

The low-overhead looping pass eliminates the overhead operations associated with a loop and replaces them with a single low-overhead looping operation that controls the execution of the loop body. Overhead operations include loop initialization, iteration-count testing, and branching to the beginning of the loop. When these operations are executed in the body of a loop, performance is greatly reduced. To overcome this problem, information about the loop, such as the iteration count and the addresses of the first and last instructions in the loop body, is gathered from the program and used to replace the overhead operations by a special low-overhead looping operation. When this operation is executed, it initializes special control registers that enable the hardware to manage the execution of the loop. By eliminating the overhead operations and managing the execution of the loop body in hardware, performance is improved.

The data allocation pass partitions program data among the data-memory banks of the architecture to expose parallelism among `load` and `store` operations. DSPs use multiple, single-ported data-memory banks, typically two, to increase bandwidth and enable simultaneous fetching of operands from memory. While this provides a cost-effective method of increasing bandwidth, compared to using a dual-ported memory, for example, it requires the data to be allocated to appropriate memory banks. For example, in a loop that computes the sum of products of two arrays, storing the arrays in different memory banks enables two elements from both arrays to be fetched simultaneously. In other words, it enables the exploitation of parallelism in fetching data from memory. On the other hand, storing both arrays in the same memory bank would necessitate fetching both array elements sequentially, and this degrades performance. One of the major reasons for the poor performance of DSP compilers is their inability to exploit multiple data-memory banks automatically. In Chapter 3, two algorithms are presented for automatically exploiting the dual data-memory banks of the model architecture.

Finally, the operation compaction pass exploits the parallelism found in the basic blocks of a program, and packs parallel machine operations into long instructions that can be executed on the target architecture. The algorithm used in the operation compaction pass is called *list scheduling*, and was initially developed for local microcode compaction [22]. By packing parallel operations into long instructions, the operation compaction pass also binds operations to their corresponding functional units.

The inherent flexibility of long instructions enables them to specify a rich combination of parallel operations. In turn, this defines a flexible instruction format that supports the synthesis of new instructions at compile-time to match the available parallelism in a given application. This is in contrast with typical DSP instruction sets that are designed to exploit only limited instances of parallelism commonly found in the inner loops of DSP algorithms. By using a more flexible instruction format, more parallelism can be exploited and better performance can be achieved. However, without an efficient means of encoding long instructions, the performance benefits are offset by the high instruction bandwidth and storage requirements. That is why, upon meeting the performance and cost requirements of a target application, the instruction set should be “frozen”, and long instructions should be efficiently encoded. The efficient encoding of long instructions is the subject of Chapter 5, and will be briefly described in Section 2.7.

2.6 Instruction-Set Simulator

The instruction-set simulator simulates the execution of a program on the model architecture, and generates a report that summarizes its run-time characteristics. Examples of the statistics gathered and reported by the simulator include the number of cycles executed, functional-unit utilization, dynamic operation counts, dynamic instruction counts, addressing modes used, registers used, branch behavior, branch frequencies, average program parallelism, data-memory requirements, and basic-block frequencies. These statistics provide useful information about the run-time behavior of an application. However, within the design framework, the instruction-set simulator is mainly used to measure execution performance. The simulator was initially developed by the author [16], and was later redesigned by Mark Stoodley to make it more extensible and easier to upgrade [17].

2.7 Long Instruction Encoder

The long instruction format of VLIW architectures offers great flexibility in exploiting parallelism, and is ideally suited to matching the performance requirements of a given application to the hardware resources of a target architecture. However, in its basic form, it is too impractical and expensive to support. Fetching long instructions from off-chip memory increases the pin requirements of the chip, results in more expensive packaging, and increases power consumption. Furthermore, since it may not be possible to utilize all operation fields in a long instruction, storing such instructions in their basic form is wasteful of memory.

One way to reduce bandwidth and storage requirements, and still preserve the flexibility of the long-instruction format, is to build an instruction decoder around a memory bank in a manner similar to the writable control stores of microprogrammed computers. In a microprogrammed computer, part of the instruction word typically consists of a pointer to a micro-routine or to horizontal microcode that is stored in the control store. In a similar fashion, long instructions can be encoded as pointers by the compiler, and stored in instruction memory. Once a pointer is fetched from instruction memory, it can be used to fetch the corresponding long instruction from the decoder memory bank.

Figure 2.4 shows a simplified diagram of the instruction decoder. Instruction memory is used to store single machine operations or pointers to long instructions. Machine operations stored in instruction mem-

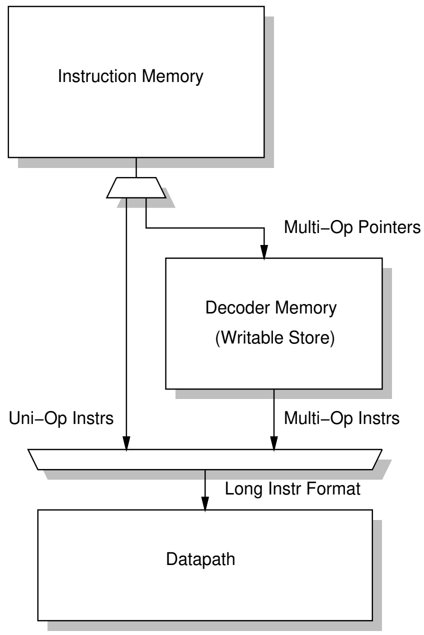


Figure 2.4: Basic structure of instruction decoder.

ory represent *uni-op* instructions, or long instructions that specify a single operation only. Since storing and fetching a single operation is easy, uni-op instructions do not require any special encoding. On the other hand, pointers stored in instruction memory are used to represent *multi-op* instructions, or long instructions that specify multiple operations. The actual instructions are stored in the decoder memory where they are packed in an efficient manner to avoid wasting decoder memory space.

When a word is fetched from instruction memory, it is first examined to determine whether it contains a single operation or a pointer to a multi-op instruction. If the word contains a single operation, it is forwarded to the appropriate functional unit where it is further decoded and executed. On the other hand, if the word contains a pointer, it is used to fetch the corresponding multi-op instruction from the decoder memory bank. The multi-op instruction is then presented to the datapath in a long-instruction format, and its different operations are decoded and executed by the corresponding functional units.

The ability to write into the decoder memory ensures that flexibility is maintained. Depending on the performance requirements of the application, specific multi-op instructions may be stored in the decoder memory. Thus, for different applications, a pointer encoding will be used to represent different multi-op instructions. Moreover, for a program to execute correctly, the contents of the decoder memory must be correctly set. Although this makes software compatibility difficult, this is not usually a problem in embedded systems, since the processor will typically be used to execute a single application, or a fixed set of applications. The decoder memory can be implemented either as a ROM, where it is programmed once at the factory, or as a RAM, where it can be loaded immediately before executing a program. Chapter 5

Datapath Components	Bit-Area Estimates (μm^2/bit)
Integer ALU	$2.30 \times 10^4 \times f^2$
Integer multiplier/divider	$2.03 \times 10^5 \times f^2$
Floating-point adder	$8.94 \times 10^4 \times f^2$
Floating-point multiplier	$8.28 \times 10^3 \times f^2$
Floating-point divide/square-root circuit	$1.95 \times 10^5 \times f^2$
Single-ported register cell	$3.96 \times 10^2 \times f^2$
Memory cell	$3.65 \times 10^2 \times f^2$
Register cell area increase per extra port	50%

Table 2.3: Technology-independent constants for different datapath components.

describes the encoding and decoding of long instructions in greater detail. It also discusses their impact on performance and cost.

2.8 Cost Estimation and the Datapath Area Model

While performance can easily be measured by execution time, cost is more difficult to quantify and measure. Since cost is an important consideration in the design of embedded systems, it is important that a system designer be able to evaluate the impact of modifying the architecture on cost, and make more informed trade-offs with performance. To evaluate the cost associated with the model architecture, a first-order model that equates cost with the area occupied by the datapath and memory was developed. Although other researchers have used a similar cost model [29], it was only used to estimate the area of the datapath without any memory. The details of the model were also not described.

Using die micro-photographs of different commercial and research processors [30],[31],[32],[33], technology-independent bit-area estimates — estimates of the area occupied by a single bit in different datapath components — were derived. The datapath components included integer ALUs, integer multiplier/divider circuits, floating-points adders, floating-point multipliers, floating-point divide/square-root circuits, register cells, and memory cells. Table 2.3 shows the bit-area estimates for different datapath circuits expressed in terms of the technology-independent factor, *f*. Expressing the bit-area estimates in terms of *f* makes it easy to scale the area estimates with the line size. Appendix B explains how the different bit-area estimates were derived. For the register cells, it was assumed that adding a write port would increase the register cell area by 50%.

To describe the overall structure of the datapath, a number of architectural parameters were also defined. The following describes the different parameters in more detail.

- **Data size, in bits**. This specifies the bit-width of all data operands, storage elements, and arithmetic and control circuits.
- **Operation length, in bits**. This specifies the length of a machine operation or an encoded multi-op instruction.
- **Number of program-control units**. In addition to the area occupied by the system stack and other control registers, a program-control unit is assumed to occupy the same area as an integer ALU.
- **System stack depth**. This represents the number of elements in the system stack.
- **Number of memory units**. Since each memory unit is connected to a single-ported data-memory bank, this is also the **number of data-memory banks**.
- **Number of words per data-memory bank**. Each word has a length specified by the **data size** parameter.
- **Number of words in instruction memory**. This includes the words used in the decoder memory. Each word has a length specified by the **operation length** parameter.
- **Number of address units**. An address unit is assumed to occupy the same area as an integer ALU.
- **Number of address registers**. All registers have a length specified by the **data size** parameter. Address registers also have R_a read ports and W_a write ports, where $R_a = 3 \times AUs + 2 \times MUs + 1 \times PCUs$ and $W_a = 1 \times AUs + 1 \times MUs + 1 \times PCUs$. The variables AUs, MUs, and $PCUs$ represent the number of address units, memory units, and program-control units, respectively.
- **Number of integer units**.
- **Number of integer multiplier/divider circuits**. This cannot exceed the number of integer units.
- **Number of integer registers**. Integer registers have R_i read ports and W_i write ports, where $R_i = 3 \times DUs + 1 \times MUs + 1 \times PCUs$ and $W_i = 1 \times DUs + 1 \times MUs + 1 \times PCUs$. The variables DUs, MUs, and $PCUs$ represent the number of integer units, memory units, and program-control units, respectively.
- **Number of floating-point units**.
- **Number of floating-point multipliers**. This cannot exceed the number of floating-point units.
- **Number of floating-point divide/square root circuits**. This cannot exceed the number of floating-point units.
- **Number of floating-point registers**. Floating-point registers have R_f read ports and W_f write ports, where $R_f = 3 \times FUs + 1 \times MUs + 1 \times PCUs$ and $W_f = 1 \times FUs + 1 \times MUs + 1 \times PCUs$. The variables FUs, MUs, and $PCUs$ represent the number of floating-point units, memory units, and program-control units, respectively.

Cost Model Parameters	Values
Data Size (bits)	32
Operation Length (bits)	32
Program Control Units	1
System Stack Depth	32
Memory Units	2
Words Per Data-Memory Bank	2048
Words in Instruction Memory	4096
Address ALUs	2
Address Registers	32
Integer ALUs	2
Integer Mul/Div Circuits	2
Integer Registers	32
Floating-Point Adders	2
Floating-Point Multipliers	2
Floating-Point Div/Sqrt Circuit	2
Floating-Point Registers	32

Datapath Components	λ = 0.25 µm	
	Area (mm^2)	Percentage Total Area
Program-Control Unit	0.17	41%
Address Units	0.09	
Integer Units	1.93	
Floating-Point Units	2.20	
Data Memory	2.99	56%
Instruction Memory	2.99	
Address Register File	0.09	3%
Integer Register File	0.08	
FP Register File	0.08	
Total Area	10.62	100%

Table 2.4: Default cost-model parameters and area of different datapath components. Area estimates are based on λ = 0.25 µm.

By setting the architectural parameters and using the technology-independent constants, the area of individual datapath components can be estimated. By adding these, an estimate of the total datapath area can be obtained. However, this cost model is by no means accurate or comprehensive. For example, it does not account for the area occupied by internal busses. Nonetheless, it is very useful in comparing the relative cost of different datapaths having different components.

Table 2.4 shows how the default values of the architectural parameters are set to describe the datapath of the model architecture. Although these parameters reflect the UTDSP architecture, they can be modified as needed to model different architectural configurations. Table 2.4 also shows the area estimates of individual datapath components, and the area estimate of the entire datapath. Assuming a line size, λ, of 0.25 µm, the area occupied by the model architecture in Figure 2.2 is approximately 11 mm^2. In this case, the functional units occupy 41% of the area, while memory and the registers occupy 56% and 3% of the area, respectively.

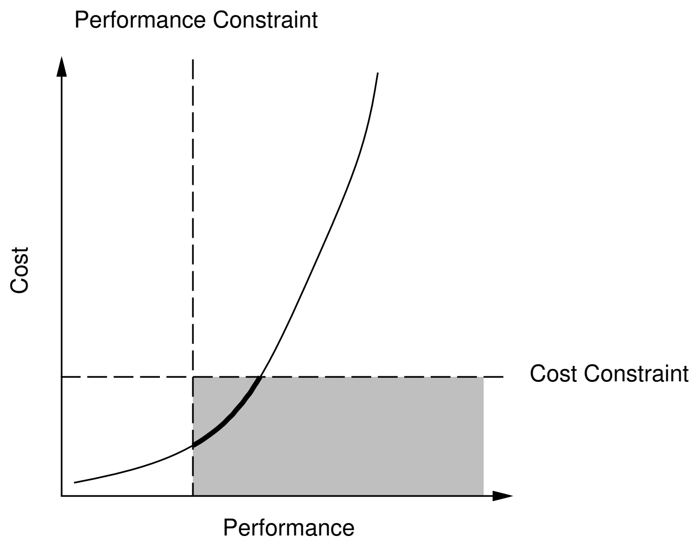


Figure 2.5: Generic cost vs. performance graph.

2.9 Performance and Cost Evaluation

The ability to evaluate performance and cost is necessary for ensuring that an architecture meets its design specifications. Typically, performance is specified by the real-time constraints of a target application, while cost is specified by the area budget of the datapath. In the application-driven design framework, performance is measured by the cycle counts obtained from the instruction-set simulator, while datapath area is estimated using the cost model. Using this information, a system designer can decide how best to configure the architecture, the compiler, or both, to achieve the desired performance and cost objectives.

Figure 2.5 shows a generic plot of cost versus performance for a given application. Each point on the curve corresponds to a pair of *(performance, cost)* coordinates associated with an architectural configuration of the datapath. The Figure also shows the performance and cost constraints of the application as horizontal and vertical dotted lines. For the architecture to meet these constraints, its *(performance, cost)* coordinates must fall within the shaded area shown on the graph. Thus, the part of the curve that falls within this area, shown in bold, corresponds to different architectural configurations that meet both performance and cost constraints.

Once the *(performance, cost)* coordinates of a given architectural configuration are known, the designer can modify the architecture to reach a better point on the *(performance, cost)* curve. For example, if performance requirements are not met, the designer can add more hardware resources, such as functional units, memory, or registers, to the architecture. Alternatively, the designer can improve the quality of the generated code by invoking more compiler optimizations or exploiting more parallelism. On the other hand, if cost requirements are not met, the designer can remove functional units that are under-utilized, or reduce the amount of memory or the number of registers used. This process is repeated until the most suitable design point is reached.

2.10 Summary

This chapter described the methodology and tools used in designing the UTDSP application-specific programmable processor. The design methodology is application-driven: an application, or set of applications, is compiled to a flexible target architecture, and its execution performance and cost are measured. The results are compared to application-specific performance and cost constraints, and the architecture is iteratively tuned until it meets these constraints.

To validate the design methodology and assess the impact of different design tools on performance and cost, a suite of DSP benchmarks, consisting of twelve kernels and ten applications, are used. All benchmarks are coded in C.

The target architecture is based on a very long instruction word (VLIW) architecture. This architecture is very flexible and can easily be configured to match the functional requirements of an application and exploit parallelism. Its long-instruction format is also an easy target for HLL compilers, and facilitates the synthesis of instructions that can match the performance requirements of an application. However, a drawback of the long-instruction format is its high instruction bandwidth and storage requirements.

To overcome these problems without affecting flexibility, long instructions with multiple operations are tightly packed into a special memory bank that is part of the instruction decoder. Long instructions are also encoded as pointers into the special decoder memory. By storing and fetching these pointers from instruction memory, storage and bandwidth requirements are reduced. Once a pointer is fetched, it is used to fetch the corresponding long instruction from decoder memory. Because decoder memory can easily be written, a pointer can represent virtually any long instruction, and this preserves flexibility.

To compile an application, a two-phase process is used. During the first phase, a C compiler is used to translate the source code into basic machine operations. It also applies scalar optimizations to enhance the quality of the generated code. During the second phase, a post-optimizer is used to exploit the DSP-specific features of the target architecture. The post-optimizer first analyzes the code, then applies a number of architecture-specific optimizations. It also exploits parallelism among machine operations and schedules them into long instructions. Finally, long instructions are encoded as pointers to minimize memory storage and bandwidth requirements.

To simulate the execution of an application on the target architecture, an instruction-set simulator is used. The simulator also gathers information about execution time, and is therefore used to measure performance. To estimate the area occupied by the datapath of the target architecture, a cost model is used.

By measuring the performance and cost of the target architecture, a system designer can reconfigure the architecture to better match an application. For example, if performance requirements are not met, the designer can add more hardware resources to the architecture. Alternatively, the designer can improve the quality of the generated code by invoking more compiler optimizations or exploiting more parallelism. On the other hand, if cost requirements are not met, the designer can remove functional units that are under-utilized or reduce the amount of memory or the number of registers used. This process is repeated until the performance and cost requirements of the application are met.

In the next chapter, one particular compiler optimization is examined. This optimization was developed to handle the automatic partitioning of data across two, independent, data-memory banks.

Chapter 3

Exploiting Dual Data-Memory Banks

A common feature of programmable DSPs is their use of multiple memory banks. Typically, DSPs use the Harvard memory architecture, where instruction memory is separated from data memory, to enable the simultaneous fetching of instructions and data. Furthermore, DSPs commonly use two separate data memory banks to double the bandwidth of the data-memory system and enable two simultaneous data fetches. When coupled with high-order data interleaving[1], dual data-memory banks provide the same bandwidth as more costly memory organizations such as a dual-ported memory. However, while exploiting separate memories for instructions and data is straightforward, compilers for DSPs rarely exploit the ability to make two data accesses simultaneously without special annotations or additions to the programming language [34].

This chapter describes two algorithms — compaction-based (CB) data partitioning and partial data duplication — that were developed to help high-level language (HLL) DSP compilers exploit dual data-memory banks. Both algorithms are implemented in the data allocation pass of the UTDSP compiler's post-optimizing back-end. The results show that CB partitioning is an effective technique for exploiting dual data-memory banks, and that partial data duplication augments CB partitioning by improving execution performance in some cases. For the kernel benchmarks, CB partitioning improves performance by 25% – 70%. For the application benchmarks, it improves performance by 2% – 19%. For one application, partial data duplication boosts performance from 4%, using CB partitioning only, to 22%.

The chapter is organized into seven sections. Section 3.1 describes the dual data-memory bank problem and outlines a possible solution. Section 3.2 then describes a number of previous and current approaches for exploiting dual or multiple data-memory banks. To provide a historical context, Section 3.3 briefly describes the maximal independent set (MIS) data partitioning algorithm, which preceded the CB parti-

1. In high-order data interleaving, consecutive address locations are stored in the same memory bank. This is in contrast with low-order data interleaving, where consecutive address locations are stored in separate memory banks.

```
sum = 0;
for (i = 0; i < N; i++)
   sum += A[i] * B[i];
```

(a)

```
[1]   CLR   A          X:(R0)+,X0    Y:(R4)+,Y0
[2]   REP   #N-1
[3]   MAC   X0,Y0,A    X:(R0)+,X0    Y:(R4)+,Y0
[4]   MACR  X0,Y0,A
```

(b)

Figure 3.1: (a) C-language implementation of an N-th order FIR filter (b) Motorola DSP56001 assembly language implementation of the same filter.

tioning algorithm, and discusses its limitations. Next, Section 3.4 describes the compaction-based data partitioning algorithm in detail, while Section 3.5 introduces partial data duplication as a means of augmenting CB partitioning. Section 3.6 then evaluates the impact of CB partitioning and partial data duplication on performance and cost. Finally, Section 3.7 summarizes the main ideas and results of this chapter.

3.1 The Dual Data-Memory Bank Problem

One reason why dual data-memory banks and high-order data interleaving are used in DSPs is that most memory accesses that can occur in parallel in signal processing algorithms are to elements of different arrays. This can best be illustrated by considering the software implementation of an Nth-order FIR filter, a common algorithm in digital signal processing.

Figure 3.1(a) shows the C-language implementation of the filter, while Figure 3.1(b) shows the corresponding code in the assembly language of the Motorola DSP56001 [23]. In addition to illustrating how dual, high-order interleaved memory banks can be used, Figure 3.1(b) shows how other specialized DSP hardware features are used.

The assembly version is a simple example of software pipelining whereby elements of arrays A and B are pre-loaded in the iteration before the one in which the elements are actually used. This is done to expose as much parallelism as possible. Instruction `[2]` is a special *repeat* instruction that causes the subsequent instruction to be executed N – 1 times.

The DSP56001 is capable of executing several independent operations per instruction in a fashion similar to a VLIW architecture. However, the types of operations that can be combined to form an instruction are more limited in the DSP56001. In Figure 3.1(b), instructions `[1]`, `[3]`, and `[4]` are examples of such restricted VLIW instructions. Instruction `[1]` executes five operations: the accumulator is cleared (`CLR A`); register X0 is loaded from X memory and address register R0 is post-incremented (`X:(R0)+,X0`); and register Y0 is loaded from Y memory and address register R4 is post-incremented (`Y:(R4)+,Y0`). Instruction `[3]` executes a multiply-accumulate operation (`MAC`) and two memory loads from X and Y memory into the X0 and Y0 registers; it also post-increments address registers R0 and R4. Finally, instruction `[4]` executes a multiply-accumulate-and-round (`MACR`) operation.

Instructions `[1]` and `[3]` demonstrate the effective use of the dual memory banks. By allocating array A to one memory bank and array B to the other, it is possible to access an element from each array in one instruction. If the arrays are both allocated to the same memory bank, the elements would have to be accessed sequentially with two instructions. This increases the size of the loop from one instruction to two instructions, and reduces performance by a factor of two. Thus, by treating each array as a single entity and allocating the arrays to different banks, parallel accesses to all elements of the two arrays are guaranteed.

A straightforward technique that uses dual memory banks efficiently with respect to performance is to duplicate all data. By doing so, there would no longer be any need for an allocation algorithm. Duplicating data, however, also increases the memory cost of an embedded system due not only to the duplicated data but also to additional store operations that would be needed to maintain the integrity of both copies of the data. Thus, there is no benefit in duplicating all data indiscriminately when comparable performance could be obtained using far less memory by judiciously allocating arrays to the appropriate memory banks. However, as Section 3.5 demonstrates, there are programs for which the only software solution to providing parallel accesses is to duplicate some of the data. In such situations, the gain in performance must be weighed against the increase in memory cost.

Ideally, program data should be allocated so that simultaneous memory accesses are possible without having to resort to data duplication. This can be done by partitioning the data into two sets, one for each memory bank. Partitioning data, in turn, requires addressing three interrelated issues. First, partitioning relationships must be established among different data. This involves identifying instances when pairs of data should be stored in separate memory banks. Second, a suitable cost metric that can be assigned to every partitioning relationship must be defined. This helps quantify the importance of satisfying a particular relationship and reflects its impact on performance. Finally, rules must be established for partitioning the data in a manner that satisfies as many partitioning relationships as possible and yields the best performance. Before describing the algorithms for partitioning data, Section 3.2 describes previous and current approaches to exploiting dual or multiple data-memory banks.

3.2 Previous and Related Work

Traditionally, the exploitation of dual data-memory banks in DSPs has been the responsibility of the programmer. Whether programming in assembly language or in a high-level language, the programmer has to allocate data manually by using assembler directives or compiler pragmas [34]. This makes it very difficult to ensure an efficient exploitation of dual memory banks, especially in large embedded applications. The author is unaware of any commercial DSP compilers that automatically exploit dual data-memory banks.

In the academic community, a group of researchers at Princeton University have been investigating techniques for exploiting multiple memory banks that are available on DSPs such as the Motorola DSP56001 and the NEC77016 [35]. Such processors place constraints on which registers can be used to store data from a particular data bank, thus making register allocation and data allocation interrelated problems. Due to the larger dimensions of the solution space, the researchers resorted to a computationally-intensive, simulated annealing algorithm that labels a constraint graph whose nodes represent symbolic register and memory references, and whose edges represent dependences and constraints among those references. The researchers also considered a simpler greedy algorithm that processes the variables in the order that they are used and allocates them to the memory banks in an alternating manner. Interestingly, the results of their study show that the performance gained from the simulated annealing algorithm is comparable to that gained from the greedy algorithm, thus suggesting that the problems of register allocation and data partitioning could be decoupled without any loss in performance. By contrast, the UTDSP architecture places no restrictions on the usage of registers. This makes it easier to exploit the orthogonality between register allocation and data allocation to arrive at a simpler solution that produces good results.

Another approach to generating DSP code is to rely on code synthesis tools such as the Ptolemy package developed at the University of California at Berkeley [36]. Ptolemy is an environment for simulation, prototyping, and software synthesis of heterogeneous systems. It enables designers to specify embedded applications in the form of hierarchical dataflow graphs of functional blocks. A block may be implemented either as a hand-optimized assembly-language routine or as a high-level language routine. To manage the interaction between different blocks, Ptolemy uses dataflow scheduling techniques, and synthesizes code that effectively stitches the different block implementations together. To simplify the interaction between different block implementations, the exploitation of dual memory banks is kept to a minimum, especially for data that is shared among different blocks. The resulting code may therefore not take full advantage of dual data-memory banks.

In a somewhat different context, low-order interleaved memory banks are used in the Multiflow Trace 7/300 VLIW computer and a memory bank disambiguator is used by the Multiflow compiler to identify the memory banks that memory operations access [37]. This enables the compiler to identify instances where different memory operations can occur in parallel. Data accesses that cannot be disambiguated at compile time are issued to a slower, central memory controller. Using a memory bank disambiguator is a different approach to the problem of using multiple data banks effectively. In contrast, the CB partitioning algorithm determines to which memory bank data should be allocated to increase the opportunity for generating parallel memory accesses. Since the algorithm determines where data are stored, there is no need for a memory bank disambiguator. Instead, the algorithm uses alias information to determine which data a memory operation accesses. Alias analysis is simplified by the use of special addressing operations and dedicated address registers in the UTDSP architecture.

3.3 Maximal Independent Set Data Partitioning

The CB partitioning algorithm has its origins in the maximal independent set (MIS) data partitioning algorithm that was also developed to exploit the dual data-memory banks of the UTDSP architecture [16],[38]. Both algorithms are based on partitioning the nodes of an undirected interference graph used to represent partitioning relationships among program data. The nodes of this graph represent the variables and arrays of a program. In both algorithms, an array is treated as a monolithic entity that is allocated in its entirety to a single memory bank. This restriction is a direct consequence of the fact that the two memory banks are high-order interleaved. An edge connecting a pair of nodes in the graph indicates that the corresponding variables may be accessed in parallel and that those variables should be stored in separate memory banks to allow parallel access to actually occur. A weight, equal to the loop nesting depth of the memory operations used to access the data, is also assigned to each edge to represent the degradation in performance if the corresponding variables are not accessed in parallel. This weight was chosen as a heuristic measure to ensure that the highest priority is given to exploiting load/store parallelism inside inner program loops. Although other, more accurate measures, such as profile-driven and programmer-supplied weights, may also be used, the results in Section 3.6.1 show that using such edge-weights does not always improve performance. Instead, other factors, discussed in Section 3.5, can also result in poor performance. Both algorithms use these weights to compute a minimum-cost data partitioning that will result in the least degradation in performance. After constructing the interference graph, the nodes are partitioned into two sets and the corresponding variables are allocated to different memory banks. All load and store operations in the program are then assigned to an appropriate memory-access unit based on the memory bank assignment of the data they access. These assignments are later used by the operation-compaction pass of the compiler when determining which operations to schedule into a long instruction.

The MIS data partitioning algorithm was developed to exploit the parallelism in load operations that fetch operands for arithmetic operations embedded within program loops. For every pair of variables corresponding to such operands, an edge is added to connect the matching nodes in the interference graph. The weight assigned to each edge in the graph gives higher importance to exploiting parallelism in accessing operands from inner loops.

Once the graph is constructed, the nodes are partitioned into two sets: a MIS on the nodes of the graph and its corresponding complement set. By definition, a MIS is any subset of nodes such that no two nodes in the subset are edge-connected, and such that each node not in the subset is edge-connected to at least one node in the subset [39]. The complement set consists of those nodes that are not in the MIS. Since the nodes of the MIS are, by definition, not edge-connected, their corresponding variables need not be fetched in parallel, and could therefore be allocated to the same memory bank. The variables corresponding to the nodes of the complement set are allocated to the other memory bank. However, because the nodes of the complement set may generally be edge-connected, allocating the corresponding variables to the same memory bank can degrade performance. To overcome this problem, the algorithm attempts to find a minimum-cost partitioning. Since an MIS is not unique, the algorithm computes all possible MIS's, and selects the partitioning that incurs the least cost. The cost of a partitioning is the sum of the weights of the edges connecting the nodes of the complement set.

A major limitation of this algorithm is that it only attempts to expose the parallelism in load operations that fetch operands for arithmetic operations. This causes it to miss out on exploiting other instances of parallelism among other load and store operations and yielding better overall performance. However, the main limitation of this algorithm is that it does not ensure an optimal data partitioning that exposes the most parallelism or yields the most speedup in execution time. This is due to partitioning being based on selecting a minimum-cost solution among a class of partitionings having the specific characteristic of being a maximal independent set, instead of finding a solution that is minimum-cost across all possible partitions. Although this latter problem is provably NP-complete [40], heuristics exist for generating near-optimal solutions [41].

3.4 Compaction-Based Data Partitioning

The compaction-based data partitioning algorithm was developed to overcome the limitations of the MIS partitioning algorithm. In particular, the construction of the interference graph was modified to maximize the exploitation of parallelism in accessing program variables. A greedy algorithm for partitioning the nodes of the graph was also introduced to ensure that a minimum-cost partitioning is found across all possible partitions. In the following subsections, the steps comprising the CB partitioning algorithm will be described in more detail.

3.4.1 Step 1: Exploiting Parallelism in Accessing the Stack

In addition to the parallelism in accessing program variables, there is also parallelism in saving and restoring registers on the stack when entering and exiting functions. However, exploiting this parallelism is made more challenging by the fact that the stack frame must now be distributed across two physical memory banks. Before discussing how to distribute the stack frame and exploit this parallelism, it is first useful to examine the stack model of the SUIF compiler and the way it handles stack references.

Figure 3.2(a) shows a typical example of the code generated by the SUIF compiler to handle the stack, and Figure 3.2(b) shows the structure of a typical stack frame from the perspective of the SUIF code generator. The frame is divided into three areas that are used to store local variables, save registers, and pass parameters, respectively. Here, it should be noted that the code for passing parameters on the stack is handled in the SUIF-based front-end, and not in the post-optimizing back-end.

At the beginning of a function, SUIF generates code to set up the stack frame. Operation `[1]` initializes a register with the size of the frame and operation `[2]` decrements the stack pointer, `a63`, by the appropriate amount. Thus, in accessing any stack location, the appropriate offset must first be added to the contents of the stack pointer.

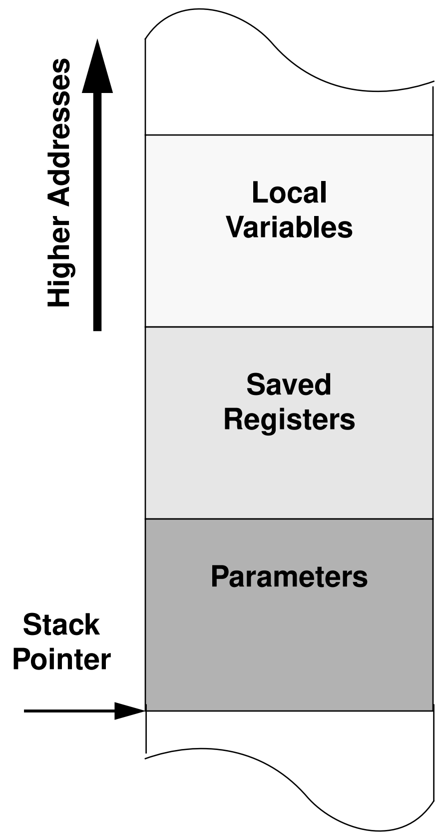
After setting up the stack frame, SUIF generates code to save registers on the stack. Operation `[3]` initializes a register with the appropriate offset, and operation `[4]` adds the offset to the stack pointer. The resulting address points to the top of the area in the frame used to store registers. This address is stored in register `a1` which is used as a pointer. Operation `[5]` decrements the pointer — `a0` is hardwired to 4 — and operation `[6]` stores a register at the corresponding location on the stack. For every register stored on the stack, a similar pair of operations is generated. Operations `[7]` and `[8]` are an example of such a pair.

Within the body of a function, local variables that are stored on the stack are accessed by adding the appropriate offsets to the stack pointer. Operations `[9]–[11]` and `[12]–[14]` demonstrate how such local variables are accessed. In operations `[9]` and `[12]`, a register is initialized with the appropriate offset. In operations `[10]` and `[13]`, the offset is added to the stack pointer and the result is used as a pointer to the appropriate stack location. Finally, in operations `[11]` and `[14]`, the appropriate stack location is used to store or load a register.

Although not shown in this example, accessing parameters passed from a calling function is handled in a similar fashion. Since these parameters would be stored at the stack locations immediately following the

```
[1]   movi.a   #frame_size,a1
[2]   dec      a63,a1,a63
[3]   movi.a   #SR_top,a1
[4]   inc      a63,a1,a1
[5]   dec      a1,a0,a1
[6]   store    (a1),register1
[7]   dec      a1,a0,a1
[8]   store    (a1),register2
               .
               .
[9]   movi.a   #local_offset1,a1
[10]  inc      a63,a1,a2
[11]  load     (a2),register3
               .
[12]  movi.a   #local_offset2,a1
[13]  inc      a63,a1,a3
[14]  store    (a3),register4
               .
               .
[15]  movi.a   #SR_top,a1
[16]  inc      a63,a1,a1
[17]  dec      a1,a0,a1
[18]  load     (a1),register1
[19]  dec      a1,a0,a1
[20]  load     (a1),register2
[21]  movi.a   #frame_size,a1
[22]  inc      a63,a1,a63
```

(a)



(b)

Figure 3.2: (a) UTDSP operations used to handle stack frame. (b) SUIF stack model.

local variables (at higher address locations), accessing parameters only requires that the correct offsets be used. In fact, the only difference between accessing a local variable and a parameter on the stack is that the latter uses an offset that is greater than the size of the stack frame.

At the end of a function, and immediately before the code that tears the stack frame down, SUIF generates code to restore the values of registers stored on the stack. These operations are similar to operations `[3]–[8]` but perform the reverse function. Like operations `[3]` and `[4]`, operations `[15]` and `[16]` initialize a register to the top of the area where registers are stored. Similarly, like the pair of operations that store a register on the stack, operations `[17]` and `[18]` restore a register to its old value. Operation `[17]` decrements the pointer, and operation `[18]` loads a register with its old value from the corresponding stack location. For every register stored on the stack, a similar pair of operations are generated. Operations `[19]` and `[20]` are an example of such a pair.

Finally, at the end of each function, SUIF generates code similar to operations `[21]` and `[22]` to tear the stack frame down. Again, these operations are similar to operations `[1]` and `[2]` but perform the reverse function. Operation `[21]` initializes a register with the size of the frame, and operation `[22]` increments the stack pointer by that amount, effectively releasing that portion of the stack.

Just as exploiting parallelism in accessing program data requires that the data be partitioned among the dual data-memory banks, exploiting parallelism in accessing the stack requires that the stack frame be partitioned and distributed across both data-memory banks. For local variables and parameters that are stored and passed on the stack, each can be represented by a node in the interference graph, and is partitioned, along with other program data, by the greedy algorithm. To prevent any differences in the partitioning of parameters from occuring between caller and callee functions, all functions are assumed to be stored in a single source file. This enables the same interference graph to be used for partitioning local variables and parameters from different functions in a consistant manner. Sections 3.4.2 and 3.4.3 describe the construction and partitioning of the interference graph in more detail. On the other hand, elements of the frame used to save registers at the beginning of a function are partitioned in a mechanical fashion that alternates between both data banks. To exploit the parallelism in accessing these stack elements, successive save/restore operations are assigned to alternating memory units. This will be shown in Figure 3.6(a), where successive register-save operations (operations `[11]` and `[12]`, corresponding to operations `[6]` and `[8]` in Figure 3.2(a)) and their corresponding register-restore operations are assigned to different memory units. The actual distribution of the stack frame, which involves modifying the code that sets up the frame and tears it down, and re-mapping frame offsets, occurs after the program data has been partitioned. Section 3.4.4 explains this in more detail.

3.4.2 Step 2: Building the Interference Graph

Building the interference graph involves identifying its nodes and deciding when to add an edge between a pair of nodes. Identifying the nodes of the graph is straightforward. The SUIF compiler generates assembler directives to specify global variables, and these directives can be used to identify all global variables. SUIF also generates a fixed pattern of operations to access local variables and parameters on the stack — see, for example, operations `[9]–[11]` and `[12]–[14]` in Figure 3.2(a). By identifying such patterns and examining the offsets used, local variables and parameters can also be identified.

Adding edges to the interference graph is done by augmenting the operation-compaction algorithm of the compiler to identify all pairs of memory operations in a basic block that can execute in parallel. The compaction algorithm is based on the list scheduling algorithm used in local microcode compaction [22]. Figure 3.3 shows the pseudo code for the algorithm; italics are used to show the portions of the algorithm used by the data allocation pass to build the interference graph. Note that this version of the compaction algorithm is used in the allocation pass to construct the interference graph, and that operations are not actually scheduled in long instructions until the compaction pass. Since the compaction algorithm is used to construct the interference graph, this algorithm is called *compaction-based* data partitioning.

The compaction algorithm is local in its scope, and is therefore applied to every basic block. A data-dependence graph is first generated to determine the order in which operations must be scheduled. A priority, equal to the number of descendents an operation has in the dependence graph, is also assigned to every operation to facilitate operation scheduling. The data-ready set (DRS), which contains all operations that are ready for scheduling at any given time, is then calculated. The main loop of the algorithm iterates until all operations in the block are scheduled and a new DRS can no longer be calculated.

During each iteration, the DRS is first sorted according to the priority values of its operations. A new, empty, long instruction is then formed to schedule as many operations from the DRS as possible. Each operation in the DRS is checked for data- and function-unit-compatibility with already scheduled operations. Checking for data-compatibility helps generate tighter code by allowing an operation to be scheduled if it has an anti-dependency with another operation that is already scheduled. On the other hand, checking for function-unit-compatibility ensures that no hardware resource conflicts will occur. If an operation is both data- and function-unit-compatible, it is added to the instruction and marked as being scheduled. Once all operations in the DRS have been processed, a new DRS is calculated and another iteration is executed.

To add edges to the interference graph, the compaction algorithm was augmented with some additional processing of memory operations. The first memory operation in the DRS can be both data-compatible and

```
for (every basic block)
   generate data-dependence graph;
   assign priority to each operation;
   calculate data-ready set (DRS);
   while (DRS not empty)
      sort DRS by priority;
      form new long instruction;
      for (all ops in DRS)
         if (op data-compatible)
            if (op function-unit compatible)
               add op to instruction and mark it as scheduled;
               if (op == ld/st)
                  ld/st_i = op;
               end if
            else if (op == ld/st)
               ld/st_j = op;
               if (ld/st_i and ld/st_j access different vars/arrays)
                  add edge to graph;
               else
                  mark var/array for duplication;
               end if
            end if
         end if
      end for
      calculate new DRS;
   end while
end for
```

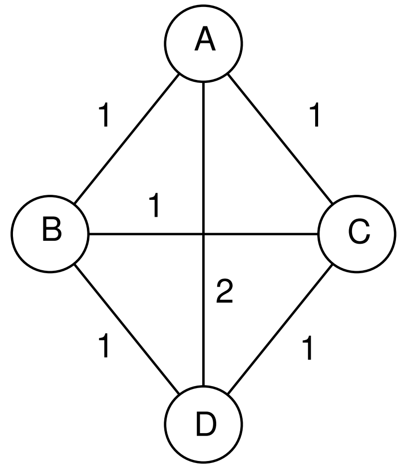
Figure 3.3: Pseudo-code for building the interference graph.

function-unit-compatible. In this case, it is scheduled in the instruction and is also saved in case another memory operation is encountered while processing the current DRS.

Subsequent memory operations in the DRS can also be data-compatible, but cannot be function-unit-compatible. In this case, a memory operation is independent of the operations already scheduled in the instruction, including the first memory operation, but uses the same memory unit as the already-scheduled

```
D[i] = A[j] + B[k]
B[i] = B[j] - D[k]
C[i] = B[j] - C[k]
C[i] = A[j] + C[k]
for (i = 0; i < 5; i++) {
   .
   C[i] = A[i] + D[i]
   .
}
A[i] = C[j] + D[k]
```

(a)



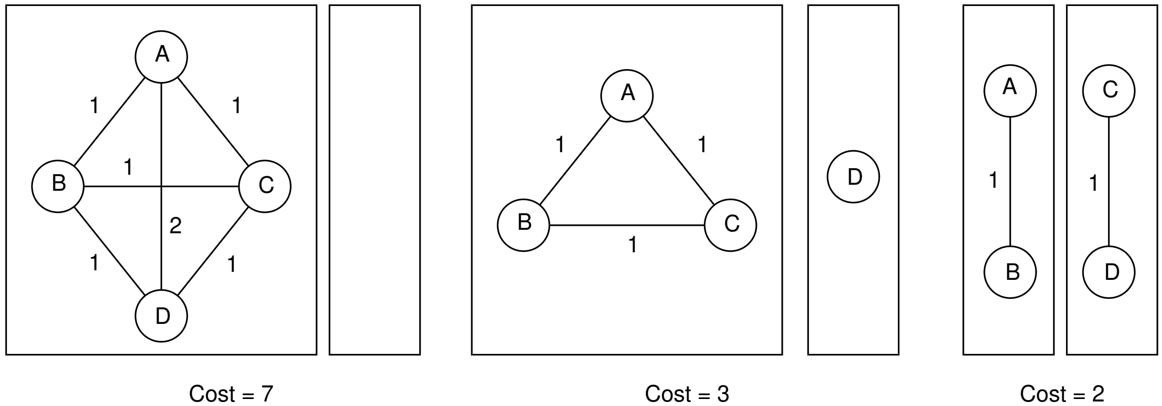
(b)

Figure 3.4: (a) Example program and (b) Corresponding interference graph.

memory operation. The two memory operations could therefore be scheduled in parallel only if they were assigned to different memory units. The algorithm records this fact by adding an interference edge between the two variables that the memory operations access. However, if the two memory operations access the same variable or array, no edge is added to the graph. Instead, the variable is marked for duplication. Section 3.5 discusses the use of duplication in more detail.

To account for the possibility of scheduling any subsequent memory operation in the DRS in parallel with the first memory operation, subsequent memory operations are not marked as being scheduled. This way, an interference edge is added between the variable accessed by the first memory operation and all variables accessed by subsequent memory operations. Since subsequent memory operations are not scheduled, they will be added to the next calculated DRS. Again, only the first memory operation in the new DRS will be scheduled, and an interference edge is added between the variable it accesses and all variables accessed by subsequent memory operations. Thus, when a DRS can no longer be calculated, an interference edge will have been added between every pair of variables that could be accessed in parallel in the basic block. Since the edges of the interference graph represent partitioning relationships among program data, an appropriate weight is assigned to each edge.

To demonstrate how the interference graph is built, Figure 3.4 shows an example program and its corresponding interference graph. In the program, every pairing of the arrays A, B, C, and D may be accessed simultaneously. Since each node in the graph corresponds to one of these arrays, an edge is added between every pair of nodes in the graph. Since arrays A and D can be accessed in parallel inside the loop, edge



(a) (b) (c)

Figure 3.5: Partitioning the nodes of an interference graph.

(A, D) is assigned a weight = 2. On the other hand, since all other pairs of arrays can be accessed in parallel only outside the loop, all other edges of the graph are assigned a weight = 1.

3.4.3 Step 3: Partitioning the Graph

Once the interference graph is constructed, the nodes are partitioned into two sets by searching for a minimum-cost partitioning among all possible partitions. Although this problem is NP-complete [40], a greedy algorithm is used to ensure a near-optimal partitioning. This is by no means the only method that can be used to partition the nodes of the graph, and other algorithms, such as graph coloring, will probably work just as well. However, the greedy algorithm was initially chosen for its simplicity and the need to validate the current approach to data partitioning. Moreover, the performance results in Section 3.6 show that the greedy algorithm yields near-optimal results, thus suggesting that the use of more sophisticated partitioning algorithms is not necessary.

Figure 3.5 shows how the nodes of an interference graph are partitioned. Initially, the algorithm stores all nodes in one of two sets, with the second set being empty. It also sets the initial cost for the partitioning to the sum of the weights of the edges connecting the nodes in the first set. This is shown in Figure 3.5(a). The algorithm then selects a node from the first set such that its removal results in the greatest decrease in cost, and stores the node in the second set. This is shown in Figure 3.5(b), where node D is moved from the first set to the second. The algorithm then selects another node from the first set such that its removal from that set and its addition to the second set results in the greatest decrease in cost. Here, it should be noted that adding a node to the second set may increase the cost if the node is edge-connected to any of the nodes in the second set. That is why a node is only moved from the first set to the second if doing so results in a net decrease in cost. This is shown in Figure 3.5(c), where node C is moved to the second set. This process continues until it is no longer possible to reduce cost, at which point the data is partitioned. This, again, is shown in Figure 3.5(c).

3.4.4 Step 4: Reorganizing the Stack

After partitioning the program data, the variables corresponding to the nodes in the two sets are assigned to separate memory banks. Global variables are easily assigned to a particular memory bank by using special assembly directives. For local variables stored on the stack, the stack frame in each function should be distributed across the dual data-memory banks to ensure that variables get accessed from the proper locations in memory. To do this, the code used to set-up and tear-down a stack frame is modified and augmented to handle two stack frames in different memory banks. To facilitate the handling of both stack frames, a second stack pointer is introduced. Finally, the original stack offsets are re-mapped to their new values.

Figure 3.6 shows the same code example shown in Figure 3.2 after reorganizing the stack. Operations `[1]–[4]` initialize both frames and two stack pointers, `a63` and `a62`, are used to access the X and Y stack frames, respectively. Operations `[5]–[8]` initialize two pointers, `a1` and `a61`, to the top of the area used to store registers in different frames. Operations `[9]–[12]` save two registers in different frames. Since the store operations use different pointers, they can be executed in parallel. Operations `[13]–[15]` and `[16]–[18]` are each used to load and store a register from different locations on different stack frames. Operations `[19]–[22]` initialize `a1` and `a61` in a manner similar to operations `[5]–[8]`. Operations `[23]–[26]` restore the two registers from different frames. Again, since the load operations use different pointers, they can be executed in parallel. Finally, operations `[27]–[30]` release both stack frames. Although the code in Figure 3.6 uses more operations than the code in Figure 3.2, the high levels of available parallelism, especially in saving/restoring registers on the stack and in accessing local variables, enable fewer instructions to be generated and executed, and hence, higher performance to be achieved.

3.4.5 Step 5: Propagating Partitioning Information

After all program variables have been partitioned and assigned to a memory bank, memory operations that access these variables are assigned to the appropriate memory units. Thus, a memory operation used to access a variable assigned to X memory is assigned to memory unit MU0, while a memory operation used to access a variable assigned to Y memory is assigned to memory unit MU1. These assignments are later used by the operation-compaction pass when deciding on which operations to pack into a long instruction. In general, the operation-compaction pass is responsible for scheduling operations and assigning them to the appropriate functional units. However, it relies on the data-allocation pass to assign load and store operations to the appropriate memory units since each memory unit is connected to only one of the memory

```
[1]   movi.a   #frame_size_X,a1
[2]   dec      a63,a1,a63
[3]   movi.a   #frame_size_Y,a61
[4]   dec      a62,a61,a62
[5]   movi.a   #SR_top_X,a1
[6]   inc      a63,a1,a1
[7]   movi.a   #SR_top_Y,a61
[8]   inc      a62,a61,a61
[9]   dec      a1,a0,a1
[10]  dec      a61,a0,a61
[11]  store    (a1),register1
[12]  store    (a61),register2
               .
[13]  movi.a   #new_local_offset_1,a1
[14]  inc      a63,a1,a2
[15]  load     (a2),register3
               .
[16]  movi.a   #new_local_offset_2,a1
[17]  inc      a62,a1,a3
[18]  store    (a3),register4
               .
[19]  movi.a   #SR_top_X,a1
[20]  inc      a63,a1,a1
[21]  movi.a   #SR_top_Y,a61
[22]  inc      a62,a61,a61
[23]  dec      a1,a0,a1
[24]  dec      a61,a0,a61
[25]  load     (a1),register1
[26]  load     (a61),register2
[27]  movi.a   #frame_size_X,a1
[28]  inc      a63,a1,a63
[29]  movi.a   #frame_size_Y,a61
[30]  inc      a62,a61,a62
```

Figure 3.6: UTDSP code to handle dual stack frame.

```
for (n = 1; n < r; n++) {
   R[n] += signal[n] * signal[n+m];
}
```

Figure 3.7: C loop for calculating an autocorrelation vector.

banks. For store operations that access duplicated data, an additional store operation is also added to the code, and the two store operations are assigned to different memory units. Section 3.5 describes this in more detail.

3.4.6 Time Complexity

The time complexity for constructing the interference graph is O($B \times n^2$), where B is the number of basic blocks in a program, and n is the number of operations in a basic block. The time complexity of partitioning the interference graph using the greedy algorithm is O(v^2), where v is the number of nodes in the graph, which is also the number of variables and arrays in the program. Finally, the time complexity of assigning variables and arrays to their corresponding memory banks is O(v). Thus, the time complexity of the CB data partitioning algorithm is O($B \times n^2 + v^2$).

3.5 Partial Data Duplication

Although many of the benchmark programs approached ideal performance using compaction-based partitioning, some did not. Initially, the poor performance was believed to be the result of poorly approximated edge weights in the interference graph. Recall that the weights are used to determine which pairs of variables should be given higher priority when partitioning the graph. However, as Section 3.6.1 demonstrates, using profiling to derive more accurate edge weights has no effect on the performance of the benchmark programs.

Upon further investigation, the loss in performance was found to be due to data-access patterns like the one shown in Figure 3.7, where accesses are made to two different elements of the same array. In such a case, the data cannot be divided to allow simultaneous accesses because the entire array is stored in one memory bank. Three solutions are possible for this problem: using a low-order interleaved memory system, using dual-ported memory cells, or duplicating the data and storing a copy in each memory bank.

Using low-order interleaving would provide dual parallel access for the example in Figure 3.7, but only if the value of `m` is odd. Even values of `m` would cause the two references to access the same memory bank. Thus, low-order interleaving does not provide a general solution for such situations.

Another solution is to use dual-ported memory cells instead of separate single-ported memory banks. Two elements from the same array could then be accessed simultaneously, even when the entire array is stored in one memory bank, since each access would be made through a separate port. Additionally, with dual-ported memory, no special data partitioning is needed to guarantee simultaneous accesses. However, the major disadvantage to using dual-ported memory is its cost. Adding another port to each memory cell increases its area, as well as the power consumption of the memory system. Moreover, the memory access time may be negatively affected. Finally, the dual-ported nature of such a memory system may not be necessary for all applications. Since most applications can rely on a data partitioning algorithm to obtain the necessary memory bandwidth, providing a dual-ported memory for such applications is not cost-effective.

The third solution, which is implemented entirely in software, is to duplicate the arrays in both memory banks. The immediate cost is the extra memory area required to hold the duplicated arrays, and the additional operations needed to maintain the integrity of these arrays. However, with an effective data-partitioning algorithm, it is not necessary to duplicate arrays for all applications, nor for all arrays in a particular application. Thus, by judiciously choosing which arrays to duplicate, increased cost would only be incurred when a gain in performance is possible. Moreover, although memory size is fixed, using data duplication in this manner provides flexibility. Memory is “wasted” for duplication only when necessary; otherwise it can be used as additional storage for those applications that do not require duplication.

To study the impact of data duplication on performance and cost, the CB data-partitioning algorithm was augmented to replicate all arrays that could be accessed simultaneously. Such arrays can easily be identified during the construction of the interference graph. Recall from Figure 3.3 that adding an edge to the graph requires examining a pair of memory operations to determine if they access the same array. If they do, the array must be duplicated to allow parallel access. Since the arrays that can be duplicated can be determined at compile-time, this approach is called *partial* data duplication.

To avoid fragmenting memory, duplicated arrays are allocated to both banks before other variables. This enables the same address to be used to access the same array element from both data banks. The algorithm was also modified to maintain the data integrity of duplicated arrays: each time the algorithm identifies a store operation to a duplicated array, it adds another store operation to the program to keep the data in both memory banks current.

Duplicating data may not always result in an increase in performance since additional store operations could potentially degrade performance. This would happen if the compaction pass could not pack these additional operations into existing long instructions, and would have to create additional long instructions. If this were to occur in critical portions of the code that account for a significant portion of the execution time, it is possible that any performance gains due to parallel memory access could be negated by the performance degradations due to the additional operations. The overall effect would then be either no change in performance, or a possible decrease in performance.

Finally, duplication could complicate interrupt handling. Since stores to different copies of duplicated data may be scheduled in different instructions, it is possible that an interrupt may occur after the instruction containing a store to one copy of duplicated data and before the instruction containing the store to the other copy. It is further possible that the interrupt may update the duplicated data, a common occurrence in embedded systems where external data is fed to the system on a continuous basis. To ensure that both copies are the same, even in the presence of interrupts, a special pair of store operations could be used for updating duplicated data. The first update would use a store-lock operation which would prevent interrupts from occurring and the second update would use a store-unlock operation to enable interrupts again. Alternatively, the interrupt handling routine would need to know that there are two copies of the data to update.

3.6 Performance and Cost Evaluation

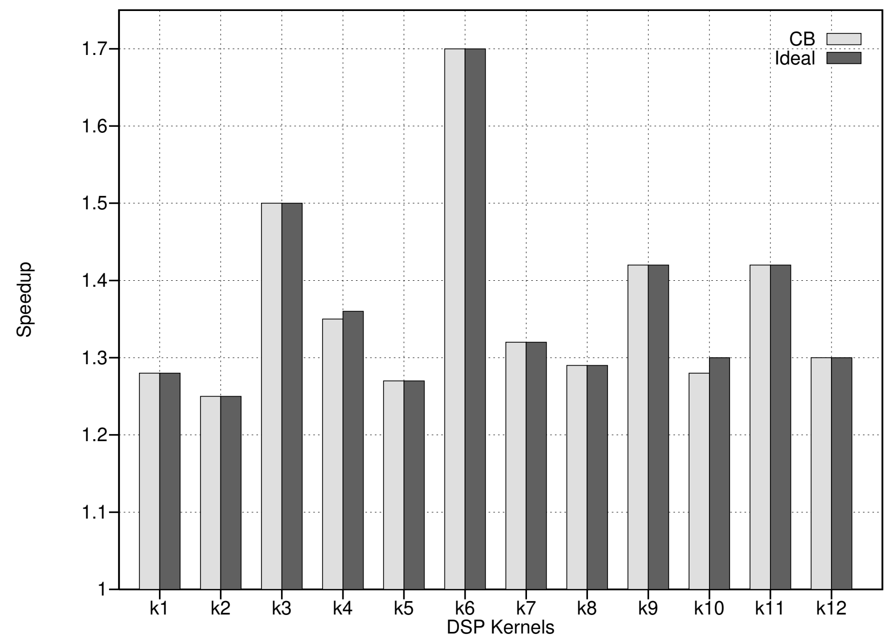
To evaluate the effectiveness of both algorithms, the suite of DSP benchmarks was compiled, the generated code was executed on the instruction-set simulator, and the performance gain relative to code that allocates all data to one memory bank was measured. The additional storage requirements when using duplication were also studied to determine whether this technique is cost-effective.

3.6.1 Performance Results

Figures 3.8 and 3.9 show the performance results for the kernels and the applications, respectively. The results are given as the percentage increase in performance over the case that uses neither partitioning nor duplication. To produce the performance data for this latter case, the programs were compiled with the data allocation pass disabled, but with all other optimizations enabled. In this case, data is stored in only one memory bank. Performance is measured by the number of cycles executed.

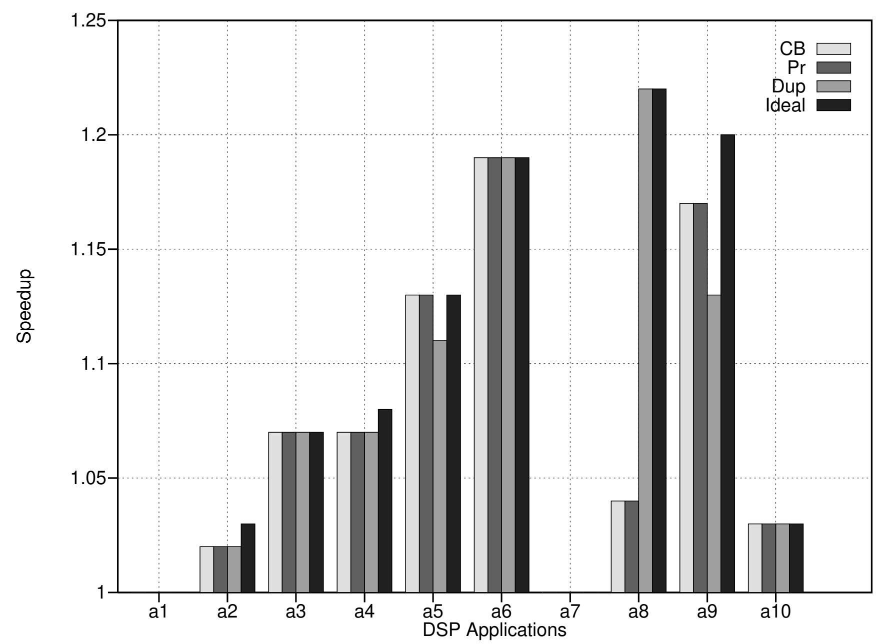
The Ideal data is the performance gain if a dual-ported memory is used. Since accesses in dual-ported memory can occur in parallel without being constrained by the placement of data, such a memory provides the best possible performance, albeit at the expense of using dual-ported memory cells. An objective of developing the CB partitioning algorithm is to achieve Ideal performance by using only single-ported memory banks.

When only compaction-based partitioning and no duplication is used (labeled CB in Figures 3.8 and 3.9), Ideal performance is obtained for all but two of the kernels. Even for the two kernels, `fir_32_1` and `lmsfir_8_1` (k4 and k10 in Figure 3.8), using CB partitioning improves performance by 35% and 28%, respectively. These figures are only one and two percentage points less than the gain achieved with



k1	fft_1024
k2	fft_256
k3	fir_256_64
k4	fir_32_1
k5	iir_1_1
k6	iir_4_64
k7	latnrm_32_64
k8	latnrm_8_1
k9	lmsfir_32_64
k10	lmsfir_8_1
k11	mult_10_10
k12	mult_4_4

Figure 3.8: Performance gain for kernel benchmarks.



a1	G721_A
a2	G721_B
a3	V32.modem
a4	adpcm
a5	compress
a6	edge_detect
a7	histogram
a8	lpc
a9	spectral
a10	trellis

Figure 3.9: Performance gain for application benchmarks.

dual-ported memory. Overall, partitioning improves performance for all the kernels. Performance gains range from 25% – 70%, with an average of 36%.

In contrast, the performance gains for the applications are less pronounced even when using an Ideal memory system. The reason for this is that the kernels consist of loops with large amounts of parallelism and several memory operations. Exploiting load/ store parallelism inside these loops has a significant impact on overall performance. On the other hand, applications, while also containing loops, consist of other sections of code, such as control code or the intervening code between loops. These sections of code generally contain less parallelism than loops, but also contain many memory operations. Exploiting memory parallelism in these sections of code has little impact on performance.

The complete absence of memory parallelism is evident in the two applications that do not benefit at all from a dual-ported memory — `G721_A` and `histogram` (a7 and a9 in Figure 3.9). These applications show that any partitioning algorithm is also unlikely to improve their performance, and that other means must first be applied to expose more memory parallelism in these programs.

Still, applications with some amounts of memory parallelism benefit from using CB partitioning. When performance gains are possible — Ideal improvement is greater than 0% — the improvement in performance when using CB partitioning ranges from 2% – 19%, with an average of 9% when excluding those programs whose performance cannot be improved. The average decreases to 7% when taken over all applications.

In comparison, Ideal improvement for the applications ranges from 3% – 22%, with an average of 12%. When taken over all applications, the average decreases to 9%. This shows that CB partitioning is missing significant amounts of parallelism in some applications. This is most evident in the `lpc` application (a8), where Ideal achieves a performance gain of 22% versus only 4% for the CB algorithm.

Comparing the code generated for the Ideal case and for the CB algorithm revealed that the higher Ideal performance was due to exploiting memory parallelism in loops whose lengths were unknown at compile-time. This led to the hypothesis that CB partitioning was not giving the variables in these loops high enough priority due to the simple heuristic of using loop nesting depth for the edge weights in the interference graph. In turn, this led to the use of profiling to assign more accurate edge weights. The results, however, labeled Pr in Figure 3.9, were not as expected.

The use of profile-driven edge weights resulted in different data partitionings for only a few benchmarks. In these benchmarks, slight changes in the code schedules were observed due to the new partitioning. However, these changes resulted in identical performance improvements to those of the original CB partitioning. In turn, this suggested that the heuristic edge weights are satisfactory for this set of benchmarks, and that there is no need for profiling when constructing the interference graph. It also indicated that the inability of CB partitioning to exploit memory parallelism in some loops was due to other factors.

Upon further inspection, it became clear that the performance degradation was due to simultaneous accesses to elements of the same array, and this led to the addition of partial data duplication. Partial duplication was used in six applications — `G721_B`, `V32.modem`, `compress`, `lpc`, `spectral`, and `trellis` (a2, a3, a5, a8, a9, and a10 in Figure 3.9) — since these were the only ones where simultaneous accesses to the same array were observed. As Figure 3.9 shows, partial duplication, labeled Dup, achieves an ideal performance gain of 22% for `lpc` (a8) compared to a mere 4% gain with CB partitioning. On the other hand, partial duplication achieves performance gains identical to CB partitioning for each of `G721_B`, `V32.modem`, and `trellis` (a2, a3, and a10 in Figure 3.9). For `V32.modem` and `trellis`, the performance achieved by partial duplication is also identical to Ideal. For `G721_B`, the performance achieved by partial duplication is only one percentage point less than Ideal. Finally, partial duplication achieves lower performance gains than CB partitioning for `compress` and `spectral` (a5 and a9 in Figure 3.9). For these applications, the execution of the additional store operations required to maintain the integrity of the duplicated data offsets the performance that would otherwise be gained from parallel memory accesses.

3.6.2 Performance/Cost Trade-Offs of Data Duplication

The performance gained from partial data duplication is achieved at the cost of increased memory requirements for storing the extra copy of data as well as for storing additional operations. To assess the impact of data duplication on system costs, a first-order cost model was used. The model equates cost to an estimate of the area occupied by memory, and is described by the following equation:

$$Cost = Area(X + Y + (2 \times S) + I)$$

In this equation, X, Y, S, and I represent the memory sizes, in words, of X memory, Y memory, the stack, and the instruction-memory bank respectively. The stack size, S, is multiplied by two since it is used in both data-memory banks. The instruction-memory size, I, represents the number of words needed to store a given program, which depend on the way instruction memory is organized and on the way long instructions are encoded. The organization of instruction memory and the encoding and storage of long instructions are the subject of Chapter 5.

Estimating the area occupied by memory follows the same methodology, described in Chapter 2, for estimating the area of the processor. In calculating the memory-area estimates, it is assumed that a memory word consists of 32-bits and that the feature size is 0.25 μm. To enable a fair comparison with the Ideal case, it is also assumed that the area occupied by a dual-ported memory cell is 50% larger than that of a single-ported memory cell. Recall that using a dual-ported data memory bank does not require additional storage but instead increases cost in terms of larger chip area, higher power consumption, and possibly longer access times. The latter two factors are not addressed in this study.

The memory cost of using partial duplication is compared to that of using CB partitioning, the Ideal case, and full duplication. Full duplication is included in the comparison to demonstrate the significant cost savings of duplicating only those arrays that would result in a performance gain when used in conjunction with partitioning.

Table 3.1 shows the Performance Gain (PG) of each technique relative to the unoptimized case. Interestingly, full duplication does not always improve performance as much as the other techniques do. In fact, for each application there is at least one other technique that performs as well as, if not better than, full duplication. This is because the additional book-keeping operations offset the performance gains of duplicating all data. Consequently, the average performance gain due to full duplication is slightly greater than partial duplication, and is less than that for the Ideal case.

Table 3.1 also shows the Cost Increase (CI) of each technique, due to changes in storage requirements, relative to the case when no memory parallelism is exploited. Here, it is important to note that changes in storage requirements include the effects on both instruction and data memories.

Not surprisingly, full duplication incurs a large increase in cost. On average, 49% more memory is needed. Similarly, using a dual-ported memory increases cost by an average of 39%. In contrast, the average increase in cost when using partial duplication with partitioning is only 4%, demonstrating the importance of using partitioning to keep cost increases minimal while improving performance. For `compress` and `spectral`, the cost difference for CB partitioning is actually a decrease in cost that is due to parallel memory accesses being packed into fewer instructions.

Finally, Table 3.1 shows the Performance/Cost Ratio (PCR) of each technique. PCR is the ratio of the Performance Gain to the Cost Increase, and indicates the goodness of the performance/cost trade-off for each technique. A Performance/Cost Ratio greater than 1.00 indicates that the improvement in performance outweighs the increase in cost. When using full duplication or dual-ported data-memory, the increase in memory requirements is always greater than any gains in performance. When compared to partial duplication or CB partitioning, it is clear that neither technique is cost-effective. The same holds true for partial duplication when it is applied to `V32.modem` and `trellis`.

For most applications, partial duplication and CB partitioning achieve PCR's that are greater than or equal to 1.00. However, this does not always imply that the gain in performance is achieved cost-effectively. For applications that do not require the use of duplication, but that benefit from CB partitioning

Application Benchmarks	Full Duplication			Partial Duplication			CB Partitioning			Ideal Dual-Ported Memory		
	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR
G721_A	1.00	1.69	0.59	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.49	0.67
G721_B	1.02	1.71	0.60	1.02	1.01	1.01	1.02	1.00	1.02	1.03	1.48	0.70
V32.modem	1.07	1.26	0.85	1.07	1.15	0.93	1.07	1.00	1.07	1.07	1.36	0.79
adpcm	1.05	1.21	0.87	1.07	1.00	1.07	1.07	1.00	1.07	1.08	1.30	0.83
compress	1.11	1.23	0.90	1.11	1.03	1.08	1.13	0.99	1.14	1.13	1.34	0.84
edge_detect	1.19	1.97	0.60	1.19	1.00	1.19	1.19	1.00	1.19	1.19	1.50	0.79
histogram	1.00	1.94	0.52	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.50	0.67
lpc	1.22	1.43	0.85	1.22	1.13	1.08	1.04	1.01	1.03	1.22	1.34	0.91
spectral	1.14	1.22	0.93	1.13	1.06	1.07	1.17	0.99	1.18	1.20	1.32	0.91
trellis	1.03	1.24	0.83	1.03	1.05	0.98	1.03	1.03	1.00	1.03	1.26	0.82
Geometric Mean	1.08	1.46	0.74	1.08	1.04	1.04	1.07	1.00	1.07	1.09	1.39	0.79

Table 3.1: Impact of CB partitioning and partial duplication on performance and cost.

(`adpcm` and `edge_detect`, for example), the gain in performance is achieved without an increase in cost. On the other hand, for applications that require duplication, the trade-off is not so consistent. For `lpc`, the PCR when using duplication (1.08) is greater than the PCR when using CB partitioning only (1.03), suggesting that applying partial duplication is advantageous. For the other benchmarks — `G721_B`, `V32.modem`, `compress`, `spectral`, and `trellis` — the PCR when using duplication is less than that when using CB partitioning. This indicates that duplication is not cost-effective for these applications.

3.6.3 Selective Partial Duplication

While PCR could be used as a metric for determining the cost-effectiveness of data duplication, more information is required from the application designer to determine the acceptable trade-offs. Depending on the real-time performance constraints and area budget for the system, a high PCR or a low PCR may be acceptable. By considering maximum execution times and storage capacities the designer could be more selective in duplicating data to minimize storage while meeting the performance requirements. That is, rather than duplicating all arrays that may be accessed simultaneously, the designer would only choose to duplicate a subset of these arrays based on performance/cost trade-offs.

To evaluate the impact of duplicating a subset of the arrays on cost and performance, the compiler should be augmented to generate different code schedules for each subset of arrays duplicated. Recall that duplicating an array affects the code schedule through the introduction of additional store operations that are used to ensure the integrity of data in both memory banks. Depending on the subset of arrays that is chosen and the corresponding code schedule, the impact on data and program storage requirements, and hence on cost, can easily be evaluated. Profiling information should also be used to evaluate the impact on performance. Although selective duplication is not fully supported by the current tools, the data allocation pass assumes that the designer has studied the performance/cost trade-offs, and queries the user about duplicating arrays that may be accessed simultaneously.

3.7 Summary

A common feature of programmable DSPs is their use of dual data-memory banks to double the bandwidth of the memory system. To use dual memory banks effectively, most programmers resort to assembly language programming or high-level languages with special features to direct the use of memory by the compiler. This chapter demonstrated that it is possible to use a standard, high-level, programming language, such as C, and effectively allocate the data to the two memories without adding any pragmas or other features.

Two complementary algorithms, called compaction-based (CB) data partitioning and partial data duplication, were presented. These were implemented as part of a data allocation pass in the UTDSP optimizing C compiler. Although dual-ported memories would obviate the need for these algorithms, the added cost in memory area is not justified, particularly when good performance can be achieved using single-ported, dual memory banks.

The CB data partitioning algorithm allocates program variables to different memory banks so that as much parallelism, and hence performance, can be obtained. The algorithm is based on partitioning the nodes of an interference graph whose nodes represent variables and whose edges represent potential parallel accesses to pairs of variables. To build the graph, an operation-compaction algorithm is used to suggest pairs of memory operations that may be executed in parallel, and this information is used to add edges to the graph. A weight is also assigned to each edge to represent the degradation in performance if the corresponding variables are not accessed in parallel. A greedy algorithm is used to partition the nodes into two sets, corresponding to the two memory banks of the UTDSP architecture. Although not considered in this study, the greedy algorithm can also be used to partition the nodes of the interference graph into more than two sets, and can therefore be used to partition data among multiple data-memory banks.

When simultaneous accesses to the same array occur, data partitioning alone is not sufficient. Instead, it is necessary to duplicate the data in both memory banks to exploit the available parallelism. However, data duplication increases memory requirements, and hence system costs, due to the extra space needed to store duplicated data and the extra operations needed to ensure the integrity of duplicated data. Executing the extra operations may also degrade performance. Thus, partial duplication should only be applied if the performance gain justifies the cost increase. Results were presented for the case where all arrays with simultaneous accesses are duplicated. If the Performance/Cost Ratio is too low, a further refinement is to determine whether some of these arrays do not have to be duplicated since doing so does not significantly affect performance.

The performance results indicate that CB partitioning is an effective technique for exploiting dual memory banks. This is evident in the kernel benchmarks where CB partitioning improves performance by 25% – 70%. When a dual-ported memory is used as an ideal reference for how well the algorithms perform, the performance gains for all kernels were identical, or nearly identical for two of the kernels, to those attained in the ideal case. CB partitioning also improves the performance of the application benchmarks by 2% – 19%. In the ideal case, the performance gain is 3% – 22%. For one application, `lpc`, CB partitioning improves performance by only 4%. Using partial data duplication, performance improves by an ideal 22%.

Having examined the problem, faced by most DSP compilers, of allocating data automatically to dual data-memory banks, the focus now shifts to architectural issues. The next chapter examines the use of VLIW architectures in embedded DSP applications.

Chapter 4

The Case for VLIW DSP Architectures

Digital signal processors (DSPs) are specialized microprocessors optimized to execute the computationally-intensive operations commonly found in the inner loops of digital signal processing algorithms. Since DSPs combine high performance with low cost, they are used extensively in embedded systems. A common characteristic of DSPs is their use of tightly-encoded instruction sets. For example, a common DSP instruction can specify up to five parallel operations: multiply-accumulate, two pointer updates, and two data moves using 16-bit to 48-bit instruction words [23],[24],[25],[26],[27],[28].

Initially, tightly-encoded instruction sets were used to improve code density with the belief that this also reduced instruction memory requirements, and hence cost. Tightly-encoded instructions were also used to reduce instruction memory bandwidth, which was of particular concern when packaging was not very advanced. As a result of this encoding style, DSP instruction-set architectures (ISAs) are not well suited for automatic code generation by modern, high-level language (HLL) compilers. For example, most operations are accumulator based, and very few registers can be specified in a DSP instruction. Although this reduces the number of bits required for encoding instructions, it also makes it more difficult for the compiler to generate efficient code. Furthermore, DSP instructions can only specify limited instances of parallelism that are commonly used in the inner loops of DSP algorithms. As a result, DSPs cannot exploit parallelism beyond what is supported by the ISA, and this can degrade performance significantly.

For the early generation of programmable DSPs, tightly-encoded instruction sets were an acceptable solution. At that time, DSPs were mainly used to implement simple algorithms that were relatively easy to code and optimize in assembly language. Furthermore, the compiler technology of that time was not advanced enough to generate code with the same efficiency and compactness as that of a human programmer. However, as DSP and multimedia applications become larger and more sophisticated, and as design and development cycles are required to be shorter, it is no longer feasible to program DSPs in assembly language. Furthermore, current compiler technology for general-purpose processors has advanced to the point where it can result in very efficient and compact code. For these reasons, DSP manufacturers are currently seeking alternative instruction set architectures. One such alternative, used in new media processors and DSPs such as the Texas Instruments TMS320C6x, is the VLIW architecture.

In this chapter, the performance and cost of VLIW-based architectures are compared to those achieved with more traditional DSP architectures. Section 4.1 examines several architectural alternatives for implementing embedded DSPs, and explains why VLIW-based architectures are the most suitable among them. Section 4.2 then describes the methodology used to model two commercial DSPs and compare their performance and cost to those of similar VLIW-based architectures. Section 4.3 presents the results of these comparisons, and Section 4.4 summarizes the main points of this chapter.

4.1 Architectural Alternatives to Traditional DSPs

Among the features that are desirable to have in an embedded DSP architecture is the ability to exploit parallelism. This is especially important with DSP applications since they exhibit high levels of parallelism. Another desirable feature is that the architecture be an easy target for HLL compilers. This enables the processor to be programmed in a HLL, which reduces code development time, increases portability, and improves maintainability. It also makes it easier for a modern compiler to optimize the generated code and improve execution performance. Finally, to meet the low cost requirements of embedded DSP systems, an architecture must occupy a small die area, and must use functional and control units having low complexity. This section describes some of the architectural alternatives that can be used to implement embedded DSPs, and discusses their advantages and disadvantages.

4.1.1 Superscalar Architectures

Superscalar architectures are the current style used to implement general-purpose processors. They exploit parallelism in the stream of instructions fetched from memory by issuing multiple instructions per cycle to the datapath. As instructions are fetched from memory, they are temporarily stored in an instruction buffer where they are examined by a complex control unit. The control unit determines the interdependencies between these instructions, as well as the dependencies with instructions already executing. Once these dependencies are resolved, and hardware resources become available, the control unit issues as many instructions in parallel as possible for execution on the datapath.

In older superscalar architectures, instructions could only be issued in parallel if they were in the correct static order inside the instruction buffer. Since instructions could only be issued in the same order they were fetched, these architectures were said to exploit *static parallelism*. However, more recent superscalar architectures are capable of exploiting parallelism among any group of instructions in the buffer. As such, they are said to exploit *dynamic parallelism*. This is achieved, in part, by using such techniques as dynamic register renaming or branch prediction, which eliminate false dependencies between instructions and increase the ability to issue instructions in parallel [42]. Instructions in the buffer can therefore be issued out of the order in which they were fetched, and the control unit is responsible for ensuring that the state of the processor is modified in the same order that instructions are fetched.

The compiler technology associated with superscalar architectures is also very advanced and well understood. In addition to generating efficient code by using machine-independent, scalar optimizations [13], contemporary optimizing compilers are capable of exploiting the underlying features of their target architectures. Typically, they apply machine-dependent optimizations such as instruction scheduling, which minimizes pipeline latencies and helps expose parallel instructions to the hardware; data prefetching, which exploits the memory hierarchy to minimize memory latencies; register renaming, which statically removes false dependencies between instructions; and branch prediction, which minimizes branch penalties by predicting execution control paths.

Given their ability to exploit parallelism dynamically, and their sophisticated compilers, superscalar architectures are capable of exploiting the high levels of parallelism found in DSP applications. However, as available parallelism increases, and the number of functional units and registers increase to support it, the complexity and cost of their control units also increase. Given the stringent low-cost requirements of embedded DSP systems, superscalar architectures may not therefore be the most cost-effective architectural choice.

4.1.2 SIMD Architectures

With the recent growth in multimedia applications for desktop computer systems, many general-purpose processor vendors have introduced multimedia extensions to their instruction set architectures [43]. Typically, these extensions take the form of special, single-instruction, multiple data (SIMD) instructions that perform identical, parallel operations on a fixed number of packed operands. The operands are typically 8-bit, 16-bit, or 32-bit integers that are packed into 32-bit or 64-bit registers. That is why these SIMD instructions are sometimes called *packed arithmetic* instructions, or are said to exploit *subword parallelism*.

SIMD instructions enhance the execution performance of *vectorizable* loops[1] that perform homogeneous operations on arrays of data. Since such loops are very common in multimedia and DSP applications, SIMD architectures can be used for these applications. Programs that use SIMD instructions can therefore achieve higher levels of code density and require less memory to store. The amount of hardware needed to support SIMD instructions is also minimal, and consists mainly of the additional control circuitry that

1. A loop is vectorizable if it has no real dependence cycles between its different iterations.

enables an ALU to perform multiple slices of the same operation on individual subwords of its input operands.

Currently, SIMD architectures are difficult targets for HLL compilers, and, like traditional DSPs, must be programmed in assembly language for their full benefits to be achieved. Since high-level languages, such as C, do not specify sub-word parallelism, it is very difficult for compilers to extract such parallelism. Instead, compilers for SIMD architectures will have to incorporate features found in vectorizing compilers, and this is the subject of current research [44],[45],[46].

4.1.3 Multiprocessor Architectures

For applications that exhibit high levels of coarse-grained parallelism, such as video or radar signal processing, multiprocessor architectures can be used. For example, the Texas Instruments TMS320C80 [47] packs four DSP processors and a RISC controller into the same chip. The four DSPs typically execute the same code, and process four different data sets simultaneously. Since compiler support for such multiprocessor architectures has typically been poor or non-existent, applications must be carefully coded to benefit from the multiple processors.

In general, the number and type of processors used in a multiprocessor architecture, whether traditional DSPs, VLIW architectures, superscalar architectures, or SIMD architectures, depends on the application. However, since multiprocessor architectures are more costly than single-processor architectures, they are only used when the required level of performance justifies the added cost.

4.1.4 VLIW Architectures vs. Other Architectural Styles

Since DSP applications typically exhibit static control flow, compilers can easily be used to schedule instructions statically and exploit the available parallelism. This makes VLIW architectures, with their simpler control units, a more cost-effective choice than superscalar architectures in embedded systems.

VLIW architectures are also better suited for exploiting parallelism in DSP applications than SIMD architectures. Although SIMD architectures are suitable for applications that exhibit homogeneous parallelism, they are not well suited for applications that exhibit more heterogeneous parallelism, such as that found in the inner loops of DSP kernels, or the control portions of DSP applications. The greater flexibility in VLIW architectures also makes them easier targets for HLL compilers. However, in their most basic form, they are also more expensive to implement. For example, while SIMD architectures use a single ALU to perform four parallel operations, a VLIW architecture requires four full-precision ALUs to perform the same operations. Moreover, while the four operations can be encoded as a single SIMD instruction, they have to be encoded as four, separate operations in a long instruction. One way to reduce the cost of a VLIW architecture is to customize it to the target application. For example, smaller-precision ALUs and datapaths can be used to match the requirements of the application. Instruction storage and bandwidth requirements can also be reduced by using efficient long-instruction encoding techniques. Even specialized SIMD operations can be used to supplement a VLIW architecture and reduce its cost. This, however, requires more sophisticated compiler technology to exploit available sub-word parallelism.

Finally, VLIW architectures are better suited for exploiting the fine-grained parallelism found in DSP applications than multiprocessor architectures. Even though some applications can benefit from the exploitation of coarse-grained parallelism, most applications cannot justify the additional cost of a multiprocessor architecture. Multiprocessor DSP architectures are also difficult targets for HLL compilers, making them very cumbersome to program.

Since VLIW architectures have the flexibility to exploit different levels of parallelism, have simple control structures, and are easy targets for HLL compilers, they are the most suitable architecture for embedded DSP systems. However, a major impediment to using VLIW architectures in cost-sensitive embedded systems is their high instruction bandwidth and storage requirements. Chapter 5 proposes a solution to this problem that preserves the flexibility of the long-instruction format while reducing its bandwidth and storage requirements. The remainder of this chapter shows the performance and cost benefits of using VLIW architectures compared to more traditional DSP architectures.

4.2 Experimental Methodology

The previous section described different architectural alternatives to traditional DSP architectures, and compared their advantages and disadvantages with the VLIW architecture. This section describes the methodology used to model and compare the performance and cost of two traditional DSP architectures to the VLIW-based UTDSP architecture.

4.2.1 Modeling Traditional DSP Architectures

To compare existing commercial DSP architectures and their compilers, the same source code should ideally be compiled to the different architectures, and the results compared. However, since DSP compilers vary in their levels of sophistication and their ability to exploit the specialized features of their target architectures, it is difficult to rely on existing commercial compilers. In addition to reflecting the differences between the architectures, the code generated by these compilers would reflect the differences in the compilers. Furthermore, it is usually difficult to compile DSP applications on different compilers without modifying the source code to suite the requirements imposed by the different compilers. One way to overcome these problems was to use the UTDSP compiler, and model the resources and parallelism available in existing DSPs. This would allow a more accurate comparison between the different architectures to be made.

Most DSPs use tightly-encoded instruction sets that support the exploitation of specific instances of parallelism. DSPs are also memory-based, and use few general-purpose registers. To study the effects of restricted parallelism and small register sets on performance and cost, the UTDSP compiler was modified to generate code for two architectures modeled after commercial DSPs. In both cases, the number of functional units on the UTDSP model architecture was constrained to reflect the datapaths of the DSPs under study. The compaction algorithm in the compiler back-end was also modified to only generate combinations of parallel operations that are supported by the instruction sets of the respective architectures. Finally, the number of registers available to the compiler was also changed to reflect the number of registers used in each architecture.

The code generated by the modified UTDSP compiler are still VLIW instructions, and they can be run on the instruction-set simulator. However, the parallelism is restricted to what is found in the processors being modeled. Each VLIW instruction corresponds, approximately, to an existing tightly-encoded instruction in the DSP being modeled. However, the actual DSPs restrict the way the registers are used in each instruction. Compiling code with these constraints is a difficult problem that is still being tackled by DSP compiler writers [35]. For the modified UTDSP compilers, these restrictions were not modeled. Instead, all available registers could be used as the source or destination of any operation, which is one of the keys to generating efficient code using known compiler technology. Consequently, the results for the modeled processors will be overly optimistic.

To provide some insight into the difference in performance that can result from relaxing the constraints on the registers, two kernels, `fir` and `iir`, were examined. The C code for the kernels was taken from an EDN magazine article comparing the performance of different commercial DSP compilers [48]. Initially, the C code was compiled using the UTDSP compiler, after restricting the parallelism that it can exploit to the amount found on the Motorola DSP56002. The DSP56002 was chosen because it was one of the DSP targets used in the EDN article, and because it was easy to model on the UTDSP architecture. The instructions generated by the UTDSP compiler were then compared to the corresponding assembly code generated by the Tasking compiler for the DSP56002. The execution time for each set of assembly code was then calculated assuming an execution rate of one cycle per instruction, and accounting for loop counts. The results showed that the code generated by the Tasking compiler was 1.4 – 2.0 times slower than that generated by the modified UTDSP compiler.

Since both the Tasking compiler and the modified UTDSP compiler were capable of exploiting the parallelism supported by the DSP56002, and since DSPs such as the DSP56002 are specifically designed to exe-

<table>
<tr><th colspan="3">Bin 1</th><th colspan="4">Bin 2</th></tr>
<tr><td>Multiply or Logical</td><td>Add</td><td>Subtract</td><td>Memory</td><td>Memory</td><td>Address</td><td>Address</td></tr>
<tr><td colspan="3" rowspan="2">Shift or Bit-Manipulation</td><td colspan="4">Register Move</td></tr>
<tr><td colspan="4">Branch</td></tr>
</table>

Table 4.1: Parallelism supported by the ADSP-21020. Operations in Bin 1 may be executed in parallel with operations in Bin 2. Within each bin, several operations may also be executed in parallel.

cute kernels such as `fir` and `iir` efficiently, the results highlight the significance of restricting the use of registers on overall performance. Since the UTDSP compiler does not restrict the use of registers, the effects of such restrictions should therefore be remembered when comparing the performance results of the UTDSP architecture and the modeled DSP architectures.

The first DSP, *DSP_mot*, was modeled after the Motorola DSP56001 [23]. This DSP was chosen because it is a popular, fixed-point DSP, others being the Texas Instruments TMS320C50 [25] and Analog Devices ADSP-2100 [27]. However, since all the benchmarks use floating-point data types and operations, the *DSP_mot* was allowed to execute floating-point operations subject to the same restrictions imposed by its instruction set architecture. However, this is acceptable since the available parallelism can still be modeled. The DSP56001 can execute most arithmetic and logic operations in parallel with up to two data moves and two address calculations. A data move is either a memory access or a register move. The DSP56001 also uses eight data registers and 12 address registers. Even though the address registers in the DSP56001 have specialized uses — there are specific base, offset, and modulo registers — these restrictions were not programmed into the UTDSP compiler. Instead, the compiler was free to use all 12 address registers to store base, offset, or modulo values.

The second DSP, *DSP_ad*, was modeled after the Analog Devices ADSP-21020 [28]. This DSP was chosen because it is a representative floating-point DSP, others being the Texas Instruments TMS320C30 [26] and Motorola DSP96002 [24]. Table 4.1 shows the parallelism supported by the ADSP-21020's instruction set, which is more flexible than that of the DSP56001 because it allows more combinations of operations to be executed in parallel. The ADSP-21020 also uses 16 data registers and 64 address registers. However, of these 64 address registers, 16 are used to store segment base addresses. Since the UTDSP uses a flat address space, these registers are not required, and the *DSP_ad* is modeled with only 48 address registers.

Finally, to study the effect of using a VLIW with restricted parallelism and more registers on performance and cost, the number of registers used in *DSP_mot* and *DSP_ad* was increased to match the UTDSP architecture. Since *DSP_ad* has more address registers than the UTDSP architecture, only the number of its data registers was increased. The resulting models are called *UTDSP_mot* and *UTDSP_ad*, and are true VLIW versions of the *DSP_mot* and *DSP_ad* architectures. This means that, even though they have the same functional units as *DSP_mot* and *DSP_ad*, they are not restricted by the instruction-set architecture in exploiting available parallelism. Furthermore, since they have more registers, they make it easier for the compiler to generate more efficient code.

4.2.2 Measuring Performance and Cost

Execution performance is typically measured by execution time. To calculate the execution time of the benchmarks on the more traditional DSP architectures, basic-block execution frequencies were used. For every basic block, the number of instructions generated by the compiler was multiplied by the execution frequency of the block. The total execution time was then calculated as the sum, over all basic blocks, of this product. Since the compaction algorithm is limited to exploiting parallelism within a basic block, it does not affect branch operations, nor does it change a program's control-flow structure.

To compare the performance of the traditional DSP and restricted VLIW architectures with the full-fledged UTDSP architecture, their *relative performance* was measured. Relative performance is defined as the ratio of the execution time of the UTDSP architecture while exploiting full parallelism — this will be referred to as *UTDSP_parallel* — to the execution time of the given processor. Thus, a processor with relative performance less than 1.0 has worse performance than the UTDSP architecture. To put these results in perspective, the relative performance of the UTDSP architecture when no attempt is made to exploit parallelism is also calculated. This architecture will be referred to as *UTDSP_serial*, and can be thought of as a statically-scheduled, single-issue processor executing one operation per cycle. It represents the UTDSP architecture's worst-case performance. Table 4.2 shows the actual cycle counts for each of the benchmarks when executed on *UTDSP_parallel*.

To measure the impact of using different architectures on cost, the area-estimation model, described in Chapter 2, was used. Recall that this model uses technology-independent estimates of the area occupied by different datapath circuits and components, and a set of architectural parameters that describe a datapath. Using the area estimates and the architectural parameters, the model can calculate a first-order estimate of the area occupied by the datapath. By changing the model's parameters, different datapaths, such as *DSP_mot*, *DSP_ad*, *UTDSP_mot*, and *UTDSP_ad* can be modeled, and their areas can be estimated.

4.3 Results and Analysis

This section examines the impact of tightly-encoded DSP instruction sets on performance, and compares it to the performance achieved by the more flexible, VLIW-based UTDSP architecture. It also examines the cost of using a VLIW architecture in terms of chip area.

Kernels		
k1	fft_1024	41834
k2	fft_256	10332
k3	fir_256_64	32995
k4	fir_32_1	94
k5	iir_4_64	50
k6	iir_1_1	1643
k7	latnrm_32_64	6346
k8	latnrm_8_1	93
k9	lmsfir_32_64	14689
k10	lmsfir_8_1	85
k11	mult_10_10	2365
k12	mult_4_4	223

Applications		
a1	G721_A	158921
a2	G721_B	357678
a3	V32.modem	1197325
a4	adpcm	90625
a5	compress	18407
a6	edge_detect	2248024
a7	histogram	215313
a8	lpc	34781
a9	spectral	84743
a10	trellis	2603

Table 4.2: Cycle counts for benchmarks executing on *UTDSP_parallel*.

4.3.1 Impact on Performance

Figures 4.1 and 4.2 show the relative performance of *DSP_mot* and *DSP_ad* for the kernel and application benchmarks, respectively. Figure 4.1 shows that the relative performance of *DSP_mot* and *DSP_ad* varies considerably among the kernels. For example, the relative performance of *DSP_mot* is lower than *UTDSP_serial* for the `fft` and `latnrm` kernels (k1, k2, k7, and k8). Similarly, the relative performance of *DSP_ad* is lower than that of *UTDSP_serial* for the `latnrm_32_64` kernel (k7). On the other hand, the relative performance of both *DSP_mot* and *DSP_ad* is nearly identical to *UTDSP_parallel* for `fir_256_64` (k3). Overall, the relative performance of *DSP_mot* varies between 0.15 – 0.99 (0.51 on average). On the other hand, the relative performance of *DSP_ad* varies between 0.25 – 0.99 (0.57 on average).

Figure 4.2 shows that the relative performance of *DSP_mot* and *DSP_ad* is generally poor across the application benchmarks. On all applications except `histogram` (a7), the relative performance of *DSP_mot* is lower than *UTDSP_serial*. Even for `histogram`, the relative performance of *DSP_mot* is only slightly better than that of *UTDSP_serial*. Similarly, the relative performance of *DSP_ad* on `G721_A` and `edge_detect` (a1 and a6) is lower than *UTDSP_serial*. On all other applications, the relative performance of *DSP_ad* is either marginally better or slightly better than *UTDSP_serial*. Overall, the relative performance of *DSP_mot* varies between 0.16 – 0.65 (0.35 on average), while the relative performance of *DSP_ad* varies between 0.25 – 0.72 (0.56 on average).

The poor performance of *DSP_mot* and *DSP_ad* compared to the UTDSP architecture is especially evident in the applications. This is due to the smaller number of data registers used in *DSP_mot* and *DSP_ad*, and to the restricted instances of parallelism that can be exploited by their instruction sets. The smaller

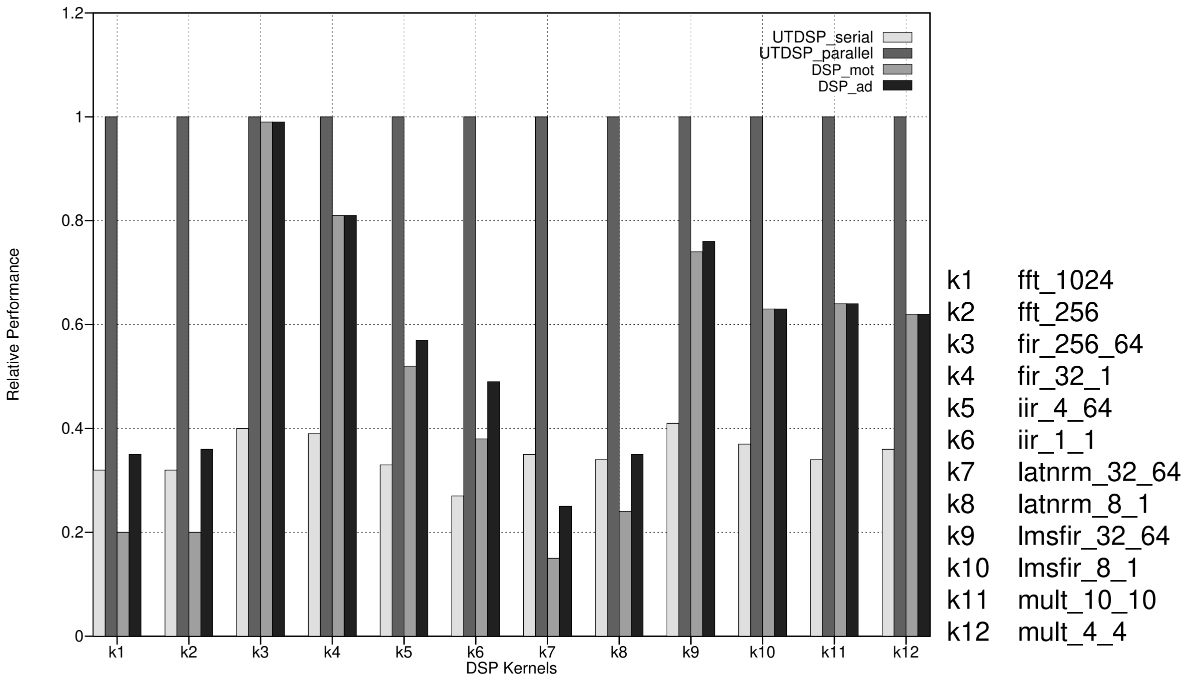


Figure 4.1: Relative performance of *DSP_mot* and *DSP_ad* on kernel benchmarks.

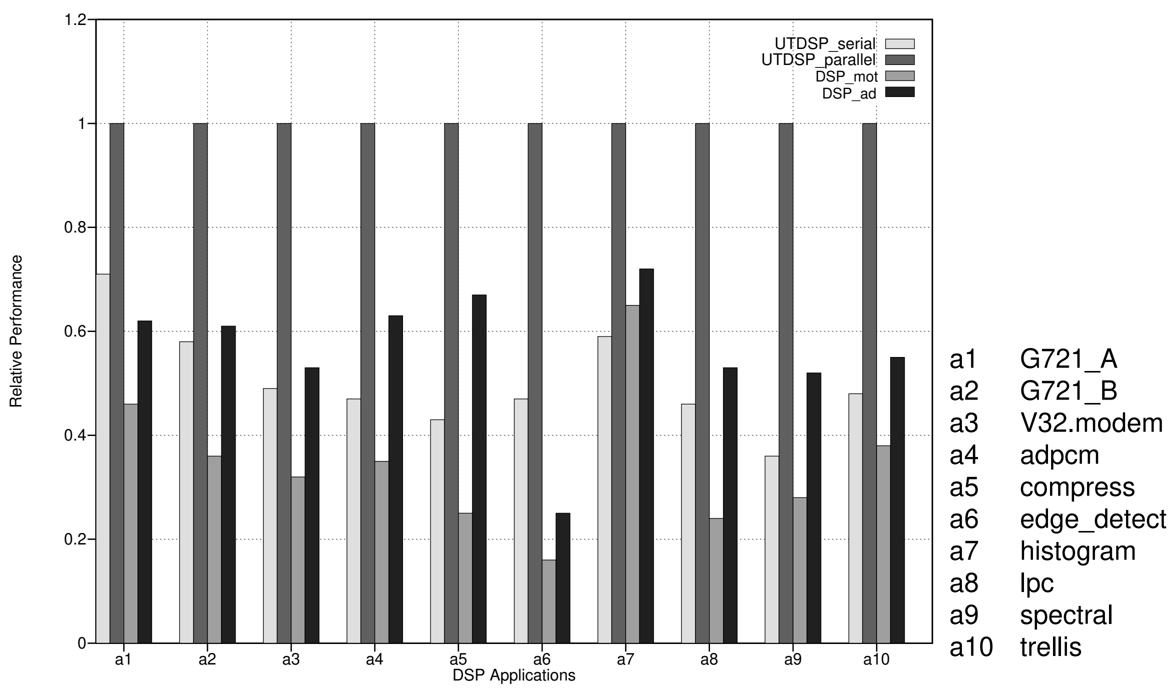


Figure 4.2: Relative performance of *DSP_mot* and *DSP_ad* on application benchmarks.

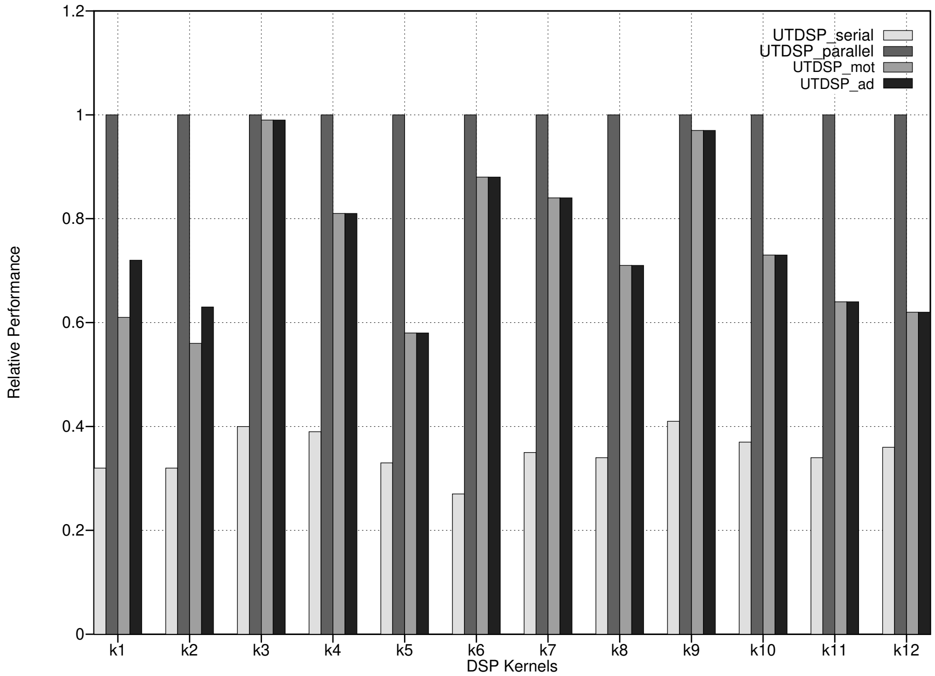
number of data registers increases the number of memory instructions needed in a program and this increases execution time. Furthermore, the limited instances of parallelism supported by the instruction sets does not always match the instances of parallelism that occur in the applications, and this causes both DSPs to miss opportunities to exploit other instances of parallelism in the code. One reason for the better overall performance of *DSP_mot* and *DSP_ad* on the kernels is the higher incidence of parallelism that matches the parallelism supported by their instruction sets. Finally, for both the kernel and application benchmarks, the higher performance of *DSP_ad*, relative to *DSP_mot*, is due to its larger number of data registers and to its more flexible instruction set that enables it to exploit more instances of parallelism. Recall, however, that since the UTDSP compiler does not restrict the use of registers in *DSP_mot* and *DSP_ad*, these performance results are more optimistic than what would be achieved in reality.

To study the effect of using restricted, and hence less costly, VLIW architectures on performance, the relative performance of *UTDSP_mot* and *UTDSP_ad* was measured. Recall that these architectures contain a subset of the functional units used in the full-fledged UTDSP architecture, and are therefore more restricted in the amount of parallelism they can exploit. However, both architectures use the same number of registers as the UTDSP architecture, and can therefore help the compiler generate more efficient code. Figures 4.3 and 4.4 show the relative performance results on the kernels and applications for *UTDSP_mot* and *UTDSP_ad*.

As expected, Figure 4.3 shows that the greater number of registers in *UTDSP_mot* and *UTDSP_ad* significantly increases their relative performance across all the kernels. Overall, the relative performance of *UTDSP_mot* varies between 0.56 – 0.99 (0.74 on average). Similarly, the relative performance of *UTDSP_ad* varies between 0.58 – 0.99 (0.76 on average). For all kernels except `fft_1024` and `fft_256`, *UTDSP_mot* and *UTDSP_ad* achieve identical levels of performance. This is due to both instruction-set architectures supporting similar instances of parallelism that match the ones occurring in the kernels.

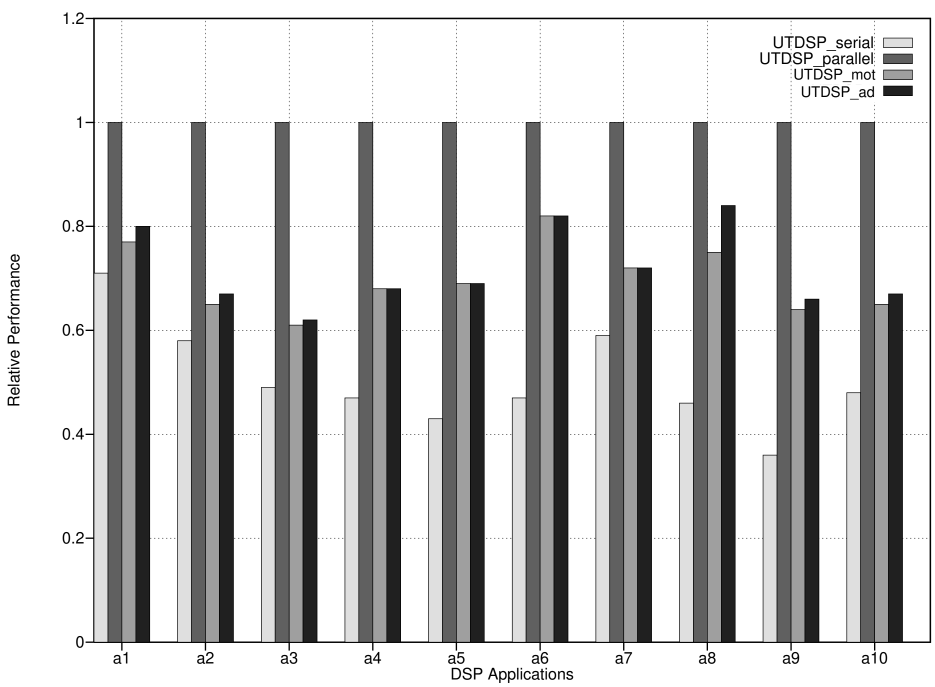
Figure 4.4 also shows that the greater number of registers in *UTDSP_mot* and *UTDSP_ad* enable them to achieve significantly higher performance across all applications. This improvement is mainly due to the elimination of additional memory instructions. The relative performance of *UTDSP_mot* across all applications varies between 0.61 – 0.82 (0.70 on average). The relative performance of *UTDSP_ad* varies between 0.65 – 0.86 (0.76 on average). These results also show that the restricted parallelism remains a limiting factor to attaining higher levels of performance even after the number of registers is increased. Finally, the results show that *UTDSP_ad* achieves slightly higher levels of performance than *UTDSP_mot* for most applications, and this is, once again, due to the greater flexibility in its instruction set architecture.

These results also show that, even with restricted parallelism, the greater flexibility of the VLIW architecture and the large number of registers result in significant performance improvements over more tradi-



k1	fft_1024
k2	fft_256
k3	fir_256_64
k4	fir_32_1
k5	iir_4_64
k6	iir_1_1
k7	latnrm_32_64
k8	latnrm_8_1
k9	lmsfir_32_64
k10	lmsfir_8_1
k11	mult_10_10
k12	mult_4_4

Figure 4.3: Relative performance of *UTDSP_mot* and *UTDSP_ad* on the kernel benchmarks.



a1	G721_A
a2	G721_B
a3	V32.modem
a4	adpcm
a5	compress
a6	edge_detect
a7	histogram
a8	lpc
a9	spectral
a10	trellis

Figure 4.4: Relative performance of *UTDSP_mot* and *UTDSP_ad* on the application benchmarks.

tional architectures. For example, the combined effect of these two factors enable the *UTDSP_mot* to execute an average of 1.5 – 2.0 times faster than the *DSP_mot*, and the *UTDSP_ad* to execute an average of 1.3 times faster than the *DSP_ad*. Recall, however, that since these results do not account for the limited orthogonality in the use of registers that exists in tightly-encoded instruction sets, they are much more conservative than what could be achieved with the real processors.

Finally, the results show that, given the opportunity to exploit more parallelism than what is typically supported in traditional DSP architectures, the VLIW architecture can achieve even higher levels of performance. For example, by using its full hardware resources, the UTDSP architecture can execute an average of 2.0 – 2.9 times faster than DSPs with architectures similar to *DSP_mot*, and 1.8 times faster than DSPs with architectures similar to *DSP_ad*. This performance, however, is achieved at the expense of higher cost. The next sub-section will examine the impact of using a VLIW architecture on cost.

4.3.2 Impact on Cost

Figure 4.5 and Table 4.3 show the area estimate of the UTDSP model architecture, in both its restricted and unrestricted VLIW forms, and the more traditional DSPs. Recall that the datapaths of the restricted VLIW UTDSP models are identical to the traditional DSPs, except they have more registers. The estimates are calculated for a feature size $\lambda = 0.25$ µm, and assuming there is no on-chip data memory in any of the models considered. Including the data memory area would only reduce the percentages, but would not change the conclusions because each processor would require the same amount of data memory. Finally, the size of instruction memory, in words, for each DSP is set to the average number of instructions generated for the application benchmarks.

The data in Table 4.3 shows that registers occupy a small percentage of total chip area for both traditional and VLIW-based DSPs. For the UTDSP model architecture, registers occupy only 3% of the area. Even when the hardware resources are constrained to restrict parallelism, the registers in *UTDSP_mot* and *UTDSP_ad* occupy 4% of the area. Similarly, for *DSP_mot* and *DSP_ad*, registers occupy 3% and 7% of the area, respectively. Since the impact on overall cost is small, using more registers helps the compiler produce more efficient code. However, since traditional DSPs use tightly-encoded instructions, they can only specify small register sets. On the other hand, the simple, RISC-like operations of the VLIW-based architectures can be used to specify larger register sets, which, in turn, results in more efficient compiled code.

Table 4.3 also shows that instruction memory in *DSP_ad* occupies an average of 9% more area than instruction memory in *DSP_mot*. This is due to the longer instruction width of *DSP_ad* (48 bits) compared to *DSP_mot* (24 bits), even though *DSP_ad* needs to store an average of 46% fewer instructions than *DSP_mot*. This highlights the trade-off that must be made when designing DSP ISAs. *DSP_ad* uses more

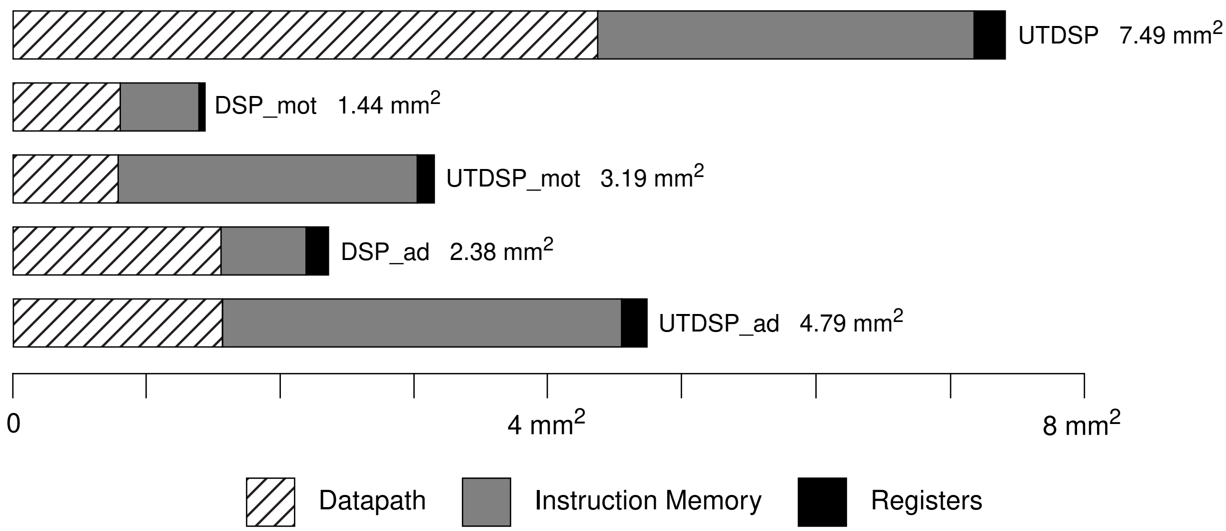


Figure 4.5: Area estimates for the different architectures assuming λ = 0.25 μm.

DSP	Total Area (mm^2)	Datapath			Instruction Memory			Registers		
		Funct. Units	Area (mm^2)	% Total Area	Ave. Instrs. Stored	Area (mm^2)	% Total Area	Regs Used	Area (mm^2)	% Total Area
UTDSP	7.49	9	4.39	59	436	2.86	38	96	0.24	3
DSP_mot	1.44	4	0.82	56	1068	0.59	41	20	0.04	3
UTDSP_mot	3.19	4	0.82	25	688	2.26	71	64	0.11	4
DSP_ad	2.38	7	1.58	66	576	0.63	27	64	0.17	7
UTDSP_ad	4.79	7	1.58	33	514	3.00	63	80	0.21	4

Table 4.3: Area estimates for λ = 0.25 μm.

registers and allows more flexibility in exploiting parallelism than *DSP_mot*. Although this enables it to achieve an average of 1.6 times the performance of *DSP_mot* on the application benchmarks, the higher performance is achieved at the expense of longer, less tightly-encoded instructions and larger memory area.

A similar trade-off may also be necessary when adopting a VLIW-based architecture such as the UTDSP. For example, even though the *UTDSP_mot* and *UTDSP_ad* achieve 1.3 – 2.0 times the performance of *DSP_mot* and *DSP_ad*, the memory needed to store their long instructions cause them to occupy 2.0 – 2.2 times the area. Similarly, the full-fledged UTDSP model architecture achieves 2 – 3 times the performance of *DSP_mot* and *DSP_ad*, but its larger datapath and instruction memory bank cause it to occupy 3.1 – 5.2 times their area. Whether the higher performance justifies the higher costs depends on the application, and

Architecture	Integer ALUs	Integer Mul	FP Adders	FP Mul	FP Div/Sqrt	Datapath Area (mm^2)	Normalized Area
UTDSP	2	2	2	2	2	4.39	1.00
DSP_mot/ UTDSP_mot	1	1	—	—	—	0.82	0.19
DSP_ad/ UTDSP_ad	1	—	2	1	1	1.58	0.36
Customized UTDSP	1	1	1	1	1	2.32	0.53

Table 4.4: Datapath components used in the different architectures assuming λ = 0.25 µm.

must be considered by the designer. However, since cost is an important consideration in embedded systems, it is clear that implementing a VLIW architecture in its classical form is not very cost-effective.

Table 4.3 showed that the major factor contributing to the difference in area between the VLIW-based UTDSP and the more traditional *DSP_mot* and *DSP_ad* is instruction memory requirements. Since most fields in a long instruction are typically empty — on average, 70% of the space consumed by the long instructions generated by the UTDSP compiler is occupied by NOPs — storing long instructions in an unencoded format is very wasteful of memory. This suggests that a significant reduction in the storage requirements of long instructions can be achieved by using more efficient instruction encoding techniques. In Chapter 5, a new method for storing and coding long instructions will be introduced. This method greatly reduces the amount of memory needed to store a program, without affecting the flexibility or orthogonality, and hence the performance, achieved with long instruction formats.

Another factor contributing to the difference in area is the organization of the datapath. Table 4.4 shows the organization and area of the full-fledged UTDSP, *DSP_mot*, *UTDSP_mot*, *DSP_ad* and *UTDSP_ad* datapaths, respectively. Though not shown in the table, each datapath uses a single program-control unit, two memory units, and two addressing units. From this table, it is clear that the greater area of the UTDSP datapath is due to its greater use of datapath components. This suggests that one way to reduce the area of the UTDSP datapath is to customize the datapath to the target application. In Chapter 6, a set of tools that can be used to customize the UTDSP datapath, and hence reduce its area, will be described. Table 4.4 shows the components used and area occupied by one such customized version of the UTDSP datapath. This particular example shows that using single integer and floating-point functional units, instead of pairs, reduces the UTDSP datapath area by 47%. Naturally, the choice of datapath components depends on trading-off the performance requirements and cost constraints of the target application. The tools described in Chapter 6 help the designer make these trade-offs.

By efficiently encoding long instructions and customizing the datapath to use a minimal set of components, VLIW architectures can provide the high performance and low costs required by embedded DSP applications. When coupled with their ease of programming using HLLs, VLIW architectures become an ideal choice for implementing embedded DSP systems. However, one factor that was not considered in this study, and that is an important criterion in designing embedded systems, is the power consumption of VLIW architectures. In general, power consumption is equal to the product of the system clock frequency, the square of the supply voltage, and load and parasitic capacitances. One way VLIW architectures help reduce power requirements is through exploiting fine-grained parallelism. This enables specific performance requirements to be met using lower system clock frequencies. Other standard ways to reduce power consumption include using lower voltage supplies, turning off datapath components that are not being used, and using clever circuit-design techniques. A further study is therefore needed to examine the power requirements of VLIW architectures, and assess their suitability for low-power embedded systems.

4.4 Summary

Programmable DSPs have traditionally been designed around tightly-encoded instruction sets. Since DSPs are mainly used in cost-conscious embedded systems, tightly-encoded instruction sets were used to improve code density, with the belief that this also reduced instruction memory requirements, and hence cost. Tightly-encoded instruction sets were also used to reduce instruction memory bandwidth, which was of particular concern when packaging was not very advanced. However, due to their tightly-encoded instruction sets, DSPs are difficult targets for optimizing high-level language compilers. For example, to save on the number of bits needed to encode instructions, most DSPs are accumulator-based and use very few registers. Furthermore, DSPs cannot exploit parallelism beyond that which is supported by their instruction sets. This makes it difficult for a compiler to generate efficient code and fully exploit available parallelism. Thus, to achieve the most efficient and the most compact code, DSPs must still be programmed in assembly language. However, as DSP applications become more sophisticated, and as design and development cycles become shorter, it is becoming more difficult and more costly to develop applications in assembly. That is why DSP vendors are currently considering new instruction set architectures that can provide the high performance and low costs demanded by embedded DSP applications, while also being good targets for optimizing HLL compilers.

One architectural style that is particularly well suited for embedded DSP applications is the VLIW architecture. VLIW architectures provide the hardware resources and the instruction set flexibility that enable the exploitation of high levels of parallelism. They are also easy targets for optimizing HLL compilers, and make it easy for the compiler to generate efficient code.

This chapter examined the performance benefits and cost trade-offs of using the VLIW-based UTDSP architecture compared to more tightly-encoded DSPs. By constraining the functional units used in the

UTDSP model architecture, two commercial DSPs, based on the Motorola DSP56001 and the Analog Devices ADSP-21020, were modeled. The compiler was also modified to only generate combinations of parallel operations that are supported by the instruction sets of both architectures, and to use the number of registers available in each architecture. However, the orthogonality between available registers and machine operations was not constrained, leading to more optimistic performance results than could actually be achieved on these architectures.

To enable a fairer comparison to be made between the hardware-limited traditional DSP architectures and the UTDSP model architecture, two instances of the UTDSP architecture were modeled with the same hardware resources as the traditional DSPs. The main differences were that the VLIW-based models were given more registers, they were allowed more flexibility in exploiting parallelism, and they used long instructions instead of tightly-encoded ones.

The results showed that, for the kernel and application benchmarks, the restricted UTDSP architectures achieve 1.3 – 2.0 times the performance of the more traditional architectures. This is mainly due to their more flexible instruction set architecture and their larger set of data registers, which enable the compiler to exploit more parallelism and generate more efficient code. The results of an experiment measuring the effect of restricted register use in traditional DSP architectures also shows that the impact on performance is a factor of 1.4 – 2.0, suggesting that the actual performance achieved by the restricted UTDSP architectures is at least 1.8 – 2.8 times the performance of the more traditional architectures. However, since the experiment involved kernel code, the effect of more restrictive register use is expected to be more significant in larger applications. When the full-fledged, unrestricted UTDSP architecture is used, it achieves 2 – 3 times the performance of the more traditional architectures. When accounting for the effect of more restricted register use, the actual performance is likely closer to 2.8 – 4.2 times that of the more traditional architectures.

The results also showed that the restricted UTDSP architectures occupy 2.0 – 2.2 times the area of the more traditional architectures, while the unrestricted UTDSP architecture occupies 3.1 – 5.2 times their area. This was mainly due to the large memory needed to store long instructions and, in the case of the unrestricted UTDSP architecture, to its larger datapath. Thus, in their classical form, VLIW architectures are too expensive to use in cost-conscious, embedded DSP applications. However, as Chapters 5 and 6 demonstrate next, the cost of a VLIW architecture can be reduced by efficiently encoding and storing long instructions, and by customizing the datapath to the functional and performance requirements of the target application.

Chapter 5

Long-Instruction Decoding and Encoding for the UTDSP

VLIW architectures are well-suited for implementing embedded, application-specific programmable processors (ASPPs). The long-instruction format, for example, is ideal for exploiting high levels of parallelism, such as those commonly found in embedded DSP applications. This not only improves execution performance, but can also be used to reduce power consumption — an increasingly important factor in portable embedded systems — by using a low-frequency system clock. The long-instruction format also provides great flexibility in synthesizing application-specific instructions that can be tuned to match the functional or performance requirements of one or more target applications. VLIW architectures also rely on optimizing compilers to expose parallelism and schedule the long instructions. This simplifies the hardware, and helps reduce system costs. Finally, VLIW architectures are easy targets for optimizing, high-level language (HLL) compilers, making them easy to program using a high-level language. This reduces the code-development time, makes the code easier to debug and maintain, and reduces time-to-market cycles. However, the major drawbacks of VLIW architectures are their large instruction storage and bandwidth requirements. That is why, and until very recently, VLIW architectures have been regarded as too impractical and expensive to implement as single-chip processors.

This chapter presents a solution to the problem of storing and fetching long instructions for the UTDSP architecture. The solution is based on the idea of a two-level control store, first used in early microprogrammed computers to support machine emulation. For the UTDSP, a first-level instruction memory is used to store uni-op instructions and pointers to multi-op instructions. This greatly reduces the storage requirements of instruction memory and the bandwidth needed to access it. Once a pointer is fetched from instruction memory, it is used to access a second-level decoder memory that is used to store the multi-op instructions. To reduce the storage requirements of decoder memory, these instructions are stored in a packed format. However, once fetched from decoder memory, they are issued to the datapath in a long-instruction format. This simplifies the instruction-decoding circuitry, and preserves the flexibility of the long-instruction format to exploit parallelism and synthesize application-specific instructions.

The remainder of this chapter is divided into four sections. Section 5.1 describes past and present VLIW architectures, and examines the techniques they use for storing and fetching long instructions. It also assesses the strengths, weaknesses, and cost-effectiveness of these techniques in the context of a single-chip, embedded processor. Section 5.2 describes the basic design of the long-instruction decoder for the UTDSP. In addition to describing the two-level storage scheme in greater detail, it describes techniques for efficiently storing multi-op instructions in decoder memory. It also presents two algorithms developed for packing multi-op instructions in decoder memory. Finally, this section examines the impact of the long-instruction decoder on the performance of the pipeline. Section 5.3 describes the different phases of the encoding pass, which is responsible for processing the long-instructions generated by the operation-compaction pass in the compiler and for storing them in an efficient manner in both instruction and decoder memory. The encoding pass implements the algorithms described in Section 5.2. Finally, Section 5.4 analyzes the impact of the long-instruction decoder on bandwidth requirements, instruction storage requirements, and the execution performance of the kernel and application benchmarks.

5.1 Long-Instruction Architectures: Past and Present

Before describing the techniques used to store and fetch long instructions in the UTDSP, it is useful to examine the techniques used in past and present long-instruction architectures. Of particular interest are the ways in which these techniques affect storage and bandwidth requirements, and how they preserve the flexibility and scalability of the long-instruction format. It is also interesting to examine the cost-effectiveness of these techniques when applied to embedded processors.

5.1.1 Microprogrammed Computers

The earliest examples of long instructions are the microinstructions used in microprogrammed computers. In a microprogrammed computer, each machine instruction is executed by a microroutine consisting of one or more microinstructions. The microinstructions are stored in a special memory called the *control store*, and each microinstruction could be used to specify one or more microoperations. Since memory was an expensive resource in the 1960's and 1970's, there was a great deal of interest in reducing the size of control stores, and one way for achieving this was to encode microinstructions efficiently.

There are several ways to encode microinstructions. In their most basic form, microinstructions are horizontally microcoded — also referred to as *direct-control encoded*. In this encoding, every control signal is specified by a distinct bit in the control word. Although this provides the most flexibility and supports the exploitation of the most parallelism, it is also the most inefficient and the most expensive in terms of its storage requirements. That is why direct-control encoding was seldom used in practice. Another way of encoding microinstructions is to use vertical microcoding — also referred to as *maximal encoding* — where the microinstructions are enumerated, and each microinstruction is assigned a distinct code. Although this encoding results in the most compact microinstructions, it is also the least flexible because it requires a dedicated decoder that makes it difficult to modify existing microinstructions or to add new ones. For this reason, maximal encoding was also seldom used in practice. The microinstruction encoding that balances the flexibility of direct-control with the compactness of maximal encoding is called *minimal encoding*. In a minimally-encoded microinstruction, each control word is divided into a fixed number of fields that are each used to maximally encode a group of mutually exclusive microoperations. Most of the research into control store design in the 1970's was focused on the minimal encoding of microinstructions, and on ways of identifying mutually exclusive groups of microoperations [49]. Because of the balance it struck between flexibility and compactness, minimal encoding became widely used.

Another approach to encoding microinstructions is to use a two-level control store. This approach was mainly used to support machine emulation, where microinstructions are used to emulate the execution of machine instructions for different instruction-set architectures. In this scheme, a first-level store is used to store short microinstructions that are used as pointers into a writable, second-level, nanostore. The nanostore is used to store nanoinstructions that are usually direct-control encoded. This enables them to exploit the most parallelism, and provides the most flexibility in specifying different combinations of nano-operations. Since the nanostore is writable, it can be easily programmed with the appropriate mix of nanoinstructions. Two machines that have been designed with two-level control stores are the Burroughs B700 Interpreter [50] and the Nanodata QM-1 [51].

5.1.1.1 Burroughs B700 Interpreter

In the Burroughs Interpreter, the first-level store is used to store two kinds of 16-bit microinstructions: Type-I instructions are used to specify pointers into the second-level store, while Type-II instructions are used to control the datapath directly. The second-level store is used to store 54-bit nanoinstructions. Each nanoinstruction specifies multiple operations, control signals, and conditions that can be used to control the underlying datapath directly. Since the nanostore is writable, the Interpreter can easily be tuned to its target architecture.

5.1.1.2 Nanodata QM-1

In the Nanodata QM-1, the first-level store is also used to store 16-bit microinstructions. Each microinstruction contains a field that is concatenated with the contents of a control register to form an address used to access the second-level nanostore. The nanostore is used to store 342-bit nanoinstructions. Each nanoinstruction contains a 38-bit *K-field* used to specify control signals and conditions. The K-field can also be used to specify a branch to another nanoinstruction. Each nanoinstruction also contains four, 76-bit *T-fields* that are each used to specify different nanoinstructions. The T-fields could either be executed sequentially or as a loop, and this enabled the definition of very powerful microinstructions.

For both the Interpreter and the QM-1, the two-level control store enables the synthesis of very powerful microinstructions. Since these use fewer bits than the nanoinstructions, they could be stored in smaller memories, and they require less bandwidth. However, in both the Interpreter and the QM-1, no attempts are made to minimize the storage requirements of nanoinstructions within the nanoprogram memory. For large nanostores this can result in significant amounts of wasted space.

5.1.2 Classical VLIWs

Two of the first commercial computers designed around a VLIW architecture were the Multiflow Trace 7/300 [52] and the Cydrome Cydra-5 [53]. Both computers appeared in the late 1980's and were classical VLIWs in that they consisted of multiple functional units that were each controlled by a specific field in a long-instruction word. Both computers also relied on very sophisticated optimizing compilers to expose instruction-level parallelism and schedule long-instructions. Although neither computer was a commercial success, it is still useful to study their designs and the way they stored and fetched long instructions. The AT&T CRISP microprocessor [54] also uses interesting techniques for reducing instruction storage and bandwidth requirements in main memory, and exploiting limited parallelism.

5.1.2.1 Multiflow Trace 7/300

The Multiflow Trace 7/300 consisted of a pair of integer and floating-point units that could execute up to seven operations every clock cycle. These operations were encoded into 256-bit long instructions. The architecture could also be expanded to include either two or four pairs of integer and floating-point units, in which case the instruction length would increase to 512 bits or 1024 bits, respectively.

Instructions in the Trace 7/300 were fetched from a distributed instruction cache that could hold up to 8K long-instructions. Since the integer and floating-point units were built as separate printed-circuit boards, instruction fields were cached on different boards. On a cache miss, instructions were fetched from main memory, which consisted of eight separate banks that were eight-way, low-order interleaved. This provided the high instruction bandwidth necessary for loading the cache quickly.

To reduce storage requirements in main memory, instructions were stored in a compressed format whereby NOPs were simply not stored. A special header was also stored for each instruction to identify the fields to which the instructions stored in memory correspond. When instructions were being fetched from main memory, a dedicated controller was used to read the header and route the different instruction fields to their appropriate slots in the cache word.

Although the Trace 7/300 used several techniques to reduce instruction storage requirements and provide the bandwidth necessary for fetching long instructions quickly, these techniques might be too costly or too complex to implement in a single-chip, embedded processor. For example, although the Trace 7/300 stored its instructions in a compressed format in main memory, the dedicated controller used to route instruction packets to their appropriate slots in the cache adds to the complexity and the cost of the processor. Furthermore, although storing instructions in the cache in their long-instruction formats simplifies instruction decoding, it also increases the size, and hence the cost, of the cache. Finally, although using an interleaved memory provides the high bandwidth needed to load the cache quickly, it may not be practical, cost-effective, or even necessary for an embedded processor.

5.1.2.2 Cydrome Cydra-5

The Cydra-5 contained seven functional units that could execute up to seven operations per cycle. These operations were encoded into 256-bit long instruction words. Instructions for the Cydra-5 were fetched from an instruction cache that could store up to 1000 long instructions. The cache was loaded from main memory, which also consisted of eight separate banks that were eight-way, low-order interleaved to provide high instruction bandwidth.

To reduce instruction storage requirements in both main memory and the instruction cache, two different instruction formats were used. The first format was called *MultiOp* and was used to encode up to seven operations that could be executed in parallel. The MultiOp format was used when there was a great deal of instruction-level parallelism that could be exploited, such as that found in the inner loops of scientific applications. The second format was called *UniOp* and was used to store six instructions that were executed sequentially. Since the UniOp format did not place any restrictions on the type of instructions that could be stored in its different slots, a crossbar switch was needed to route each instruction to the appropriate functional unit. The UniOp format was used when there was less parallelism to exploit, and this resulted in a dense encoding of instructions.

Using the MultiOp and UniOp formats enables the Cydra-5 to make efficient use of its memory resources. However, it still requires the use of an interleaved memory to provide the high bandwidth necessary to quickly load the cache from main memory, and this may not be practical or cost-effective in an embedded processor. Moreover, when the UniOp format is used, a crossbar switch is needed to route the instructions in the different slots to the appropriate functional units. Since the area of the crossbar switch increases as a quadratic function of the number of inputs, its impact on cost will increase significantly as the width of the instruction is increased to accommodate more functional units.

5.1.2.3 AT&T CRISP

Though not based on a VLIW architecture, the AT&T CRISP microprocessor can execute arithmetic/logic and branch operations simultaneously. Initially, instructions are stored in main memory as one, three, or five halfwords. A *prefetch/decode unit* is used to fetch instructions from main memory and store them in a 512-byte *prefetch buffer*. The prefetch/decode unit is also used to convert the variable-length instructions into 192-bit instructions, and store them in a 32-entry *decoded instruction cache*. The instructions in the decoded instruction cache are then executed by the execution unit.

Each 192-bit decoded instruction consists of three main parts: an operation field, a next-PC field — used to specify the address of the following instruction, and an alternate next-PC field — used to specify the target address of a branch instruction. A branch-prediction bit, set by the compiler, is also used to predict the outcome of a branch instruction. When the prefetch/decode unit identifies a branch instruction, it folds it with the previous arithmetic/logic instruction, and stores them in the same 192-bit decoded instruction. This helps hide the latency associated with the branch.

Having a separate instruction prefetch buffer and decoded instruction cache allows the prefetch/decode unit to operate independently of the execution unit. Using variable-length instructions in main memory also results in high code density, and reduces instruction storage and bandwidth requirements. However, with the exception of folded branches, the CRISP can only execute a single instruction per cycle. To exploit the high levels of parallelism commonly found in embedded DSP applications, the prefetch/decode unit will have to find more parallel instructions in the prefetch buffer, and store them in longer decoded-instruction words. This would not only increase the cost and complexity of the prefetch/decode unit — similar to the control units of contemporary superscalar processors — but would also increase the bandwidth requirements of main memory and the size of the decoded instruction cache. Moreover, since the CRISP uses two cache memories — the prefetch buffer and the decoded instruction cache — it may not be able to satisfy the stringent requirements for deterministic instruction access times of certain hard real-time embedded applications.

5.1.3 Contemporary VLIWs

More recently, a number of VLIW-based media processors have emerged in the embedded-multimedia and PC-multimedia-accelerator markets. These processors are designed to be used in stand-alone systems, such as set-top boxes or DVD players, or for off-loading the processing of high-fidelity audio, 3D graphics, or full-motion video from a host processor. Examples of these processors include the Philips TriMedia TM-1 [7],[8], the Chromatic Mpact/3000 [9], and the IBM Mfast [10]. Another recent VLIW-based processor is the Texas Instruments TMS320C6201 [11],[12]. The ‘C6201 is aimed at the telecommunication market, and is intended for applications that currently require a group of DSPs working together. Examples of such applications include central-office switches, modem banks, and cellular base stations. This section briefly describes the architectures of these processors, and the methods they use for storing and fetching long-instructions.

5.1.3.1 Philips TriMedia TM-1

The Philips TriMedia TM-1 contains 27 functional units, and its instructions are 160 bits long. Each instruction can store up to five, 32-bit operations, enabling the TM-1 to execute up to five operations in parallel. Although most operations are simple and RISC-like, others are SIMD-like. Such operations use a single opcode to specify multiple operations, and are useful for implementing application-specific functions such as MPEG decoding.

Instructions are fetched from a 32 Kbyte instruction cache. To reduce memory requirements in both the instruction cache and main memory, instructions are stored in a compressed format. In addition to using SIMD-type operations, compression is achieved by removing all NOPs, and by encoding the most frequently used operations with the fewest bits. A header is also stored with each compressed instruction to identify the number of fields used, and the encoding used for each field. Instructions are decompressed by special decompression hardware as they are fetched from the instruction cache. Once an instruction has been decompressed, its operations are routed to the appropriate functional units using a crossbar switch. To provide the high bandwidth needed to load the cache quickly, main memory is implemented using synchronous DRAM (SDRAM). The TM-1 uses a 32-bit bus to interface with main memory, and the bus implements a burst packet protocol to match the speed of the SDRAM.

Like the Trace 7/300, the TM-1 reduces storage and bandwidth requirements by using compressed instructions. To decompress these instructions and route operations to the appropriate functional units, the TM-1 requires special hardware and a crossbar switch. The additional hardware increases the complexity and the cost of the chip, and adding more functional units or increasing the operation issue width becomes very costly. Finally, although using SDRAM to provide high instruction bandwidth might be reasonable for PC-based applications, it might still be too expensive for portable or hand-held devices.

5.1.3.2 Chromatic Mpact/3000

The Chromatic Mpact/3000 is both a VLIW and a SIMD processor. Like a VLIW processor, it can issue up to two instructions per cycle, and like a SIMD processor, some of its instructions can specify a group of identical operations. The Mpact/3000 contains five functional units called *ALU groups*. Each group consists of a functional unit that can perform a single operation on a pair of operands, or a group of identical operations on two, four, or eight pairs of operands.

Instructions are fetched from a 4 Kbyte unified cache that can be configured to store 32, 64, or 128 long-instruction words. Each long-instruction word is 72-bits wide, and can be used to specify two, 36-bit instructions. Since the instructions specified in a long-instruction word can be executed sequentially or in parallel, good code density is achieved. To provide the high bandwidth necessary to load the cache quickly, the cache is connected to main memory through a 72-bit Rambus interface [55]. Main memory is implemented as 2 Mbytes of RDRAM.

Although the techniques used in the Mpact/3000 to achieve high code density and instruction bandwidth might be reasonable for PC-based systems, they are too complex and costly for embedded systems. For example, to increase the issue width of the Mpact/3000, a wider long-instruction word, a larger cache, and a wider Rambus interface are required. In fact, the recently announced Mpact 2 will be able to issue four instructions in parallel, but will require 144-bit long-instruction words, an 8 Kbyte cache, and two Rambus channels [56].

5.1.3.3 IBM Mfast

The IBM Mfast consists of a 4×4 mesh of 32-bit processing elements, and is aimed at high-throughput applications such as image and video processing. Every four processing elements execute identical code sequences by sharing a common 32-bit instruction bus used to fetch instructions from a 1 Kbyte instruction cache. In addition to executing 32-bit single-operation instructions, processing elements can execute 160-bit encapsulated VLIW (eVLIW) instructions capable of specifying up to five parallel instructions. Within each processing element, eVLIW instructions are stored in local, 16-entry, writable VLIW instruction memories (VIMs). Encapsulated VLIW instructions are dispatched from the VIMs using special surrogate execute-VLIW-indirect instructions. These are also fetched from the shared instruction cache, and are mainly used to specify an offset into the local VIMs.

Like the Burroughs Interpreter and the Nanodata QM-1, each processing element in the IBM Mfast uses a two-level instruction store. While the shared instruction cache represents the first-level store, the local VIMs represent the second-level store. Although each VIM can only store 16 eVLIW instructions at any given time, the eVLIW instructions can be customized to the functional and performance requirements of a target application. The VIMs also help reduce instruction storage and bandwidth requirements by substituting eVLIW instructions with the appropriate surrogate instructions. The VIMs are also easy to scale, and can support eVLIW instructions of different lengths. However, since it may not always be possible to exploit the full parallelism supported by an eVLIW instruction, a significant amount of the space occupied by the VIMs may be wasted on NOPs. Given the small size each VIM, this may not currently be a problem. However, as the size of the VIMs is increased, the amount of wasted space will also increase.

5.1.3.4 Texas Instruments TMS320C6201

The TMS320C6201 has eight functional units, and it can execute up to eight, RISC-like instructions every clock cycle. The instructions in the ‘C6201 are analogous to the operations in a classical VLIW, and each instruction is encoded in a 32-bit instruction word. Instructions are fetched from a 64 Kbyte on-chip instruction memory that can also be configured as a direct-mapped cache. The on-chip program memory has a 256-bit path into the ‘C6201 datapath, making it possible to fetch eight instruction words — called a *fetch packet* — in a single cycle. For programs that are too big to fit in the on-chip instruction memory, instructions must be fetched, one at a time, from external memory over a 32-bit bus.

A fetch packet may contain several *execution packets*. These are groups of instructions that can be executed in parallel. An execution packet may consist of a single instruction or as many as eight instructions. Instructions can easily be linked to form execution packets by setting the least-significant bits of their instruction words. An execution packet is therefore a variable-length long instruction, and this greatly reduces the instruction storage requirements in both the memory and the fetch packets. However, NOPs are still needed to pad fetch packets that cannot be completely filled with execution packets. Execution packets are issued to the datapath one at a time. Since the instructions in an execution packet are not position dependent, the ‘C6201 uses a crossbar switch to route instructions to the appropriate functional units. Only after all execution packets in a fetch packet have been issued will a new fetch packet be fetched from on-chip instruction memory.

The ‘C6201 uses a number of features that reduce instruction storage and bandwidth requirements. These include the variable-length execution packets, which reduce the space needed to store instructions, and minimizes the space wasted on NOPs. They also include the on-chip instruction memory, which provides high instruction bandwidth through its 256-bit path. However, if a program is too large to fit in the on-chip instruction memory, instructions must be fetched over a 32-bit bus, and this can significantly degrade execution performance. Finally, since the ‘C6201 uses a crossbar switch, adding new functional units or increasing the issue width of an execution packet beyond eight instructions can significantly increase overall system costs.

5.2 Storing and Decoding Long Instructions

Like the nanostores and the VIM used in the Burroughs Interpreter, Nanodata QM-1, and IBM Mfast, respectively, the UTDSP uses a *decoder memory* to implement a two-level instruction store. The first-level instruction memory is used to store uni-op instructions and pointers to multi-op instructions. By storing and fetching multi-op pointers instead of multi-op instructions, the storage and bandwidth requirements of instruction memory are reduced. This is especially advantageous if instruction memory is implemented off-chip. The second-level decoder memory is used to store multi-op instructions in a packed format, and supply these instructions to the datapath in a long-instruction format. This reduces the cost of decoder memory while preserving the flexibility of the long-instruction format, and simplifying the instruction decode logic.

Since decoder memory is easy to scale, it can support different architectural configurations requiring different long-instruction formats. Furthermore, since decoder memory is programmable, it is capable of supporting the synthesis of application-specific instructions that can be tuned to the functional and performance requirements of the target application. For these reasons, decoder memory is the basis for the long-instruction decoder. The remainder of this section describes the design of the instruction decoder and the algorithms used to store and pack multi-op instructions in decoder memory. It also examines the impact of the decoder on the pipeline.

5.2.1 Basic Decoder Design

Figure 5.1 shows a conceptual diagram of the instruction decoder for the UTDSP architecture. The decoder is built around *M* memory banks — *M* is the number of functional units used in the architecture — collectively referred to as the *decoder memory.* Each decoder-memory bank is associated with one of the functional units in the architecture. Since every functional unit in a VLIW architecture is controlled by a specific field in a long-instruction word, each decoder-memory bank is used to store the operations that will be specified by the corresponding field. As such, the long-instruction word consists of *M* fields that are distributed among the decoder-memory banks. Nonetheless, a multi-op instruction can be supplied to the rest of the processor in a long-instruction format by using a multi-op pointer to access specific decoder-memory banks simultaneously. As the remainder of this section explains, distributing the fields of the long-instruction word across multiple memory banks enables the efficient packing of multi-op instructions in decoder memory without compromising the flexibility of the long-instruction format.

When a word is fetched from instruction memory, its most-significant bit is examined to determine if the word represents a uni-op instruction or a multi-op pointer. If it is a uni-op instruction, it is issued to an appropriate functional unit where it is further decoded. Since it is possible for several functional units to execute the same operation, conflicts are avoided by designating specific functional units to execute specific types of operations. On the other hand, if the word contains a multi-op pointer, the pointer is used to access the appropriate locations in decoder memory and fetch the corresponding multi-op instruction. Since each memory bank is associated with a specific functional unit, the corresponding operations are for-

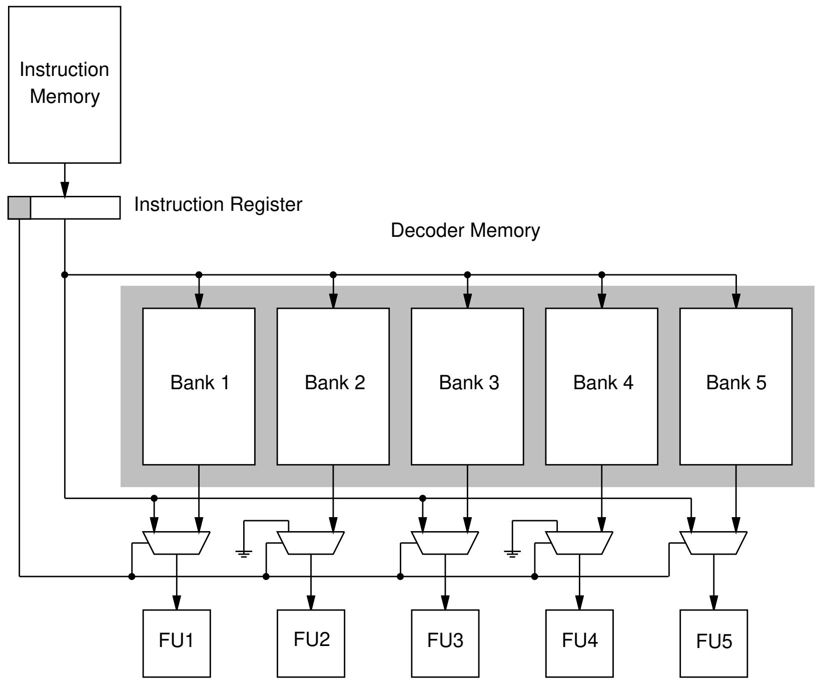


Figure 5.1: Block diagram of the long-instruction decoder.

warded to the appropriate functional units for further decoding and eventual execution. Although Figure 5.1 shows the decoder memory consisting of different banks, it can also be implemented as a single, wide-memory bank. In this case, the necessary control circuitry must be added to ensure that only the appropriate portions of a wide-memory word are issued to the datapath.

5.2.2 Reducing the Size of Decoder Memory

The size, and hence the cost, of decoder memory is directly related to the way multi-op instructions are stored inside of it. Since the main reason for introducing decoder memory is to minimize instruction storage requirements, it would be very wasteful to store multi-op instructions in their original, long-instruction formats. That is why multi-op instructions must be stored in a more efficient manner to minimize the size of decoder memory.

Since multi-op instructions are represented by multi-op pointers in a program, the actual order in which they are stored in decoder memory is not important. This suggests that one way to reduce the overall size of decoder memory is to exploit the order in which multi-op instructions are stored in decoder memory. Figure 5.2 demonstrates how the order in which multi-op instructions are stored helps reduce the size of individual decoder-memory banks, and hence, the entire decoder memory.

Figure 5.2 (a) shows seven multi-op instructions stored in an arbitrary manner in a decoder memory. In this case, the decoder memory consists of four banks, and the corresponding long-instruction word con-

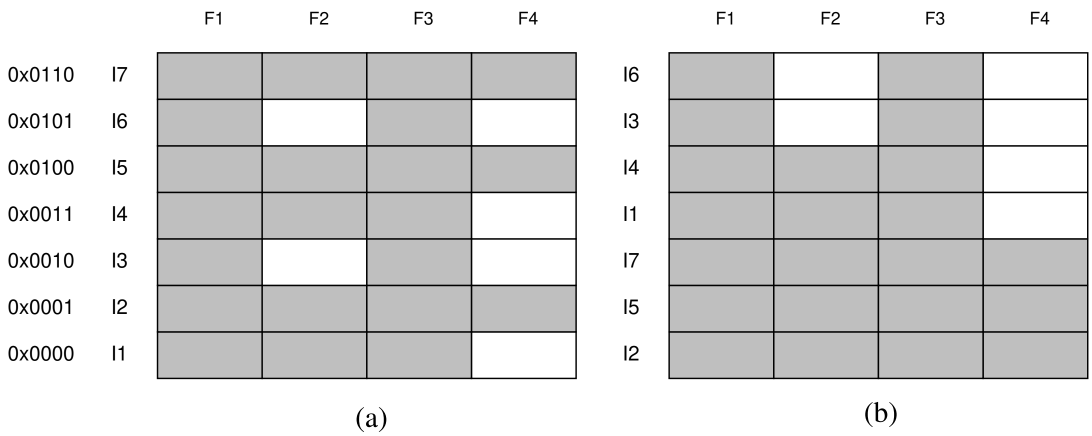


Figure 5.2: (a) Multi-op instructions stored in an arbitrary manner. (b) Multi-op instructions stored according to field priorities.

sists of four fields: F1, F2, F3, and F4. Furthermore, each multi-op instruction is stored in a separate long-instruction word. For example, instruction I1 is stored in the long-instruction word at address 0x0000, and uses fields F1, F2, and F3. Since the fields are distributed across four decoder-memory banks, the actual operations are stored at address 0x0000 of the decoder-memory banks associated with fields F1, F2, and F3, respectively. Since I1 does not use field F4, a NOP is stored at address 0x0000 in the decoder-memory bank associated with field F4. On the other hand, instruction I6 is stored in the long-instruction word at address 0x0101, and uses fields F1 and F3. Its operations are therefore stored at address 0x0101 in the decoder-memory banks associated with fields F1 and F3, while NOPs are stored at address 0x0101 in the decoder-memory banks associated with fields F2 and F4. By storing the multi-op instructions in the arbitrary manner shown in Figure 5.2 (a), the decoder memory requires 28 words of memory, of which six are NOPs.

Ideally, the minimum size of decoder memory, in words, would be equal to the total number of useful operations used in the multi-op instructions. From this, it follows that the minimum size of each decoder-memory bank is equal to the number of useful operations associated with its field. For example, since all the multi-op instructions in Figure 5.2 (a) use fields F1 and F3, the corresponding decoder-memory banks occupy seven words each, all of which are used to store useful operations. Typically, however, the size of a decoder-memory bank will be greater than the number of useful operations associated with its field. For example, even though only three multi-op instructions — I2, I5, and I7 — use field F4 in Figure 6.2 (a), the corresponding decoder-memory bank still requires seven words to store the three useful operations. The remaining four words are used to store NOPs.

A careful examination reveals that the size of a decoder-memory bank depends on the number of useful operations associated with its field, and the order in which the multi-op instructions that use these operations are stored in decoder memory. As the number of multi-op instructions stored in decoder memory increases, the decoder-memory banks associated with fields containing relatively few operations will tend to suffer the most from the effects of poor multi-op instruction ordering. This suggests that the storage requirements of individual decoder-memory banks, and hence, the entire decoder memory, can be reduced by storing multi-op instructions that use fields with relatively few operations at lower address locations. However, this requires that a priority be assigned to each field, such that the priority is inversely proportional to the number of useful operations associated with the field. As such, the field with the least number of useful operations is assigned the highest priority, while the field with the most number of useful operations is assigned the lowest priority. Using the priorities assigned to the different fields, an order can be imposed on the multi-op instructions stored in decoder memory: instructions with operations associated with higher-priority fields are stored first, while instructions with operations associated with lower-priority fields are stored last.

Figure 5.2 (b) shows the same group of multi-op instructions stored according to the order imposed by the priorities assigned to the different fields. Since field F4 contains the least number of operations, it is assigned the highest priority, and instructions I2, I5, and I7 are stored first. Field F2 is assigned the next highest priority, and instructions I1 and I4 are stored next. Finally, fields F1 and F3 are assigned the lowest priority, and instructions I3 and I6 are stored last. By storing the instructions in this order, the size of decoder memory is reduced to 22 words. Although the decoder-memory banks associated with fields F2 and F4 contain unused space, this space is not "wasted" on storing NOPs. In fact, for highly customized implementations, the unused space can be removed to further reduce costs. Thus, by being careful about the order in which multi-op instructions are stored in decoder memory, the storage requirements of individual decoder-memory banks, and hence, of the entire decoder memory, can be reduced.

In addition to paying attention to the order in which multi-op instructions are stored in decoder memory, another way of making more efficient use of decoder-memory space is to pack multi-op instructions with mutually exclusive operation fields into the same long-instruction word. Figure 5.3 (a) shows an example of three multi-op instructions stored in three, separate, long-instruction words. When stored in this format, the instructions occupy 27 words of memory, of which 19 are NOPs. On the other hand, Figure 5.3 (b) shows the same instructions packed into a single, long-instruction word. In this packed form, the instructions occupy only nine words of memory, of which only one is a NOP. Packing several, mutually exclusive, multi-op instructions into a single, long-instruction word can therefore reduce the size of decoder memory substantially.

To enable the extraction of specific multi-op instructions from a packed long-instruction word, the multi-op pointer should include a bit mask that can be used to select specific banks from decoder memory. The

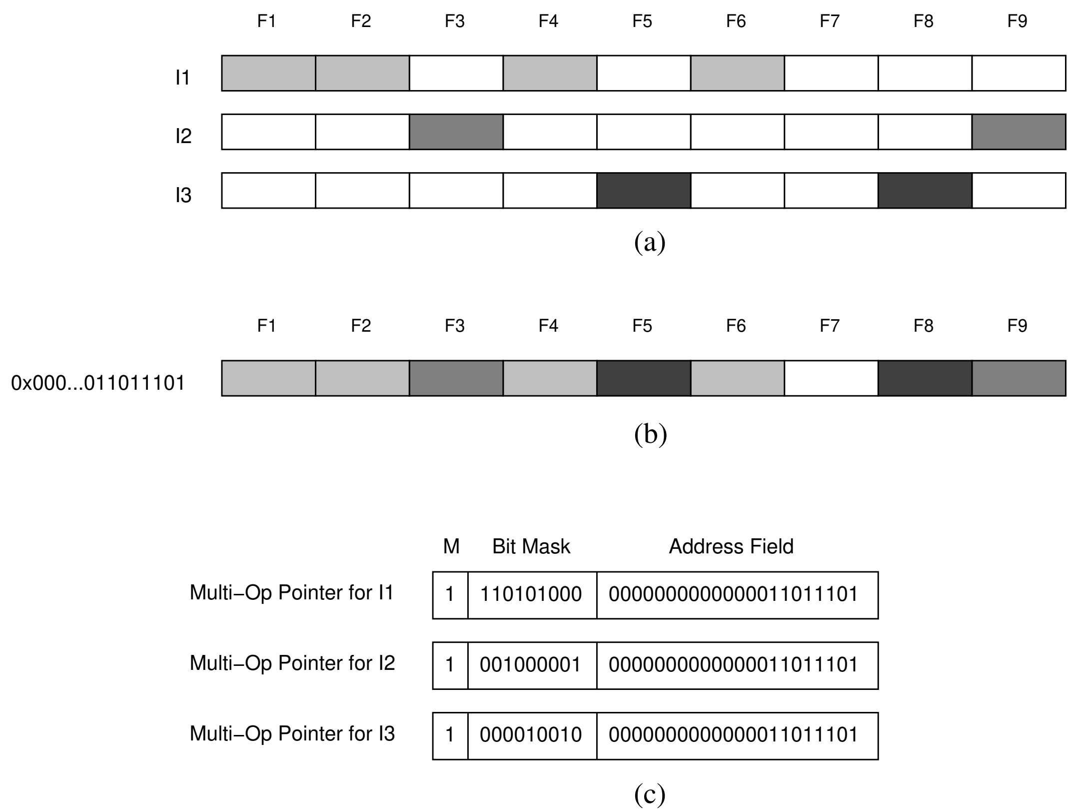
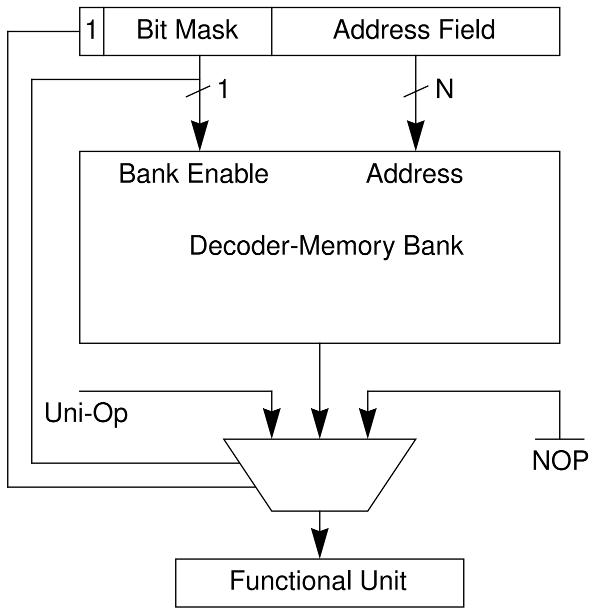


Figure 5.3: (a) Three multi-op instructions stored in separate long-instruction words. (b) The same instructions packed into a single long-instruction word. (c) The multi-op pointers used to extract the packed multi-op instructions.

bit mask should have as many bits as there are memory banks. Figure 5.3 (c) shows how the multi-op pointers used to access each of the three multi-op instructions in Figure 5.3 (b) are encoded. Each pointer consists of three components: the most significant bit which, when set, identifies the word as being a multi-op pointer; the bit mask; and the long-instruction word address. By setting the appropriate bits in the bit mask, the address specified in the address field can be supplied to only those decoder-memory banks that are used to store the corresponding machine operations. Decoder-memory banks that are not selected by the bit mask will return NOPs by default. This way, the operations of a multi-op instruction can be extracted from a packed long-instruction word and issued to the datapath in a long-instruction format.

The hardware needed to support the extraction of multi-op instructions from a packed long-instruction word is fairly simple. Figure 5.4 (a) shows how a multi-op pointer, stored in the instruction register, can be used to control the operation of a decoder-memory bank. For each decoder-memory bank, the appropriate bit from the bit mask can be hardwired to a bank-enable input. When the bit is set, the decoder-memory bank is enabled, and can be used to supply an operation to the corresponding functional unit. The address field is also connected to the address lines of each decoder-memory bank. By using the bit mask to specify



(a)

MSB	Bank Enable	MUX Output
0	0	Uni-Op
0	1	Uni-Op
1	0	NOP
1	1	Memory

(b)

Figure 5.4: (a) Using the multi-op pointer to access a decoder memory bank.
(b) Truth table for controlling the output of the multiplexer.

which decoder-memory banks to access, a multi-op instruction can easily be extracted from a packed long-instruction word, and supplied to the datapath in a long-instruction format.

Since the bank-enable input and the address lines of each decoder-memory bank are connected to the instruction register, care must be taken to ensure that only valid outputs are supplied to the corresponding functional unit. Figure 5.4 (a) shows that the output of each decoder-memory bank is connected to its associated functional unit through a multiplexer. Figure 5.4 (b) shows how the multiplexer is controlled by the most-significant bit (MSB) in the instruction register and the associated bit in the bit mask. When the MSB is cleared, the instruction register contains a uni-op instruction, and the instruction is supplied to the functional unit for further decoding. Recall that not all functional unit might be used to execute uni-op instructions. In this case, a NOP is supplied to the functional unit. When the MSB is set, the instruction register contains a multi-op pointer. If the corresponding bit in the bit mask is set, the operation stored at the specified address is fetched and supplied to the functional unit for further decoding. On the other hand, if the corresponding bit in the bit mask is cleared, a NOP is supplied to the functional unit.

5.2.3 An Algorithm for Storing and Packing Multi-Op Instructions in Decoder Memory

Section 5.2.2 described how the size of decoder memory can be reduced by storing multi-op instructions according to a priority assigned to each operation field, and by packing as many mutually-exclusive multi-op instructions as possible into the same long-instruction word. This section describes an algorithm that uses these techniques to reduce the size of decoder memory. Figure 5.5 shows the pseudo-code for the decoder-memory packing algorithm.

The main loop of the algorithm is executed as many times as there are *active fields*. By definition, an active field is an operation field in the long-instruction word that still has member operations that have not been packed. At the beginning of each loop iteration, the active fields are ranked according to the priority assigned to each field, and a *target field* is selected. At any given time, the target field is the one with the highest rank. Ranking the fields involves assigning them priorities that are inversely proportional to the number of their remaining, unpacked operations. Thus, the field with the smallest number of operations will be given the highest priority, while the field with the largest number of operations will be given the lowest priority. This, in turn, helps impose an order on the way instructions are stored in decoder memory, since instructions that use operations belonging to higher ranking fields will be stored first.

Once the active fields have been ranked, the remaining set of unpacked instructions are divided into two groups: *candidate instructions* and *reserve instructions*. Candidate instructions are those that use member operations of the target field. Since operations that belong to the same field are stored in the same memory bank, candidate instructions cannot be packed into the same long-instruction word. Instead, each candidate instruction is stored in a separate, long-instruction word. On the other hand, reserve instructions are those that do not use member operations of the target field, and that can therefore be used to fill the empty fields in the long-instruction words already used to store candidate instructions.

After assigning the remaining unpacked instructions to the set of candidate or reserve instructions, the cost of each instruction is calculated to help determine the order in which the instructions are to be stored and packed. The cost assigned to each instruction is given by the following formula:

$$Cost = \sum_{f \in \text{ active fields}} 2^{R_f} \times U_f$$

In this formula, R_f is the rank of active field f, and U_f is either one, if the instruction uses an operation associated with the active field f, or zero otherwise.

Since candidate instructions cannot be packed into the same long-instruction word, a new, long-instruction word is added for each candidate instruction. In effect, this adds a new word, or an additional address, to each memory bank associated with an active field. The candidate instruction with the highest cost is then selected and stored in the new long-instruction word.

```
while (there are active fields)
   rank active fields and select target field;
   divide unpacked instrs into candidate and reserve instrs;
   calculate the cost of each unpacked instr;
   for (all candidate instrs)
      add new long-instruction word to decoder memory;
      select candidate instr with highest cost;
      mark candidate instr as stored;
      if (active field re-ranking necessary)
         re-rank active fields;
         re-calculate costs of remaining unpacked instrs;
      end if
      do
         find highest-cost reserve instr that fits;
         mark reserve instruction as packed;
         if (active field re-ranking necessary)
            re-rank active fields;
            re-calculate costs of remaining unpacked instrs;
         end if
      while (reserve instrs are available)
   end for
   deactivate target field;
end while
```

Figure 5.5: Pseudo-code for the decoder-memory packing algorithm

When an instruction is stored in a long-instruction word, the number of operations in its corresponding active fields are decremented. This may, in turn, cause the ranking of the active fields to change. That is why, after a candidate instruction is stored, the number of remaining members in each active field is examined to determine if the active fields need to be re-ranked. If they do, the active fields are re-ranked, and new costs, based on the new rankings, are calculated for all remaining instructions that have not been stored or packed in decoder memory.

Next, to make use of any empty fields in the current long-instruction word, the set of reserve instructions is searched for the instruction with the highest cost that can fit into the current long-instruction word. If such an instruction is found, it is packed into the long instruction. If necessary, the active fields are also re-ranked and new costs are calculated. This search is repeated until no more reserve instructions can be found to fit into the remaining fields of the current long-instruction word. At this point, the next candidate instruction is stored in decoder memory.

Finally, after all candidate instructions have been stored, the target field, and any other field whose member operations have all been packed, is deactivated. A new target field is then sought and the process is repeated all over again.

The Table in Figure 5.6 shows how the algorithm can be applied to store and pack a group of eight instructions, each consisting of five fields, into a decoder memory. Even though some of the instructions, like I4 and I8, are actually uni-op instructions, they are only used here to demonstrate the operation of the algorithm. In its actual implementation, the algorithm is only used to store and pack multi-op instructions in decoder memory.

Figure 5.6 (a) shows how each instruction is initially stored in a separate long-instruction word, and at an arbitrary location in decoder memory. Since the instructions use all the fields, the fields are all considered to be active. At the beginning of the first iteration, the number of operations associated with each field is counted to determine the rank of each field. Since field F4 has the smallest number of associated operations, it is assigned the highest rank, and is designated the target field. Thus, all instructions that use field F4 — I1 and I4 — are considered candidate instructions, while the remaining instructions are considered reserve instructions. To determine which candidate instruction to store in decoder memory first, and which reserve instruction to pack first, the cost associated with each instruction is calculated. Since instruction I1 has a higher cost than instruction I4, it is stored in the first long-instruction word, at address 0x00.

Once I1 is marked as being stored, the operation counts associated with the fields it uses — F1, F3, and F4 — are decremented. Since doing so may change the ranking of the fields — in this case, it does — the number of remaining operations associated with each field is counted, and the fields are re-ranked if necessary. For example, even though field F4 still has the highest ranking, field F3 is assigned a higher rank than field F5 after the re-ranking since it now has a greater number of operations associated with it. The same is true for field F1, which is also assigned a higher ranking than field F2 because it has more operations associated with it.

To fill the unused fields in the long-instruction word used to store instruction I1, the set of reserve instructions is now searched for the highest-cost instruction that can be packed into the long-instruction word. In this case, two instructions, I5 and I8, can be packed. However, since instruction I5 has a higher cost than instruction I8, it is selected and marked as being packed. Once again, the operation counts associated with the fields used in instruction I5 — F2 and F5 — are decremented, and the fields are re-ranked if necessary. In this case, there is no need to re-rank the fields.

Address	Selected Instruction	Number of Operations					Re-rank?	Field Ranks					Instruction Costs							
		F1	F2	F3	F4	F5		F1	F2	F3	F4	F5	I1	I2	I3	I4	I5	I6	I7	I8
Iteration 1		4	4	3	2	3	Yes	0	1	2	4	3	21	6	11	16	10	5	9	2
0x00	I1	3	4	2	1	3	Yes	1	0	3	4	2	—	9	7	16	5	10	6	1
0x00	I5	3	3	2	1	2	No	1	0	3	4	2	—	9	7	16	—	10	6	1
0x01	I4	3	3	2	0	2	No	1	0	3	4	2	—	9	7	—	—	10	6	1
0x01	I6	2	3	1	0	2	Yes	1	0	3	4	2	—	9	7	—	—	—	6	1
0x01	I8	2	2	1	0	2	No	1	0	3	4	2	—	9	7	—	—	—	6	—
Iteration 2		2	2	1	—	2	Yes	0	1	3	—	2	—	10	7	—	—	—	5	—
0x10	I2	2	1	0	—	2	Yes	0	2	3	—	1	—	—	7	—	—	—	3	—
0x10	I7	1	1	0	—	1	No	0	2	3	—	1	—	—	7	—	—	—	—	—
Iteration 3		1	1	—	—	1	Yes	2	0	—	—	1	—	—	7	—	—	—	—	—
0x11	I3	0	0	—	—	0	—	—	—	—	—	—	—	—	—	—	—	—	—	—

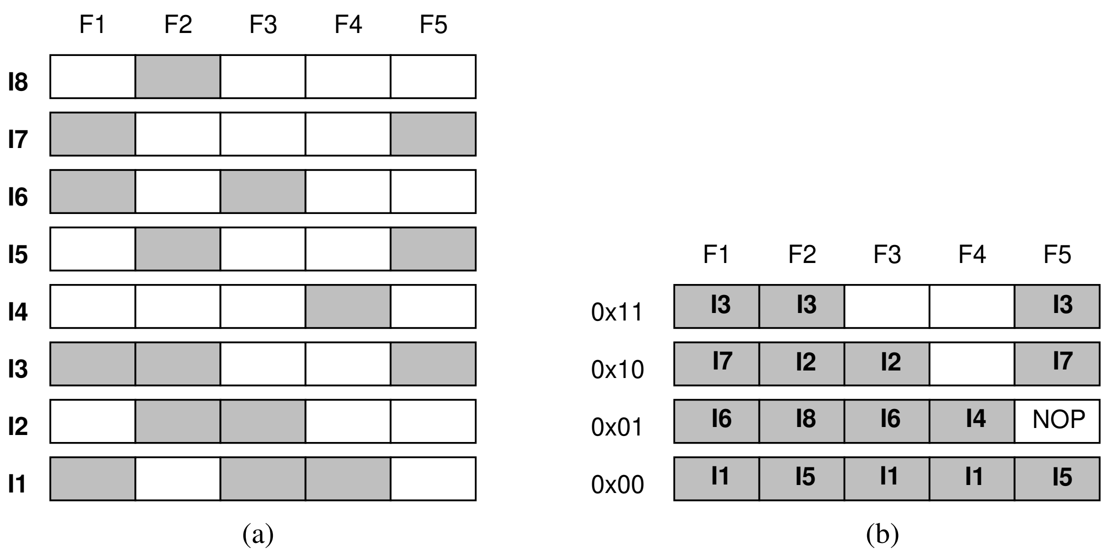


Figure 5.6: A working example of the packing algorithm. (a) Before the packing. (b) After the packing.

Since no other reserve instructions can be packed into the same long-instruction word used to store instructions I1 and I5, a new long-instruction word is used to store another candidate instruction at address 0x01. Since instruction I4 is the only remaining candidate instruction, it is stored in the new long-instruction word, and is marked as stored. The operation count associated with field F4, the only field used in instruction I4, is then decremented, and the fields are re-ranked if necessary. Once again, no re-ranking is necessary in this case.

To fill the unused fields in the long-instruction word used to store instruction I4, the set of reserve instructions is once again searched for the highest-cost instruction that can be packed into the long-instruction word. In this case, five instructions — I2, I3, I6, I7, and I8 — can be packed, but instruction I6 is selected because it has the highest cost. After decrementing the operation counts associated with the fields used in instruction I6 and re-ranking the active fields, the set of reserve instructions is again searched for an instruction that can be packed. This time, instruction I8 is selected because it has the highest cost. Since it is no longer possible to pack any other reserve instructions into the long-instruction word, and since there are no more candidate instructions to store, the first iteration is almost over. However, before starting the second iteration, the target field — F4 in this case — and any other field with no remaining associated operations are deactivated. This means that in future iterations, these fields will not be considered when ranking the active fields or when calculating the costs of different instructions.

Figure 5.6 (b) shows how the instructions are stored and packed in decoder memory after the application of the algorithm. For this example, the algorithm succeeds in reducing storage requirements from 40 words to only 17 words, a reduction of over 57%. The time complexity of this algorithm is $O(F \times I^3)$, where F is the number of operation fields and I is the number of multi-op instructions.

5.2.3.1 Comparing the Algorithm to Bin Packing

The problem of storing and packing multi-op instructions into long-instruction words is very similar to the classical bin packing problem [57]. In bin packing, a set of objects having different sizes are to be packed into the smallest possible number of fixed-capacity bins. Although this problem is NP-complete [57], several heuristic algorithms exist for finding approximate solutions in polynomial time.

However, even though there are similarities between the problem of storing and packing multi-op instructions into long-instruction words and bin packing, there are also major differences. Like the bin packing problem, the objective of storing and packing different multi-op instructions is to use the smallest possible number of long-instruction words. Here, the long instruction words are analogous to the fixed-capacity bins, while the multi-op instructions are analogous to the different-sized objects that are to be stored. However, unlike bin packing, the cost, or size, of each multi-op instruction is not fixed. Instead, it changes as the ranking of different fields change. Moreover, unlike bin packing, where objects can be stored in any bin they fit in, there are limitations imposed on packing certain multi-op instructions into the same long-instruction word. For example, candidate instructions cannot be packed into the same long-instruction word as other candidate instructions. This, in turn, results in the use of as many long-instruction words as there are candidate instructions at any given point in time. For these reasons, the heuristics used to solve the bin packing problem are not directly applicable to the problem of storing and packing multi-op instructions in long-instruction words.

5.2.4 Achieving Denser Instruction Packing with Field Clustering

Section 5.2.2 showed that the size of decoder memory can be reduced by storing multi-op instructions in a specific order, and packing multi-op instructions that use mutually-exclusive operation fields into the same long-instruction word. However, the probability of finding multi-op instructions that use mutually-exclusive operations fields is diminished by the tendency of instructions to use certain fields more frequently than others. For example, most multi-op instructions, in both the kernel and application benchmarks, use the fields associated with memory and addressing operations more frequently than other fields [2]. Since instructions that use the same field cannot be packed into the same long-instruction words, it becomes more difficult to find instructions that do not share any field and could therefore be packed into the same long-instruction word. Figure 5.7 illustrates this problem.

Figure 5.7 (a) shows two multi-op instructions stored in two, separate, long-instruction words consisting of eight fields each. Since both instructions use fields F7 and F8, they cannot be packed into the same instruction word. Storing both instructions therefore requires 16 words of memory. Here, it should be noted that the remaining fields in both instructions are mutually exclusive, and that, were it not for their use of fields F7 and F8, both instructions could have been packed into the same instruction word. This suggests that a denser packing can be achieved by dividing the fields of a long-instruction word into different groups, or clusters, and packing the fields of each cluster into separate instruction subwords.

Figure 5.7 (b) shows the same long-instruction words used in Figure 5.7 (a) divided into two subwords each. Here, the fields of the long-instruction word have been divided into two clusters, the first consisting of fields F1 – F4, and the second consisting of fields F5 – F8. This effectively divides each instruction into two sub-instructions that can each be stored in a different subword. In this example, instruction I1 is divided into sub-instructions I1-a and I1-b, while instruction I2 is divided into sub-instructions I2-a and I2-b.

Since both sub-instructions I1-b and I2-b use fields F7 and F8, they cannot be packed into the same subword. On the other hand, sub-instructions I1-a and I2-a use mutually exclusive fields, and can easily be packed into the same subword. Thus, by using clustering, and by packing sub-instructions into long-instruction subwords, instructions I1 and I2 can be stored in 12 words of memory instead of 16. This is shown in Figure 5.7 (c).

To enable the extraction of operations from different subwords, as many pointers as there are clusters must be encoded into a single instruction word. Like the single multi-op pointer used to extract operations from a long-instruction word, each of these pointers must also consist of a bit mask and an address field. Each bit mask is used to select the memory banks corresponding to the fields in the cluster. Similarly, each address field is used to access the appropriate locations from the memory banks selected by the corre-

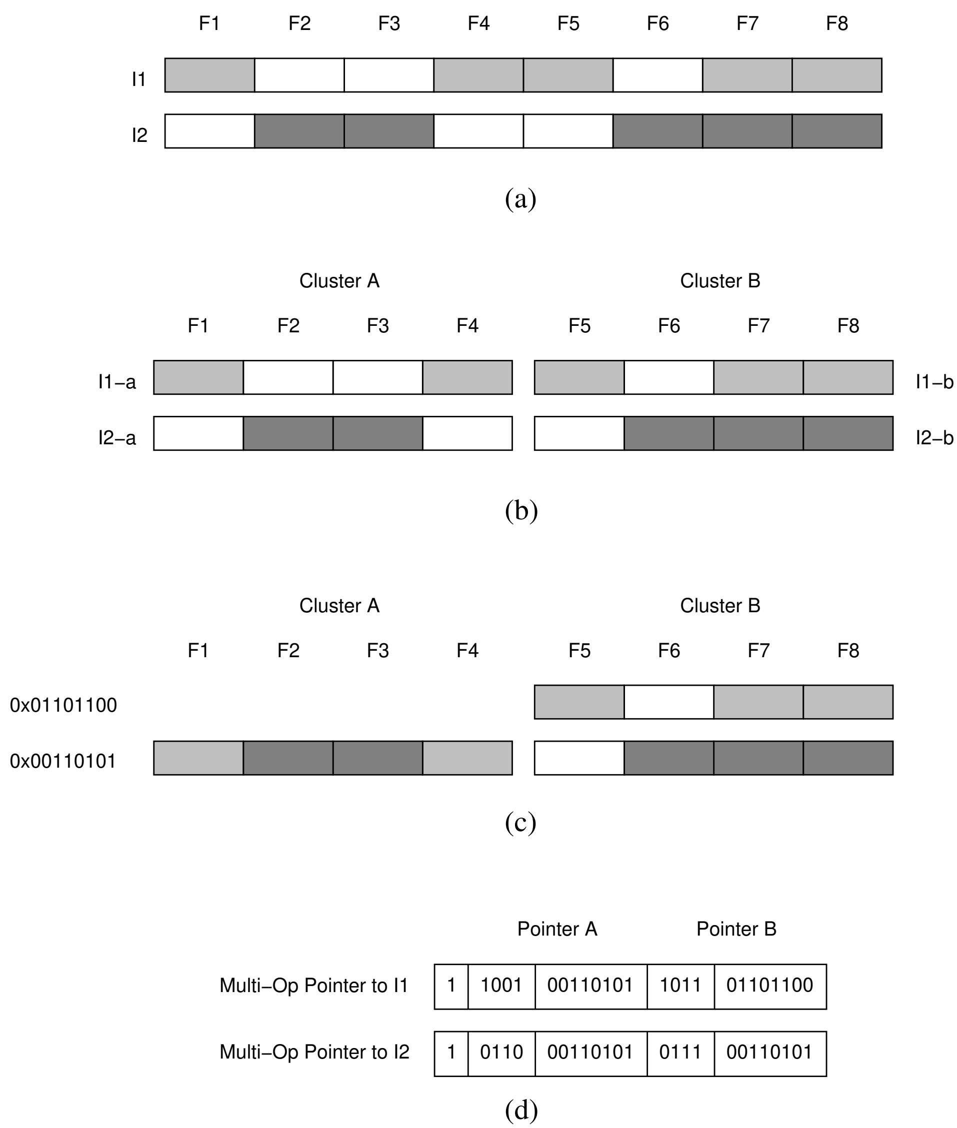


Figure 5.7: Field clustering. (a) Two multi-op instructions stored in separate long-instruction words. (b) Dividing the operation fields into two clusters. (c) Applying packing to different subwords. (d) Multi-op pointers used to extract instructions from two subwords.

sponding bit masks. Figure 5.7 (d) shows the multi-op pointers used to extract instructions I1 and I2 from the separate subwords. In this example, each multi-op pointer consists of two pointers, A and B, that are each used to extract the appropriate operations from the decoder memory banks associated with the corresponding cluster.

Although a clustering can be achieved by hardwiring the different decoder-memory banks to the appropriate fields of a multi-op pointer, a more flexible approach to clustering can be achieved by providing a dynamic means of mapping different decoder-memory banks to different multi-op pointer fields. Figure 5.8 shows a control circuit that uses a *bank control register* to specify the fields of a multi-op

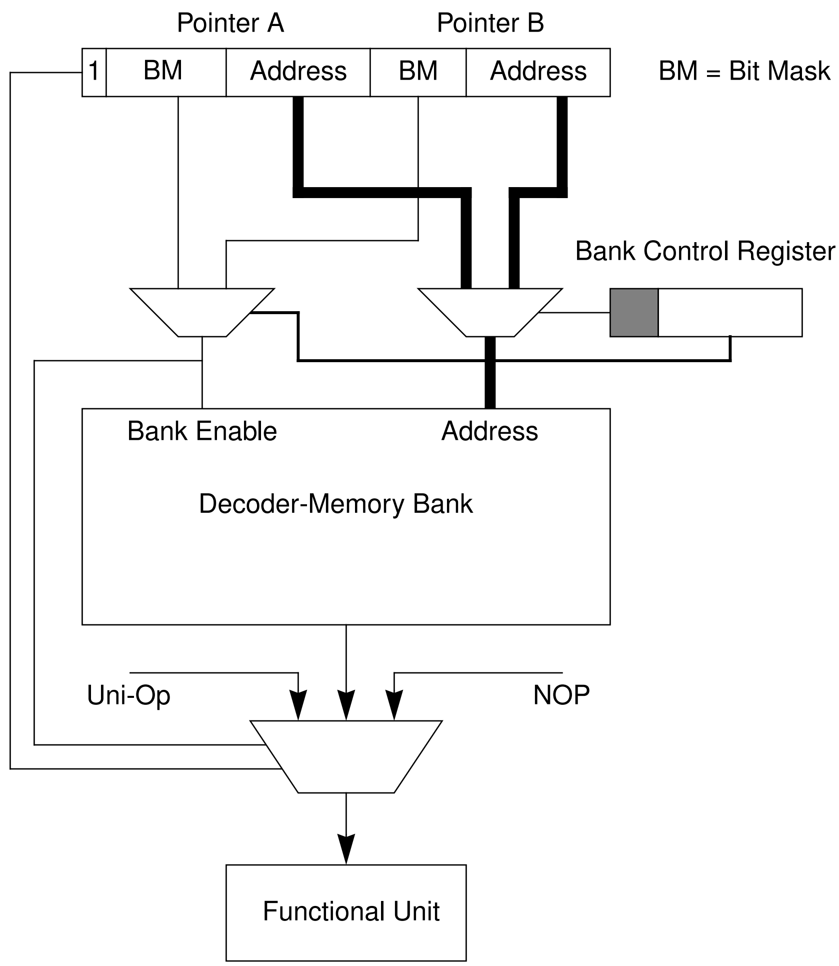


Figure 5.8: Decoder-memory bank control circuit.

pointer to which a decoder-memory bank is to be mapped to achieve a given clustering. The bank control register is used to map the bank-enable signal of a decoder-memory bank to any bit of any bit mask in the multi-op pointer. It is also used to specify the appropriate address field to use in the multi-op pointer. The bank control register can either be pre-set in a factory, or programmed in the field to implement the clustering that results in the most efficient packing of multi-op instructions for a given target application. Although Figure 5.8 shows the circuit used to support two clusters, the design can easily be extended to support more clusters. In the general case, each bank control register requires $\log_2(M)$ bits to select the appropriate bank-enable bit, and $\log_2(N)$ bits to select the appropriate address field. Here, M is the number of operation fields in a long-instruction word, and N is the number of clusters used.

Determining the number of clusters to use is an important design consideration. Ideally, there would be as many clusters as there are operation fields. In this case, a multi-op pointer would also consist of as many address fields as there are operation fields. Since each address field would only be used to access a single memory bank, there would be no need for bit masks. Such an encoding of multi-op pointers provides the most flexibility in extracting the operations of a multi-op instruction, since any multi-op instruction could be specified simply by setting the appropriate addresses in the corresponding address fields. This encoding also results in an ideal packing of operations, since each memory bank would only need to store the operations associated with its corresponding field. Each memory bank would also have to store a single NOP to handle the case when its associated fields is not used. Finally, this encoding enables the exploitation of redundancy at the operation level. For example, if two or more multi-op instructions use a specific operation with a specific mix of operands, only one copy of this operation would need to be stored in decoder memory. Any instruction that uses this operation could access it simply by setting the appropriate address in the corresponding address field. In turn, this helps to further reduce the size and cost of decoder memory.

In practice, however, the number of clusters that can be used is limited by the length of the instruction word. For most DSPs and embedded processors, the length of an instruction word varies between 16 and 48 bits [23],[24],[25],[26],[27],[28]. For the UTDSP architecture, which uses nine functional units, nine bits are needed to encode the bit mask, and another bit is needed to identify the instruction word as a multi-op pointer. Since the UTDSP uses 32-bit instruction words, this leaves 22 bits for encoding one or more address fields. Moreover, since 22 bits provides access to 4 Mwords, a very large decoder-memory address space that may not be fully utilized, clustering can be used to achieve a denser packing of multi-op instructions in decoder memory. Assuming two clusters are used, the remaining 22 bits can be divided into two, 11-bit address fields that can each provide access to 2 Kwords of decoder-memory address space. As the length of the instruction word gets shorter, however, fewer bits would be available for encoding the address fields, enabling fewer multi-op instructions to be accessed in decoder memory. Since performance is mainly achieved by exploiting the parallelism in multi-op instructions, limiting the access to multi-op instructions can have a significant impact on performance.

Choosing the number of clusters to use is therefore a trade-off between the reduced costs achieved by a denser packing of instructions, and the diminishing performance returns resulting from limiting access to multi-op instructions in decoder memory. As the length of the instruction word becomes shorter, the impact of this trade-off becomes more pronounced. However, even with longer instruction words, it may not always be necessary to use many clusters to achieve a dense packing of multi-op instructions. For example, the results in Section 5.4.1 show that for the 32-bit instruction word of the UTDSP, near-ideal packing can be achieved using only two clusters. The impact of the instruction word length on the encoding of multi-op pointers, and its resulting impact on performance and cost, are discussed in Chapter 6.

```
apply packing algorithm to multi-op instructions;
sort operation fields in decreasing order of NOPs stored in
   corresponding decoder-memory banks;
assign operation fields with greatest number of corresponding
   NOPs to first cluster;
assign remaining operation fields to second cluster;
```

Figure 5.9: Pseudo-code for dividing operation fields into two clusters.

The clustering algorithm used in this study is based on simple heuristics applied to the results of the packing algorithm outlined in Section 5.2.3. The initial motivation for using the heuristic clustering algorithm was to validate that dividing the long-instruction fields into two clusters will achieve a denser packing of multi-op instructions in decoder memory. Since the results, presented in Section 5.4.1, were near-ideal, more sophisticated algorithms, such as those based on graph theory [58], were not examined. Furthermore, since near-ideal packing was achieved without sacrificing performance, dividing the operation fields into more than two clusters was not attempted.

Figure 5.9 shows the heuristic clustering algorithm. The algorithm begins by using the packing algorithm to store and pack all multi-op instructions into decoder memory. Next, the operation fields are sorted in decreasing order of the number of NOPs stored in their corresponding decoder memory banks. Since the aim is to divide the fields into two clusters, half the fields with the greatest number of NOPs in their corresponding memory banks are merged into one cluster, while the remaining fields are merged into another cluster. For the UTDSP, which uses nine functional units, the four fields with the most NOPs in their corresponding banks are merged into one cluster, while the remaining five fields are merged into another cluster.

Since the clustering algorithm effectively breaks each instruction word into a pair of sub-instructions, a denser packing can be achieved by applying the packing algorithm to the sub-instructions formed by each cluster. The packing algorithm is therefore initially used for clustering purposes, and later used to pack the resulting sub-instructions into the corresponding subwords of decoder memory. That is why it is necessary to pass the clustering information to the packing algorithm. Clustering also enables the exploitation of redundancy at the sub-instruction level. This means that only unique instances of the sub-instructions associated with a particular cluster need to be stored in decoder memory. Identical instances of these sub-instructions can be represented by the same pointer. In the remainder of this thesis, a *single-cluster packing* will be used to denote the case when all operation fields are packed into the same instruction word, while a *double-cluster packing* will be used to denote the case when the sub-instructions formed by each cluster are packed into different subwords.

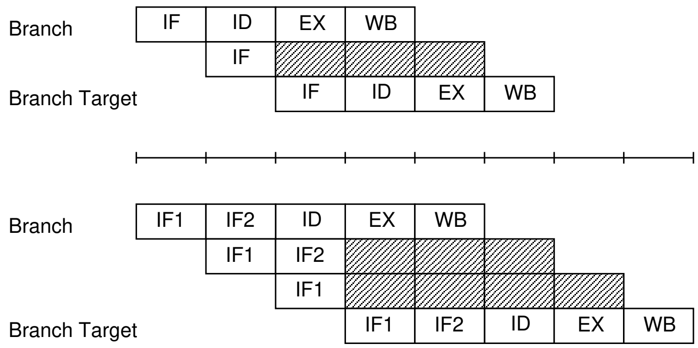


Figure 5.10: Effect of the additional pipeline stage on branch penalty

5.2.5 The Impact of Decoder Memory on Pipeline Performance

With the introduction of a decoder memory into the datapath, the impact on the pipeline should be examined more closely. Given that only one pipeline stage in the UTDSP architecture is used for fetching instructions, the introduction of a decoder memory makes it necessary for the clock cycle time to be long enough to accommodate two memory accesses: the first access is needed to fetch a uni-op instruction or a multi-op pointer from instruction memory, while the second access is needed to fetch a multi-op instruction from decoder memory.

To reduce the clock cycle time, the instruction-fetch stage can be broken into two separate stages. According to this scheme, the first instruction-fetch stage will be used to fetch a word from instruction memory, while the second instruction-fetch stage will be used to fetch a multi-op instruction from decoder memory. If the word fetched in the first instruction-fetch stage contains a uni-op instruction, the second instruction-fetch stage will not be used.

However, adding a new pipeline stage affects performance because it increases the latency of instructions that change control flow. Figure 5.10 shows the impact of adding an extra fetch stage on the execution performance of a branch instruction. In the original, four-stage pipeline of the UTDSP, the branch condition and the target address are evaluated in the instruction-decode (ID) stage, and this results in a single-cycle penalty for taken branches. On the other hand, the introduction of a second instruction-fetch stage delays the evaluation of the branch condition and the target address by an additional cycle, resulting in a two-cycle penalty for taken branches. The impact of adding an extra fetch stage on performance will be discussed in Section 5.4.2.

5.3 Long-Instruction Encoder

Section 5.2 described the organization of decoder memory, how multi-op instructions can efficiently be stored and packed in decoder memory, and the hardware needed to extract multi-op instructions from

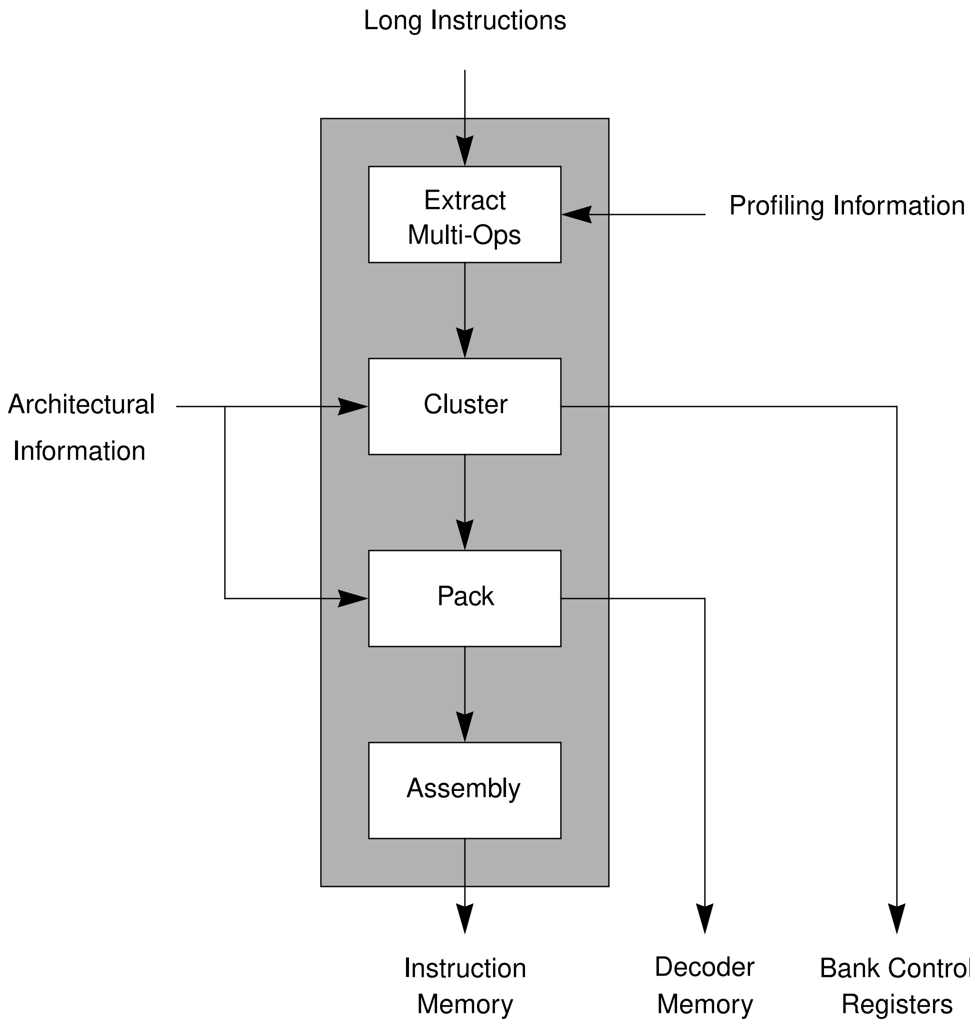


Figure 5.11: The instruction encoding pass

decoder memory. This section shows how the decoder-memory packing algorithm can be implemented as part of a long-instruction encoding pass in the UTDSP optimizing compiler.

Figure 5.11 shows the different phases of the long-instruction encoding pass. This pass operates on the long instructions generated by the operation-compaction pass of the compiler, and generates the encoded uni-op instructions or multi-op instruction pointers that are stored in instruction memory. It also specifies the contents of decoder memory by mapping the operations used in multi-op instructions to the appropriate decoder-memory banks. Within each decoder-memory bank, operations are encoded as uni-op instructions.

The encoding pass uses information about the target architecture, such as the number of functional units and the length of the instruction word used. The number of functional units helps the encoder determine the number of fields in a long-instruction word, and hence, the number of banks in decoder memory. Furthermore, the length of the instruction word helps it determine the number of bits available to encode the bit mask and address fields in a multi-op pointer. Finally, when the size of decoder memory is limited by cost constraints, the encoder pass may also use profiling information for more selective packing of multi-op instructions in decoder memory.

Initially, the encoder pass examines the code generated by the operation-compaction pass and extracts the multi-op instructions for storage and packing in decoder memory. Depending on the area budget of the processor, decoder memory may not be large enough to accommodate all multi-op instructions in a program. Alternatively, for the specified length of an instruction word, the number of bits available for encoding the address fields in a multi-op pointer may only provide a limited range for accessing multi-op instructions. In both cases, the encoder pass must be more selective of the multi-op instructions it extracts for storing and packing in decoder memory. Since limiting the number of multi-op instructions stored in decoder memory reduces the overall parallelism that may be exploited, being selective in which multi-op instructions to store in decoder memory will degrade execution performance. To limit the impact on execution performance, the selection of which multi-op instructions to store in decoder memory should be based on execution frequency: the higher the execution frequency of a multi-op instruction, the more important it is to store it in decoder memory, and vice versa. To help the encoder determine the frequency of execution of multi-op instructions, profiling information can be used. Multi-op instructions that are not selected for storage in decoder memory must be *serialized*. That is, they must be broken into their constituent operations, and each operation must be stored as a separate uni-op instruction in instruction memory. Serializing multi-op instructions helps to further reduce storage requirements because it eliminates the need for multi-op pointers and does not introduce any NOPs.

After extracting the multi-op instructions that are to be stored in decoder memory, and if the clustering phase is invoked, the different fields composing a long-instruction word are clustered into two sets. Since clustering effectively divides a long-instruction word into two subwords, this information must be communicated to the packing pass. Clustering information is also used to properly set the bank control registers associated with programmable decoder-memory banks. Finally, clustering information can be used to further reduce the size of decoder memory by only storing unique instances of sub-instructions. Even though several identical instances of a given sub-instruction might be used in a program, only one copy of the sub-instruction needs to be stored in decoder memory. The different instances of a sub-instruction can be represented by the same pointer. Recall that when clustering is used, each multi-op pointer contains two pointers. If clustering is not invoked, all fields are assumed to belong to a single cluster. In this case, the size of decoder memory can be reduced by storing unique instances of multi-op instructions.

After clustering is performed, the set of multi-op instructions is processed by the packing phase, which implements the decoder-memory packing algorithm described in Section 5.2.3. The packing phase uses information about the number of clusters used, the structure of decoder memory, and the length of an instruction word. If clustering is used, the packing algorithm is applied to the sub-instructions in each cluster. After packing the multi-op instructions, the packing phase generates information that enables the subsequent assembly phase to replace a multi-op instruction with its corresponding multi-op pointer. The packing phase also specifies the contents of decoder memory by mapping the operations used in multi-op instructions to the different decoder-memory banks. Each operation is encoded as a uni-op instruction. This mapping can then be programmed into decoder memory, or stored in a bootstrap memory and loaded into decoder memory just before the program is executed.

The last phase in the encoder pass generates the actual assembly code. The assembly phase examines the instructions generated by the post-optimizer. If an instruction contains a single operation, it is encoded as a uni-op instruction. If, on the other hand, it contains several operations, the assembly phase uses information from the packing phase to replace the instruction with the appropriate multi-op pointer. If the multi-op instruction has been serialized, it generates the corresponding sequence of uni-op instructions. The generated assembly instructions are stored in instruction memory where they can later be fetched and executed by the appropriate functional units.

Due to time limitations, the assembly phase was not implemented. Instead, the output of the packing phase was used in conjunction with the code generated by the operation-compaction pass to calculate the instruction-storage requirements, and hence its cost, for a given application. Since a uni-op instruction and a multi-op pointer both require a single instruction word, the size of the instruction memory needed to store a program, in words, is simply the total number of instructions. On the other hand, the size of decoder memory is just the sum of the words stored in each decoder memory bank.

5.4 Results and Analysis

This section examines the impact of the packing and clustering algorithms on the size of decoder memory. It also examines the impact of using the two-level instruction store on bandwidth requirements, storage requirements, and execution performance. Finally, this section examines the impact of selective multi-op instruction packing on overall performance and cost.

5.4.1 Impact of Packing and Clustering on the Size of Decoder Memory

The main objectives for using packing and clustering are to ensure that multi-op instructions are stored in an efficient manner in decoder memory. Figures 5.12 and 5.13 show the impact of packing and clustering on the size of decoder memory for both the kernel and application benchmarks, respectively. For each benchmark, the size of decoder memory is normalized to the ideal case, where only the operations used in multi-op instructions are stored in decoder memory without any NOPs. Since it may not always be possible

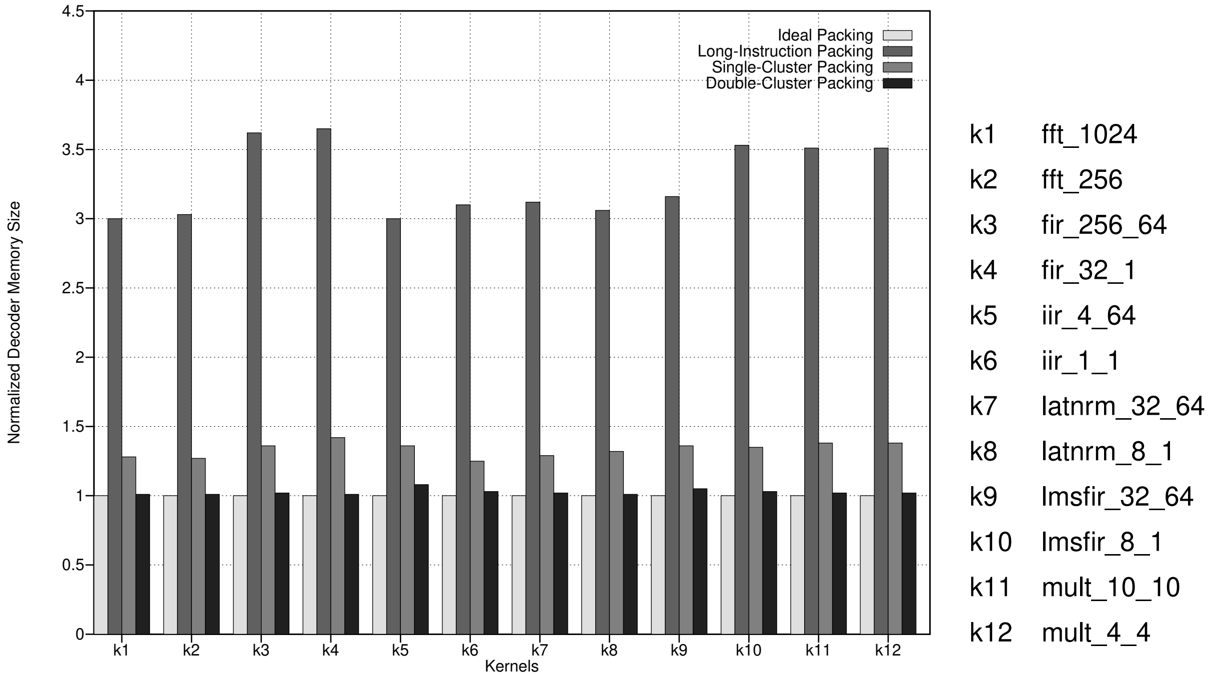


Figure 5.12: Impact of clustering on decoder-memory size for kernel benchmarks.

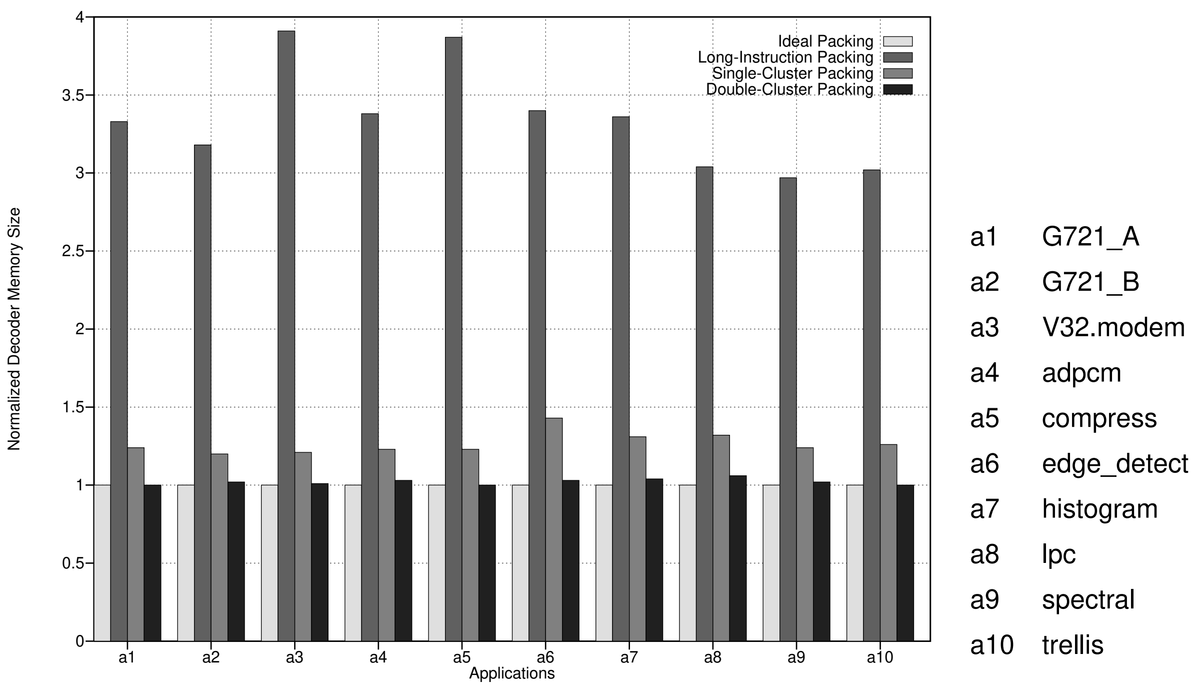


Figure 5.13: Impact of clustering on decoder-memory size for application benchmarks.

to achieve an ideal packing of multi-op instructions in decoder memory, this case represents a lower bound on the size of decoder memory. For each benchmark, the size of decoder memory is also compared to the case where multi-op instructions are stored in their original, long-instruction formats. Since doing so makes no attempt to reduce the size of decoder memory, this case is used as an upper bound.

For the kernel benchmarks, the results show that storing instructions in their long-instruction format requires 3.0 – 3.7 times more memory than an ideally-packed decoder memory (3.3 times more memory, on average). In this case, 67% – 73% of decoder memory space is wasted on NOPs (69% on average). Similarly, for the application benchmarks, storing instructions in their long-instruction format requires 3.0 – 3.9 times more memory than an ideally-packed decoder memory (3.3 times more memory, on average). Here, 66% – 74% of decoder memory space is wasted on NOPs (70% on average). From these results, it is clear that storing instructions in the long-instruction format is extremely wasteful of memory space, and that other approaches are needed to better utilize the available space.

One such approach involves exploiting the order in which multi-op instructions are stored in decoder memory, and packing as many multi-op instructions as possible into the smallest number of long-instructions words. Since this approach does not divide the fields of the long-instruction word into different clusters first, all operation fields effectively belong to the same cluster, and this approach is referred to as single-cluster packing. For the kernel benchmarks, single-cluster packing requires 1.3 – 1.4 times more memory than an ideally-packed decoder memory (1.3 times more memory, on average). In this case, only 20% – 30% of decoder memory space is wasted on NOPs (25% on average). On the other hand, for the application benchmarks, single-cluster packing requires 1.2 – 1.4 times more memory than an ideally-packed decoder memory (1.3 times more memory, on average). Here, only 17% – 30% of decoder memory space is wasted on NOPs (21% on average). Although single-cluster packing is more efficient in using decoder memory space than storing instructions in their long-instruction format, the amount of wasted space is still significant. The wasted space is mainly due to not being able to pack multi-op instructions into the empty fields of long-instruction words already packed with other multi-op instructions.

To overcome this problem, the fields of the instruction word can be divided into different clusters first, and packing can be applied to the operations in each cluster. This results in a denser packing because it reduces the granularity of the items being packed from the instruction level to the sub-instruction level. Since the fields are divided into two clusters before packing is applied, this approach is referred to as double-cluster packing. For the kernel benchmarks, double-cluster packing requires up to 1.1 times more memory than an ideally-packed decoder memory. In this case, only 1% – 7% of the decoder memory space is wasted on NOPs (3% on average). Similarly, for the application benchmarks, double-cluster packing requires up to 1.1 times more memory than an ideally-packed memory. In this case, only 1% – 5% of decoder memory space is wasted on NOPs (3% on average). For three of the application benchmarks — `G721_A`, `compress`, and `trellis` (a1, a5, and a10 in Figure 6.12) — the memory requirements achieved with double-cluster packing are identical to those in an ideally-packed memory. These results show that the combined use of clustering and packing enables a near-ideal utilization of decoder memory space.

5.4.2 Impact of Two-Level Instruction Storage on Bandwidth, Cost, and Performance

The main motivation for using the two-level instruction storage scheme was to reduce the high instruction storage and bandwidth requirements typically associated with long-instruction word architectures, while preserving the flexibility and scalability of the long-instruction format. The results presented in this sub-section show the impact of using the two-level storage scheme on the bandwidth requirements, storage requirements, and execution performance of the UTDSP.

5.4.2.1 Bandwidth Requirements

By storing only uni-op instructions and pointers to multi-op instructions in instruction memory, instruction bandwidth requirements are greatly reduced. For the UTDSP architecture, this enables the use of a 32-bit data bus, instead of a 288-bit data bus, to interface with external instruction memory. This has a direct impact on the cost of the chip since it requires fewer pins and less expensive packaging. It also results in less power consumption.

5.4.2.2 Overall Instruction Storage Requirements and Cost

In the two-level instruction storage scheme, instruction memory is only used to store uni-op instructions and pointers to multi-op instructions, while decoder memory is used to store multi-op instructions in a densely packed format. This greatly reduces storage requirements since very little space is wasted on storing NOPs. However, since every multi-op instruction in decoder memory requires a corresponding multi-op pointer, the savings are somewhat reduced.

Figures 5.14 and 5.15 show the impact of using the two-level instruction storage scheme on the overall instruction storage requirements of the kernel and application benchmarks. For each benchmark, the storage requirements for single-cluster packing, double-cluster packing, and ideal packing are shown normalized to the lower bound in which the operations used in multi-op instructions are stored as separate uni-op instructions in instruction memory. Also shown are the storage requirements for the upper bound in which all instructions are stored in their original, long-instruction formats in instruction memory. For both the lower and the upper bounds, decoder memory is not needed, and all instructions are stored in instruction memory.

To measure the storage overhead associated with the two-level instruction storage scheme, the storage requirements for the different packing schemes are compared with the lower bound. Recall that any over-

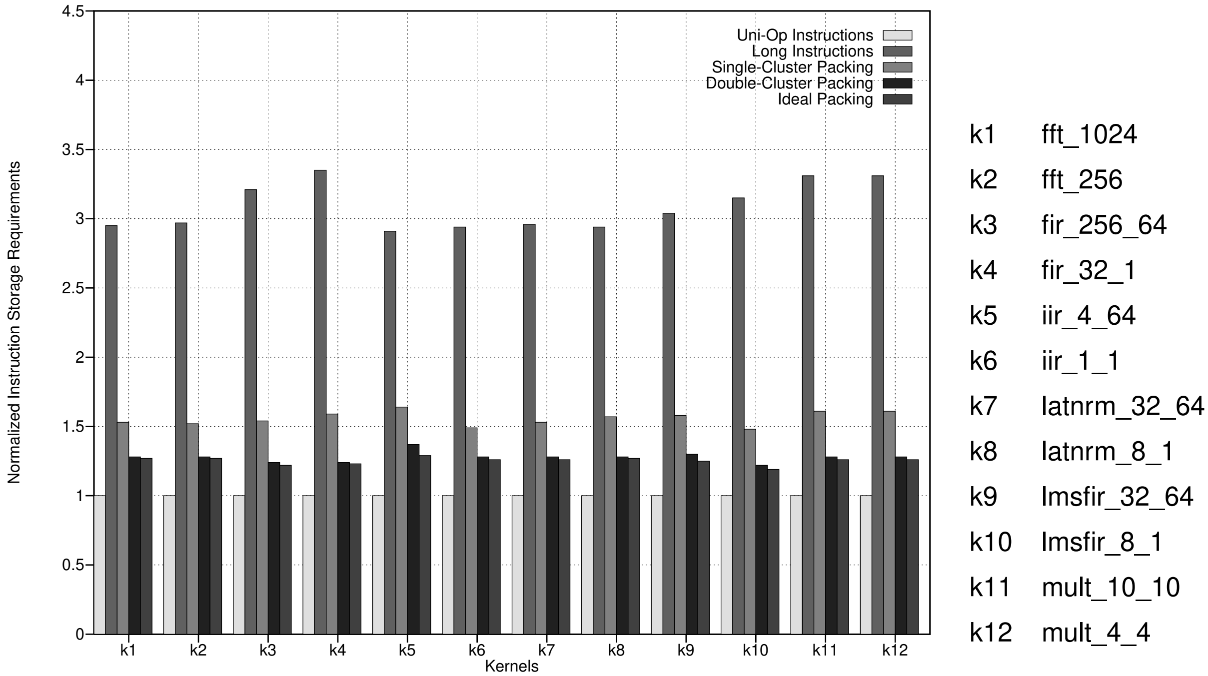


Figure 5.14: Effect of two-level instruction storage scheme on overall instruction storage requirements of kernel benchmarks.

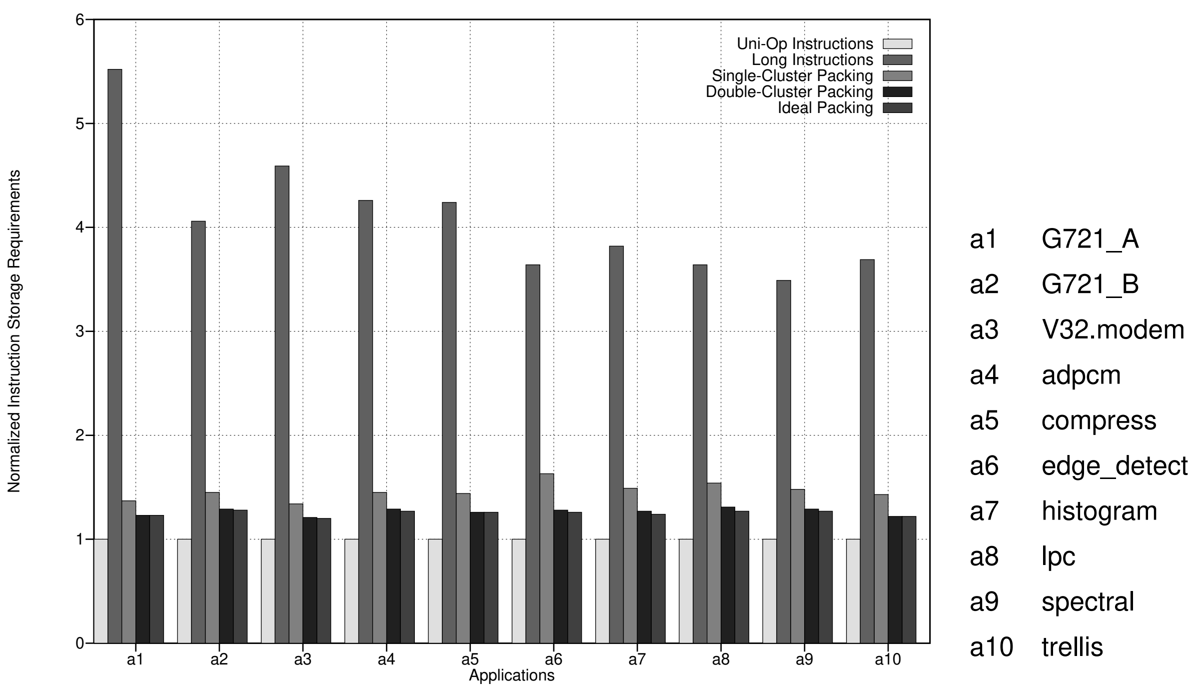


Figure 5.15: Effect of two-level instruction storage scheme on overall instruction storage requirements of application benchmarks.

head is due to storing multi-op pointers in instruction memory and NOPs in decoder memory. For the kernels, the storage requirements with single-cluster packing are 1.5 – 1.6 times greater than those for the lower bound (1.6 times greater, on average). With double-cluster packing, the storage requirements are 1.2 – 1.4 times greater than those for the lower bound (1.3 times greater, on average). Finally, with ideal packing, the storage requirements are 1.2 – 1.3 times greater than those for the lower bound (1.3 times greater, on average). For the applications, the storage requirements with single-cluster packing are 1.3 – 1.6 times greater than those for the lower bound (1.5 times greater, on average). With double-cluster packing, the storage requirements are 1.2 – 1.3 times greater (1.3 times greater, on average). Finally, with ideal packing, the storage requirements are 1.2 – 1.3 times greater than those for the lower bound (1.3 times greater, on average). These results show that the overhead introduced by the two-level instruction storage scheme is significant when compared to the lower bound. However, unlike the lower bound, where only sequential instructions can be executed, the two-level storage scheme enables the exploitation of parallelism by supporting the issue of multi-op instructions in their long-instruction format. This, in turn, results in a significant improvement in execution performance. In the next sub-section, the levels of execution performance attained with both schemes are compared.

Since the two-level instruction storage scheme supports the issue of multi-op instructions in their long-instruction format, it is also useful to compare its storage requirements, using the different packing schemes, with those of the upper bound. For the kernels, the storage requirements with single-cluster packing are 0.5 – 0.6 times those for the upper bound (0.5 times those of the upper bound, on average). With double-cluster packing, the storage requirements are 0.4 – 0.5 times those for the upper bound (0.4 times those of the upper bound, on average). Finally, with an ideal packing, the storage requirements are 0.4 times those for the upper bound, on average. For the applications, the storage requirements with single-cluster packing are 0.3 – 0.5 times those for the upper bound (0.4 times those of the upper bound, on average). With double-cluster packing, the storage requirements are 0.2 – 0.4 times those for the upper bound (0.3 times those of the upper bound, on average). Finally, with ideal packing, the storage requirements are 0.2 – 0.4 times those for the upper bound (0.3 times those of the upper bound, on average). These results show that, with the two-level instruction storage scheme, the flexibility of the long-instruction format can be preserved for a fraction of the storage requirements that would otherwise be required if instructions were stored in their original, long-instruction formats. These results also show that storage requirements using double-cluster packing are nearly identical to those using an ideal packing.

These results also provide better insight into the difference in cost between traditional and VLIW DSP architectures. Recall from Table 4.3 that the restricted VLIW architectures occupied 2.0 – 2.2 times the area of the traditional architectures, and that this difference was mainly due to their greater instruction storage requirements. Since using double-cluster packing to encode and store long instructions reduces the unencoded long instruction storage requirements of the applications by an average of 70%, the area occupied by instruction memory in the restricted VLIW architectures can be reduced by the same amount. Thus, for *UTDSP_mot*, the area occupied by instruction memory is reduced from 2.26 mm^2 (71% of the total area) to 0.68 mm^2 (42% of the total area). Similarly, for *UTDSP_ad*, the area occupied by instruction memory is reduced from 3.00 mm^2 (63% of the total area) to 0.90 mm^2 (33% of the total area). In turn, this reduces the total area of *UTDSP_mot* from 3.19 mm^2 to 1.61 mm^2, and the total area of *UTDSP_ad* from 4.79 mm^2 to 2.69 mm^2. As a result, the area occupied by the restricted VLIW architectures becomes only 1.13 times the area of traditional DSPs, demonstrating that VLIW architectures can be a cost-effective alternative to traditional DSP architectures.

5.4.2.3 Performance

The two-level instruction storage scheme introduces a decoder memory into the datapath. To shield the clock cycle time from the additional latency in accessing decoder memory, an additional instruction-fetch pipeline stage is used. However, the additional pipeline stage increases the latency of taken branches, and this degrades execution performance. Since the two-level instruction storage scheme supports the issue of multi-op instructions, it is once again useful to compare its execution performance to the upper bound, where all instructions are stored in their original, long-instruction formats. In this case, decoder memory is not needed, and a four-stage pipeline is adequate. As a reference point, it is also useful to compare the execution performance of the two-level instruction storage scheme to the lower bound, where programs are stored as uni-op instructions and executed sequentially. Since this scheme does not use decoder memory either, it also uses a four-stage pipeline.

Figures 5.16 and 5.17 show the execution speedup achieved using the two-level instruction storage scheme for both the kernel and application benchmarks. Since using any scheme to pack multi-op instructions in decoder memory will yield similar speedup results, only one data set, labeled *Two-level Storage*, is used to represent the results achieved using single-cluster packing, double-cluster packing, and ideal packing. Also shown are the results for the upper bound, labeled *Long-Instructions*, and the lower bound, labeled *Uni-Op Instructions*. For both the kernel and application benchmarks, results are normalized to the lower bound.

For the kernels, the results show that the upper bound is 2.4 – 3.7 times faster than the lower bound (2.8 times faster, on average), while the two-level instruction storage scheme is 2.2 – 3.7 times faster than the lower bound (2.6 times faster, on average). This also shows that the level of performance achieved with

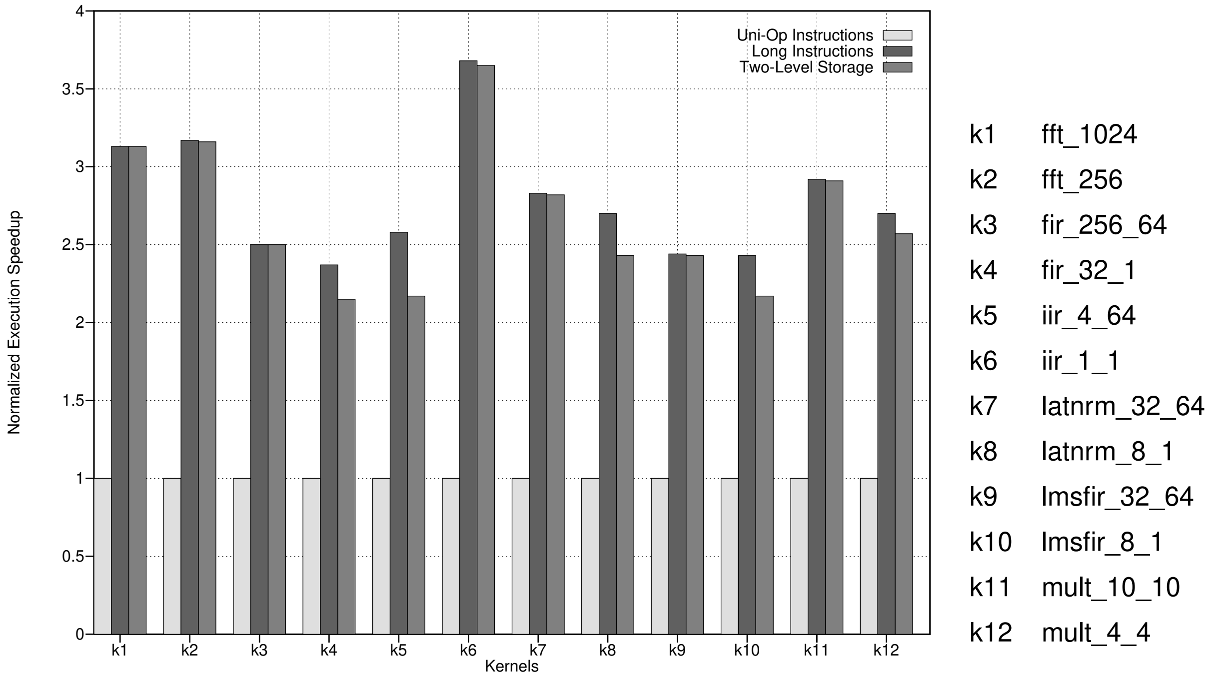


Figure 5.16: Effect of two-level instruction storage scheme on execution performance of kernel benchmarks.

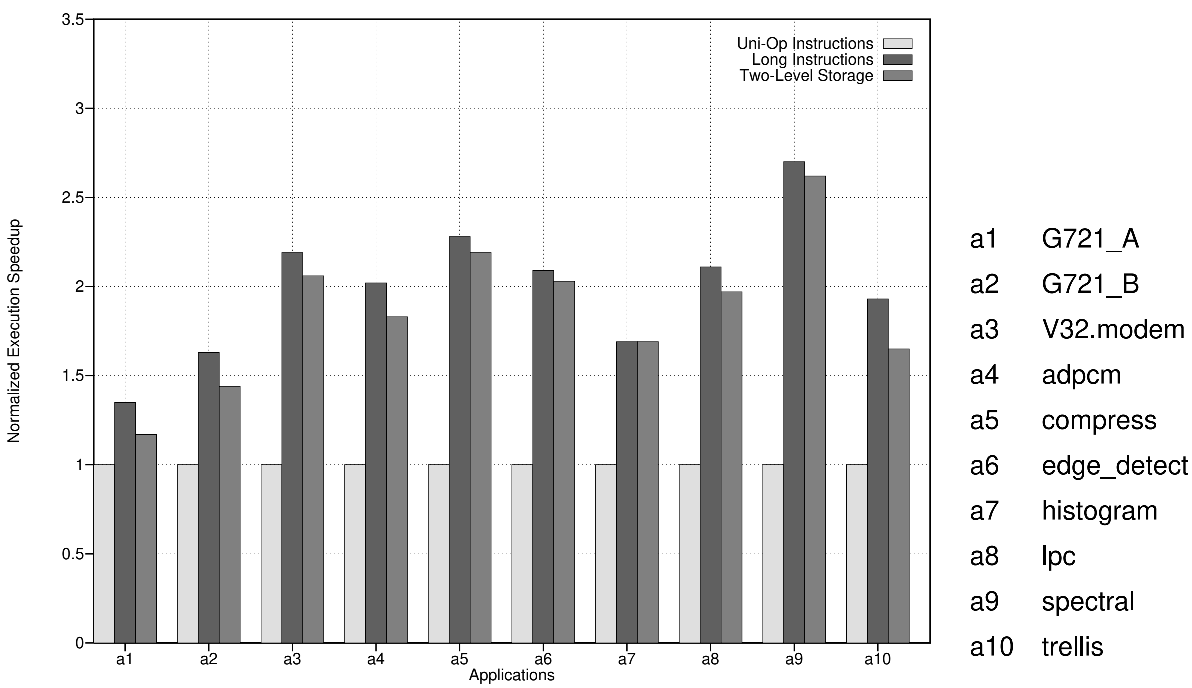


Figure 5.17: Effect of two-level instruction storage scheme on execution performance of application benchmarks.

the two-level instruction storage scheme is 0.8 – 1.0 times that achieved by the upper bound (1.0 times, on average).

For the applications, the upper bound is 1.4 – 2.7 times faster than the lower bound (2.0 times faster, on average), whereas the two-level instruction storage scheme is 1.2 – 2.6 times faster than the lower bound (1.8 times faster, on average). This again shows that the level of performance achieved with the two-level instruction storage scheme is 0.9 – 1.0 times that achieved by the upper bound (1.0 times, on average).

These results show that, even with the additional pipeline stage introduced by the use of decoder memory, the two-level instruction storage scheme achieves levels of performance that are close to the upper bound. This is mainly due to the relatively low incidence of control-flow operations and the ability of the compiler to eliminate branches by exploiting low-overhead looping operations. The latter operations are not affected by the additional pipeline stage. The slightly poorer execution performance of the applications relative to the kernels is mainly due to the higher incidence of taken branches, jumps, and subroutine calls in the applications. Finally, these results show that the introduction of an additional pipeline stage is feasible because it helps maintain the clock cycle time without significantly affecting execution time.

5.4.2.4 Performance/Cost Ratio

The performance and cost results indicate that using the two-level storage scheme provides a double advantage: *high performance*, achieved by supporting the exploitation of parallelism; and *low costs*, achieved by storing uni-op instructions and multi-op pointers in instruction memory, and densely packed multi-op instructions in decoder memory. This contrasts with the other extreme cases: storing all instructions in their original, long-instruction formats and exploiting the resulting parallelism, or only storing uni-op instructions and executing them sequentially. Although the former case achieves the highest performance, it also incurs the highest cost. Conversely, the latter case achieves the lowest performance but incurs the least cost.

Figures 5.18 and 5.19 show the impact of the two-level instruction storage scheme on the performance and cost of the kernel and application benchmarks, respectively. In both Figures, the horizontal axis shows execution performance normalized to the case when all instructions are stored in a long-instruction format, and the most parallelism is exploited. On the other hand, the vertical axis shows the area occupied by the datapath and instruction memory for the different instruction storage schemes. In both Figures, the origin of the vertical axis starts at 4.63 mm^2, the area occupied by the datapath assuming 0.25 μm technology. For the remainder of this section, the term *instruction memory* will refer to both instruction and decoder memory whenever the latter is used. Note that when the uni-op and long-instruction storage schemes are used, there is no need for decoder memory, and a four-stage pipeline is used.

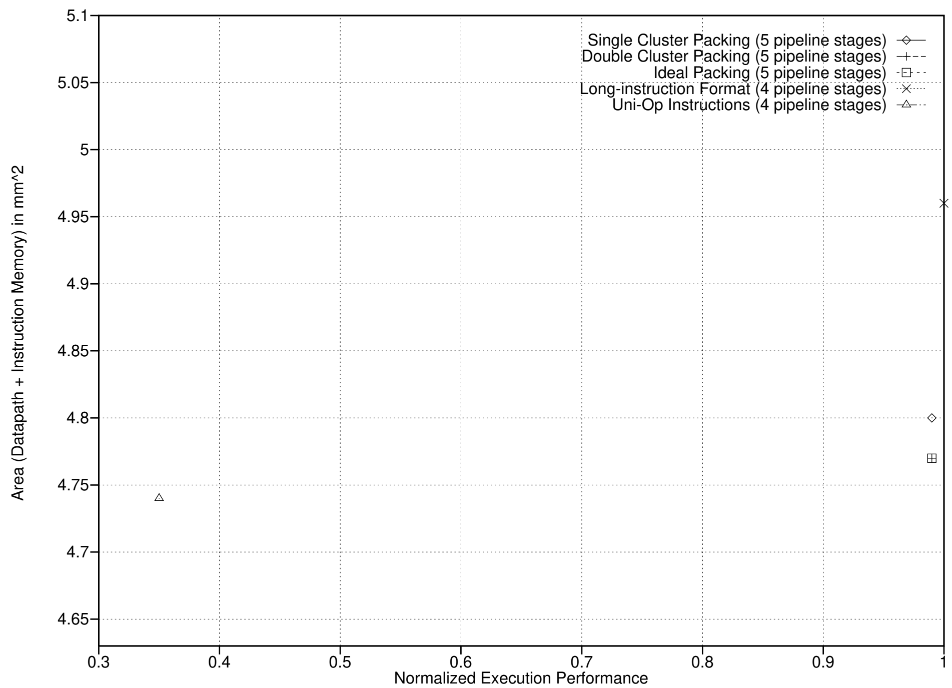


Figure 5.18: Impact of different instruction storage schemes on the performance and cost of kernel benchmarks.

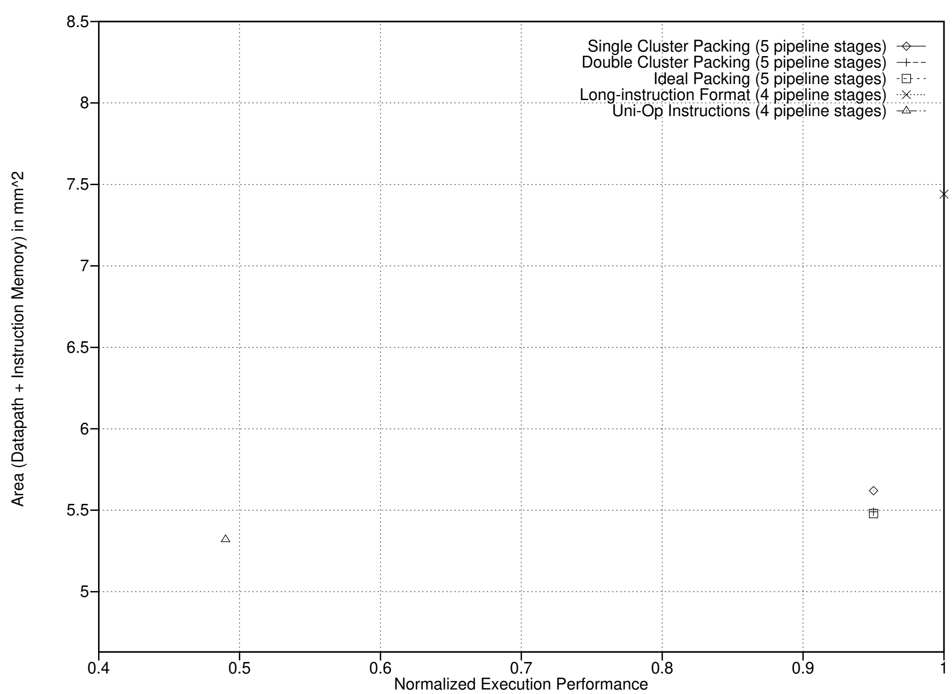


Figure 5.19: Impact of different instruction storage schemes on the performance and cost of application benchmarks.

Figure 5.18 shows that, for the kernels, the area occupied by the datapath dominates the area occupied by instruction memory. Furthermore, the difference in storage requirements between the long-instruction and uni-op storage schemes is small. However, given the small size of the kernel benchmarks, this is to be expected. Figure 5.18 also shows that the execution performance achieved by the single-cluster, double-cluster, and ideal packing schemes is nearly identical to that achieved with the long-instruction storage scheme. This shows that the former storage schemes are able to provide a similar level of flexibility, and support the same levels of parallelism, as the long-instruction storage scheme.

In Figure 5.19, the area of the datapath is still significant, but the greater difference between the storage requirements of the long-instruction and uni-op storage schemes provides better insight into the impact of the two-level storage scheme on performance and cost. Figure 5.19 shows that the single-cluster, double-cluster, and ideal packing schemes achieve 0.95 times the performance of the long-instruction format. This is, once again, due to their ability to support the same levels of parallelism as the long-instruction format. The slight degradation in performance is mainly due to the additional pipeline stage, which increases the penalty of taken branches. The two-level storage schemes also achieve 0.74 – 0.76 times the area of the long-instruction storage scheme. This significant reduction is due to the dense packing of multi-op instructions in decoder memory, which eliminates most NOPs. The two-level storage scheme also achieves 1.03 –1.06 times the area of the uni-op storage scheme, and this is mainly due to the overhead introduced by storing multi-op pointers.

These results show that the two-level storage scheme achieves similar levels of performance to those of the long-instruction format at significantly lower costs. In fact, the cost is only slightly higher than that of the uni-op storage scheme.

5.4.3 Impact of Selective Packing of Multi-Op Instructions on Performance and Cost

Section 5.3 described the way multi-op instructions can be selectively stored and packed in decoder memory to reduce the storage requirements, and hence the cost, of decoder memory. Since storing fewer multi-op instructions in decoder memory enables less parallelism to be exploited, doing so invariably affects performance. To study the effect of selective packing on performance and cost, the execution time for each of the kernel and application benchmarks was iteratively increased by serializing groups of multi-op instructions, and the corresponding impact on cost was measured.

Starting with the *parallel* execution time, which results from storing all multi-op instructions in decoder memory, execution time was increased, by serializing the least-frequently executed multi-op instructions, until the resulting increase in execution time amounted to 5% of the parallel execution time. At that point, the remaining multi-op instructions were re-clustered and re-packed in decoder memory, and the resulting impact on cost was measured. This process was repeated until the sequential execution time was reached,

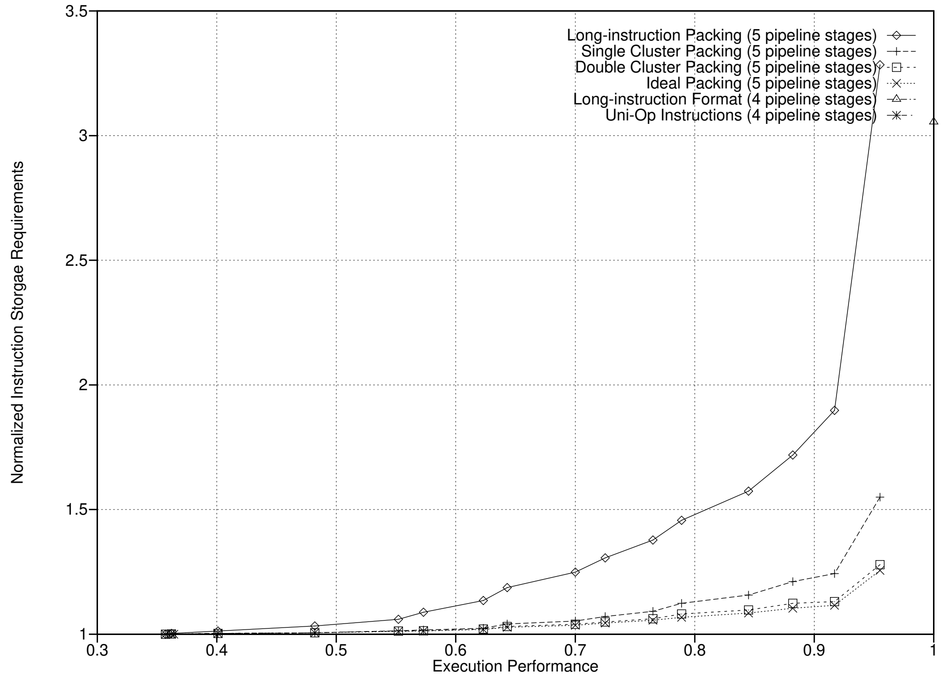


Figure 5.20: Impact of selective packing on performance and cost for kernel benchmarks.

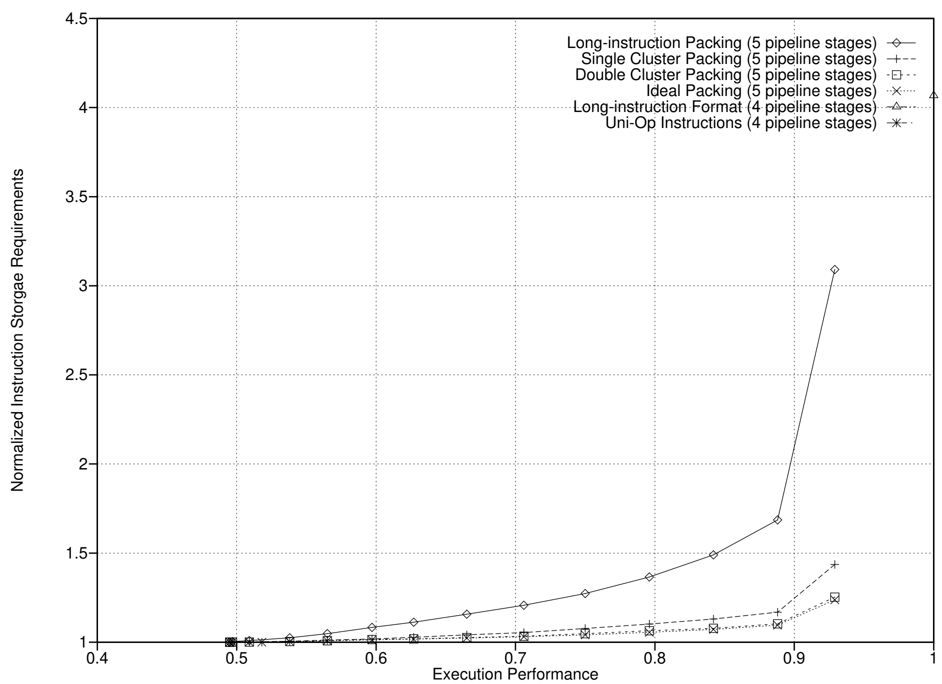


Figure 5.21: Impact of selective packing on performance and cost for application benchmarks.

in which no multi-op instructions remained in decoder memory and all operations were stored as uni-op instructions in instruction memory.

Figures 5.20 and 5.21 show the effect of selective packing on the performance and cost of the kernel and application benchmarks, respectively. Results are shown for single-cluster packing, double-cluster packing, and ideal packing. Results are also shown for long-instruction packing, where multi-op instructions in decoder memory are stored in the long-instruction format. Finally, results are shown for the upper bound, where all instructions are stored in instruction memory in the long-instruction format, and the lower bound, where all operations are stored in instruction memory as uni-op instructions. In both graphs, all performance results are normalized to the upper bound, while all cost results are normalized to the lower bound. Moreover, on both graphs, each point represents average performance and cost values over all corresponding benchmarks.

The results show that the relationship between performance and cost due to selective packing is, for the most part, linear. This means that reducing execution performance by a given amount, through the selective packing of multi-op instructions in decoder memory, results in a proportional reduction in cost. The notable exception occurs when performance is first reduced relative to the parallel execution time. In this case, reducing performance by approximately 5% results in a significant reduction in storage requirements. For the kernels, the reduction is 42% for long-instruction packing, 20% for single-cluster packing, 12% for double-cluster packing, and 11% for ideal packing. For the applications, the reduction is 46% for long-instruction packing, 19% for single-cluster packing, 12% for double-cluster packing, and 12% for ideal packing. This confirms the observation that most of the speedup achieved by exploiting parallelism is due to a fraction of the multi-op instructions. For both the kernels and the applications, almost 96% of the speedup, on average, is due to only 46% of the multi-op instructions. This also shows that, by being careful about which multi-op instructions to store in decoder memory, the size of decoder memory and the overall instruction storage requirements can be reduced without significantly affecting performance.

The results also show the effect of the additional pipeline stage on performance. This is evident for both the kernels and the applications in the performance gap between exploiting full parallelism using a four-stage pipeline versus a five-stage pipeline. However, since the gap is on the order of 5% for the kernels, and 7% for the applications, this shows, once again, that the additional pipeline stage has only a small impact on execution performance.

An interesting observation in the results for the kernel benchmarks is that the storage requirements for the long-instruction packing scheme are greater than those for storing instructions in their long-instruction formats. The reason for this is that, with long-instruction packing, a multi-op pointer must be stored in instruction memory for each instance of a multi-op instructions stored in decoder memory. Since the majority of instructions in the kernel benchmarks are multi-op instructions, the additional storage needed for the multi-op pointers is significant. On the other hand, storing instructions in their long-instruction formats does not suffer from this problem because all instructions are stored in instruction memory, and there is no need for multi-op pointers.

Finally, for both the kernel and application benchmarks, the results show that the single-cluster and double-cluster packing schemes produce near-ideal results, and significantly reduce instruction storage requirements.

5.5 Summary

In this chapter, a new method for storing and encoding the long instructions of the UTDSP model architecture was presented. The main objective was to devise an inexpensive means of reducing instruction storage and bandwidth requirements without compromising the flexibility of the long-instruction format. This was achieved by using a decoder memory to implement a two-level storage scheme similar to the two-level control stores used in early microprogrammed computers. The first-level instruction memory is used to store uni-op instructions and pointers to multi-op instructions. The second-level decoder memory is used to store multi-op instructions in a packed format. The decoder memory consists of as many banks as there are functional units, and each bank is used to store the operations that are to be executed on the corresponding functional units. When a multi-op pointer is fetched from instruction memory, it is used to enable the banks that hold the operations in the corresponding instruction. The pointer also specifies an address that is applied to all the enabled banks. While banks that are enabled return the appropriate operations, banks that are not return NOPs. This way, operations fetched from decoder memory can be issued to the datapath in a long-instruction format. Since fetching instructions in a two-level instruction store requires accessing both instruction memory and decoder memory, an additional instruction-fetch pipeline stage was introduced to reduce the clock cycle time.

To reduce instruction storage requirements and achieve a dense packing of multi-op instructions in decoder memory, several techniques were used. These included storing multi-op instructions in decoder memory according to the priorities assigned to different operation fields; using the same long-instruction word to store multi-op instructions with mutually exclusive operation fields; using field clustering to increase instruction packing density; and storing unique instances of multi-op instructions or sub-instructions. Using these techniques, the two-level storage scheme was able to preserve the flexibility and performance of the long-instruction format at a fraction of the cost.

To assess its impact on instruction bandwidth, storage requirements, and execution performance, the two-level storage scheme was compared to a lower bound where all instructions are executed sequentially, and the corresponding operations are stored as uni-op instructions. The two-level storage scheme was also compared to an upper bound where all instructions are stored in a long-instruction format, and parallelism is fully exploited. In terms of the bandwidth needed to fetch instruction words from the first-level instruction memory, the two-level storage scheme requires the same amount of bandwidth as the lower bound. For the UTDSP architecture, this is 32 bits. This represents a great saving when compared to the 288 bits required for the upper bound.

Compared to the lower bound, the two-level storage scheme requires 1.2 – 1.4 times the storage requirements for the kernels, and 1.2 – 1.3 times the storage requirements for the applications. The greater storage requirements of the two-level storage scheme are mainly due to the overhead introduced by the multi-op pointers in instruction memory and the NOPs in decoder memory. However, since the two-level storage scheme enables the exploitation of parallelism, it achieves 2.2 – 3.7 times the performance of the lower bound on the kernels (2.6 times the performance, on average), and 1.2 – 2.6 times the performance of the lower bound on the applications (1.8 times the performance, on average).

Compared to the upper bound, the two-level storage scheme requires 0.4 – 0.5 times the storage requirements for the kernels, and 0.2 – 0.4 times the storage requirements for the applications. This significant reduction in storage requirements is due to the efficiency of the instruction storage and packing techniques. At the same time, the reduction in storage requirements has little impact on performance, and the two-level storage scheme achieves 0.8 – 1.0 times the performance of the upper bound on the kernels (0.98 times the performance, on average), and 0.9 – 1.0 times the performance of the upper bound on the applications (0.95 times the performance, on average). The high levels of performance achieved by the two-level storage scheme is mainly due to its ability to preserve the flexibility of the long-instruction format.

Finally, the impact of selectively packing multi-op instructions in decoder memory was assessed for both the kernel and application benchmarks. In general, selective packing could be used when the size of decoder memory is restricted by processor area budgets, or when the number of bits available for encoding the address fields in a multi-op instruction pointer is limited. By storing the most frequently executed multi-op instructions in decoder memory, the storage requirements of decoder memory can be reduced without significantly affecting execution performance. In fact, for both the kernels and the applications, the results show that 96% of the performance achieved when all multi-op instructions are stored in decoder memory can be achieved by only storing 46% of the multi-op instructions. Multi-op instructions that cannot be stored in decoder memory are typically serialized, and their constituent operations are stored as uni-op instructions in instruction memory.

The next chapter describes how the datapath and the instruction set of the UTDSP can be customized to meet the functional, performance, and cost requirements of a target application, or group of applications.

The ability to customize the datapath and the instruction set is due, in large part, to the programmable decoder memory, which enables a multi-op pointer — in essence, a modifiable *meta-instruction* — to specify any multi-op instruction and supply it to the datapath in a long-instruction format. The next chapter also examines the impact of the instruction word size on the encoding of multi-op pointers, and how this affects performance and cost.

Chapter 6

Application-Specific Datapaths and Modifiable Instruction Sets

The UTDSP is an application-specific programmable processor (ASPP) whose flexible, VLIW architecture and cost-effective instruction-encoding scheme make it ideally suited for embedded DSP applications. Depending on the functional, performance, and cost requirements of their target applications, ASPPs can easily be configured with a minimal set of functional units and other datapath elements to meet these requirements. Since ASPPs are programmable, their instruction sets can also be customized to exploit the underlying architectural features and support the exploitation of parallelism.

This chapter examines how the UTDSP can be customized to meet the requirements of its target applications. Section 6.1 describes how datapath components and machine operations for the UTDSP architecture can be specified. It also describes the tools that can be used to customize a specified architecture to a target application or set of applications. Finally, it describes the tools used to assess the performance and cost of a specified architecture.

Section 6.2 describes the way the UTDSP instruction set can be modified to better match the requirements of a target application. Unlike the tightly-encoded instruction sets of traditional DSPs, where only limited instances of parallelism can be exploited, the UTDSP instruction set supports the exploitation of parallelism anywhere in a program. It also achieves high code density by packing the multi-op instructions generated by the compiler in decoder memory, and using multi-op pointers in instruction memory to represent them. In this sense, multi-op pointers are special *meta-instructions* whose semantics change with the contents of decoder memory. This not only provides a great deal of flexibility, but also gives rise to the notion of a *modifiable* instruction set.

Finally, Section 6.3 examines the impact of reducing the instruction word length and using different multi-op pointer encoding styles on cost and performance. In general, reducing the length of the instruction word reduces system costs by reducing the space, and hence the area, needed to store a program. It also simplifies packaging and reduces the number of pins required. However, reducing the instruction word length also degrades performance for at least three reasons. First, fewer bits are available for encoding the address fields in multi-op pointers. As a result, fewer multi-op instructions can be accessed in decoder memory, and less parallelism can be exploited. This problem is exacerbated when multiple pointers are stored in an instruction word, since the length of the address fields is effectively halved. Second, uni-op instructions that use immediate operands will have to be stored as two-word instructions. This increases overall execution time because of the added latency in fetching, decoding, and executing such instructions. Finally, reducing the instruction word length reduces the number of source and destination registers that can be specified in an operation. As the results in Chapter 4 demonstrated, using a smaller set of registers degrades execution performance due to the additional memory-access operations needed to move data between the stack and the smaller set of registers.

6.1 Application-Specific Datapaths

This section describes how the UTDSP can be customized to the functional requirements of a target application or set of applications. It also describes the experimental tools and methodology used to study the effects of different datapath configurations on performance and cost, and how the tools can be used to specify a datapath that meets performance and cost requirements. Finally, it demonstrates how different architectural configurations, with different levels of cost and performance, can be specified to execute the kernel and application benchmarks.

6.1.1 Specifying Datapath Components

To synthesize a datapath, the designer must first specify its components. For the UTDSP, these components include the length of instruction and data words, the number and type of functional units, the size of instruction- and data-memory banks, and the number of registers in each register file. The functional units are specified using a *machine-description file* that essentially describes a template architecture to the compiler. By reading the machine-description file, the compiler can generate long instructions that match the template architecture. Since the UTDSP is a VLIW architecture, each functional unit has a corresponding field in the long instruction. On the other hand, the number of registers in each register file is specified by an appropriate flag in the compiler front-end. The reason the number of registers are specified using compiler flags is that the compiler needs to know the number of available registers to perform register allocation. In this section, 32 registers will be assumed in each of the integer, address, and floating-point register files. Other components of the datapath, such as the amount of instruction and data memory to use, can be derived from the application. By only using the minimum amount of memory for storing instructions and data, costs can be reduced. Finally, the set of operations that can be executed by each of the functional units is defined in an *operation-definition file*. This file defines all the machine operations that can be generated by the compiler, and maps each operation to a class of functional units on which it can be executed.

However, this approach to specifying datapath components and mapping operations to functional units is not without its limitations. For example, since there can only be a single execution thread, only one control unit can be used. Furthermore, a maximum of two memory units can be specified, since the data partitioning pass in the post-optimizing back-end of the compiler is only configured to deal with two data memory banks. Although the data-partitioning algorithm itself may be extended to handle more than two data-memory units, this was not examined in this study. Naturally, by limiting the number of memory units that can be used, the amount of load/store parallelism that can be exploited is also limited. Finally, it is currently not possible to execute operations of a given type on functional units that execute operations from another type. For example, it is not possible to execute addressing operations on integer units. This is due to the current mechanism for defining machine operations, which makes it impossible to execute a machine operation on different classes of functional units. As a result, it is not possible to execute operations of different types on the same functional unit, and this makes it impossible to assess the impact of doing so on overall performance and cost. Finally, to ensure correct execution, there must be at least one functional unit for every type of operation used.

Given these limitations, it is currently difficult for the designer to fully explore the architectural space in search of the architecture that yields the best performance and cost. Nonetheless, this section will demonstrate how the tools can be used to explore the design space. With future refinements to the current set of tools, a more detailed investigation of the space will become possible.

6.1.2 Experimental Tools and Methodology

Figure 6.1 shows the tools and the iterative methodology used to specify a datapath and an instruction set that meet the functional, performance, and cost requirements of a target application, or set of applications.

Applications are written in the C programming language, and are mainly used to specify system functionality. Although other approaches exist for specifying system functionality, such as the use of signal-flow graphs [59], programming in C has become popular among DSP and embedded systems programmers because of the ability of the C language to support high-level abstractions while providing low-level control. Furthermore, programs written in C are portable, and are easier to debug, upgrade, and maintain than assembly code.

Using the datapath components defined by the machine-description file and the set of machine operations defined by the operation-definition file, the compiler generates long instructions for execution on the target architecture. Prior to generating the long instructions, the compiler applies a number of optimiza-

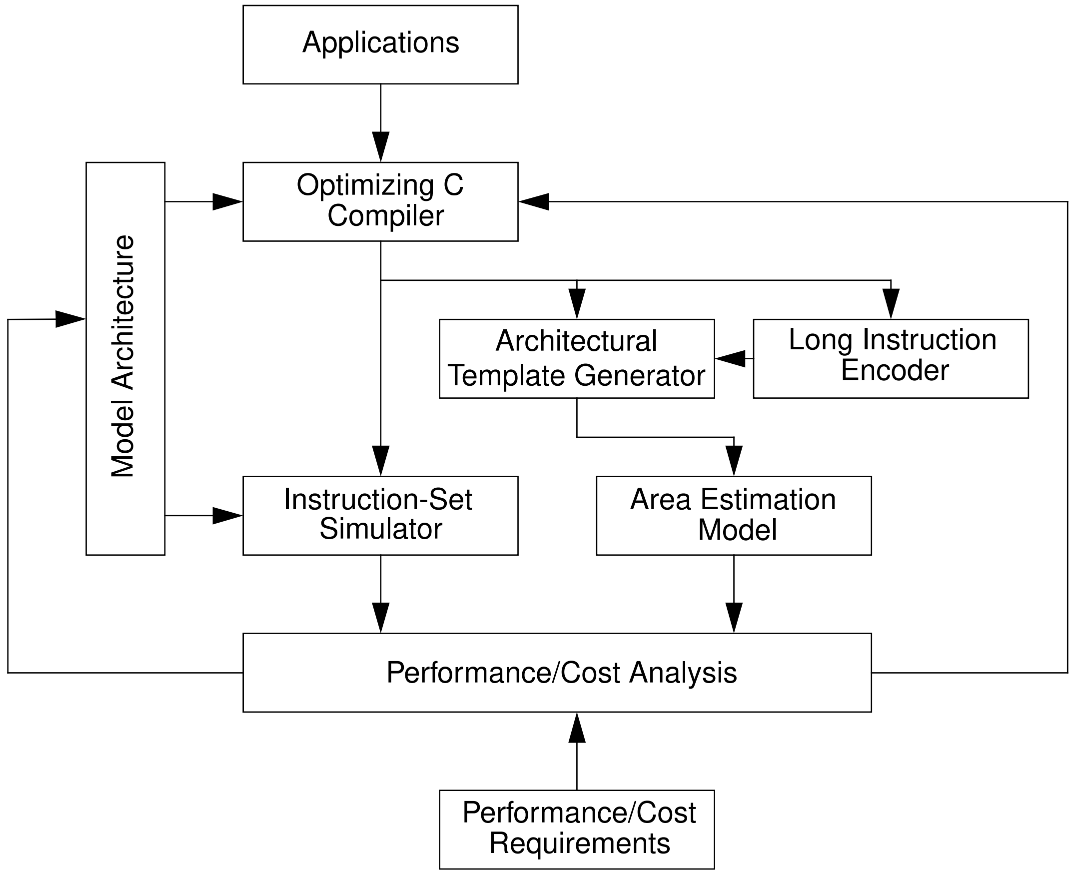


Figure 6.1: Tools used to specify application-specific datapath and instruction set

tions to ensure that the generated long instructions will execute efficiently, and that they will make the best use of the target architecture.

The instruction-set simulator is used to simulate the execution of the long instructions generated by the compiler on the target architecture. In addition to modeling the execution of an application, the simulator is used to gather run-time data about the dynamic behaviour of the application. However, the simulator is mainly used to measure the execution time of the application, and hence its performance, when executed on the target architecture.

The long instructions generated by the compiler are also processed by the *long-instruction encoder*. Recall, from Chapter 5, that the encoder is mainly responsible for packing long instructions into decoder memory and generating the appropriate multi-op pointers that can be used to access these instructions. However, in its current implementation, the encoder is only used to calculate the instruction storage requirements for the application. These include the storage requirements for both instruction memory, which is used to store uni-op instructions and multi-op pointers, and decoder memory, which is used to store multi-op instructions in a packed format

The long instructions are also processed by the *architectural template generator*. Recall, from Chapter 2, that an architectural template is used to describe the target architecture to the datapath *area-estimation model*. Although the main datapath components are specified by the machine-description file and special compiler flags, these are mainly used to guide the compiler in code generation. The template generator refines the architectural description by analyzing the actual code generated by the compiler, and specifying the minimal architectural resources needed to meet the functional requirements of the application. For example, if the machine-description file specifies a floating-point unit, and the generated code never uses a floating-point multiplication, the template generator will not include a floating-point multiplier in the architectural template. Similarly, if 32 integer registers are specified when only 24 are actually used, the template generator will only specify 24 integer registers. The template generator also uses the code generated by the compiler to determine the storage requirements for each data-memory bank. Since the generated code uses assembly directives to identify global variables and arrays, the template generator uses these directives to calculate the data storage requirements. It also adds a fixed stack size to the storage requirements of each data-memory bank to handle stack references. Finally, the template generator uses the instruction storage requirements calculated by the long-instruction encoder to specify the corresponding instruction storage requirements in the architectural template. Thus, the template generator is used to specify a minimal datapath that meets the functional and performance requirements of the target application. As such, it also specifies an application-specific datapath having minimal cost. The generated architectural template is used by the area-estimation model to estimate the area, and hence the cost, of the minimal datapath required to execute the target application.

After calculating the execution time of the target application and estimating the area of the datapath, the designer should compare these to the performance and cost requirements of the target application. If the performance requirements are not met, the designer might choose to add more hardware resources, such as functional units or registers, to the datapath. Alternatively, the designer might choose to apply more compiler optimizations like loop unrolling or trace scheduling, which are not currently implemented in the compiler, to ensure that more efficient code is generated, or that more parallelism is exploited. On the other hand, if the cost requirements are not met, the designer might have to remove or reduce datapath components that are under-utilized. One way to reduce the instruction storage requirements is to use a smaller decoder memory and be more selective in which multi-op instructions to store in decoder memory. In most cases, however, the designer has to find a compromise between performance and cost. For some applications, higher costs may have to be tolerated to meet performance requirements. In other applications, performance may have to be sacrificed to meet cost requirements.

6.1.3 Results and Analysis

To illustrate how they can be used to specify application-specific datapaths, the tools and methodology were applied to the set of kernel and application benchmarks, respectively. For each set of benchmarks, the aim was to find the datapath that is best suited for the computational and performance requirements of the entire set. Since different applications require different architectural resources, the architectural configuration with the minimal resources to meet the requirements of all applications together was used. This configuration is called the *aggregate architecture*.

Typically, a practical design begins with a specific set of cost and performance requirements that have to be met. The cost requirements are normally specified in terms of an area budget that largely depends on the required functionality and level of performance. In general, the more functions that need to be performed, or the higher the precision required, the larger the chip area. On the other hand, performance is usually specified by a hard, real-time window that is application dependent. For example, an embedded DSP must complete the processing of the current signal sample, and possibly other earlier samples, before the next sample arrives. Otherwise, real-time performance is compromised. Depending on the application, different processors will have different performance requirements. For example, an echo canceller for a telephone line has less stringent performance requirements than a real-time MPEG decoder. Moreover, the performance of an embedded processor must only be sufficient to meet the real-time constraints of the applications. By using a minimal set of functional units and running the processor at a clock rate that meets the performance requirements of the application, cost and power consumption can also be reduced. This is in contrast to general-purpose processors, where higher performance is a goal in and of itself, and where cost and power consumption are less significant design factors.

In describing the different datapath configurations, the following notation will be used: *P<n>m<u1>a<u2>d<u3>f<u4>p<u5>* [17]. Here, *<n>* is the number of functional units in the architecture; *<u1>* is the number of memory units; *<u2>* is the number of address units; *<u3>* is the number of integer units; *<u4>* is the number of floating-point units; and *<u5>* is the number of control units, which, for the UTDSP, is always set to one. For both the kernels and the applications, the lower performance bound is set by the *P5m1a1d1f1p1* configuration, where only one functional unit of each type is used, and up to five operations can be executed in parallel. On the other hand, the upper cost bound is set by the *P9m2a2d2f2p1* configuration, where two functional units of each type are used, and up to nine operations can be executed in parallel. The tools were therefore used to examine the architectural alternatives that fall between these two extremes, as well as their impact on performance and cost.

Starting with the *P5m1a1d1f1p1* configuration, a different functional unit was added to the architecture, one at a time, and the impact of adding the functional unit on performance and cost was assessed. Since only one control unit can be used in the UTDSP, four architectural configurations were possible: *P6m1a1d1f2p1*, *P6m1a1d2f1p1*, *P6m1a2d1f1p1*, and *P6m2a1d1f1p1*.

Tables 6.1 and 6.2 show the execution times and area estimates associated with each of these configurations for both the kernel and application benchmarks, respectively. In both Tables, the area estimates do not include the area of data-memory since it is the same for all configurations. Including the area of data-memory will also skew the results since it would dominate the area of the data-path. Both Tables also show the performance gain (PG), cost increase (CI), and performance/cost ratio (PCR) for each configuration relative to the baseline configuration. The baseline configuration — *P5m1a1d1f1p1* in this case — is the configuration from which the other configurations are derived. Thus, the PG is the speedup in execution time relative to the baseline configuration; the CI is the ratio of the area of a configuration relative to that of the baseline configuration; and the PCR is the ratio of the PG to CI. Although the interpretation of these metrics depends on the application, a configuration having a PCR greater than 1.0 generally indicates the configuration is cost-effective. Among the derived configurations, the one with the greatest PCR is the most cost-effective.

Table 6.1 shows that, for the kernels, the greatest PCR is achieved with the *P6m2a1d1f1p1* configuration. Given that the inner loops of most kernels access memory frequently, and that there is great potential for exploiting parallelism in accessing memory, it is not surprising that the greatest improvement in performance is achieved when a memory unit is added to the datapath. On the other hand, Table 6.2 shows that, for the applications, the greatest PCR is achieved with the *P6m1a2d1f1p1* configuration. This is due to the performance gained from adding an address unit and exploiting the parallelism in addressing operations. Although a similar level of performance could be gained by adding a memory unit and exploiting the parallelism in memory accesses, the corresponding increase in cost results in a lower PCR.

Having found the configuration with the greatest PCR, this configuration is used as the next baseline. New configurations are then derived from the baseline by adding new functional units to it. For each of the derived configurations, the associated PG, CI, and PCR are then calculated, and the configuration with the greatest PCR is once again chosen as the next baseline. In both Tables, the different baseline configurations are shown in bold. The process of deriving new configurations and selecting new baselines is essentially a pruning process that enables the exploration of the architectural space quickly. The process ends when the cost constraints are exceeded. Recall that for this study, the cost constraint is set by the *P9m2a2d2f2p1* configuration.

Architectural Configuration	Execution Time (Cycles)	Area (mm^2)	PG	CI	PCR
P5m1a1d1f1p1	169060	2.16	1.00	1.00	1.00
P6m1a1d1f2p1	156772	2.88	1.08	1.33	0.81
P6m1a1d2f1p1	166410	3.13	1.02	1.45	0.70
P6m1a2d1f1p1	167756	2.22	1.01	1.03	0.98
P6m2a1d1f1p1	145214	2.23	1.16	1.03	1.13
P7m2a1d1f2p1	131894	2.94	1.10	1.32	0.83
P7m2a1d2f1p1	142564	3.20	1.02	1.43	0.71
P7m2a2d1f1p1	140497	2.24	1.03	1.01	1.03
P8m2a2d1f2p1	114889	2.96	1.22	1.32	0.93
P8m2a2d2f1p1	136811	3.21	1.03	1.43	0.72
P9m2a2d2f2p1	111203	3.93	1.03	1.33	0.78

Table 6.1: Impact of different architectural configurations customized for the kernel benchmarks on performance and cost.

Architectural Configuration	Execution Time (Cycles)	Area (mm^2)	PG	CI	PCR
P5m1a1d1f1p1	6126057	4.29	1.00	1.00	1.00
P6m1a1d1f2p1	6112731	5.40	1.00	1.26	0.80
P6m1a1d2f1p1	5758261	5.13	1.06	1.20	0.89
P6m1a2d1f1p1	5736836	4.28	1.07	1.00	1.07
P6m2a1d1f1p1	5720373	4.55	1.07	1.06	1.01
P7m1a2d1f2p1	5723500	5.39	1.00	1.26	0.80
P7m1a2d2f1p1	5362081	5.14	1.07	1.20	0.89
P7m2a2d1f1p1	5211918	4.32	1.10	1.01	1.09
P8m2a2d1f2p1	5175303	5.43	1.01	1.26	0.80
P8m2a2d2f1p1	4836278	5.25	1.08	1.22	0.89
P9m2a2d2f2p1	4799663	6.36	1.01	1.21	0.83

Table 6.2: Impact of different architectural configurations customized for the application benchmarks on performance and cost.

Tables 6.1 and 6.2 also show that, beyond a certain point, adding more functional units is no longer cost-effective. For example, for both the kernels and applications, none of the configurations with eight or nine functional units have PCR's that are greater than 1.0. For these configurations, the cost increase is much greater than the performance gain.

Once all baseline architectures have been found, their overall impact on performance and cost can be compared. Figures 6.2 and 6.3 show the performance/cost graphs for the different baseline configurations,

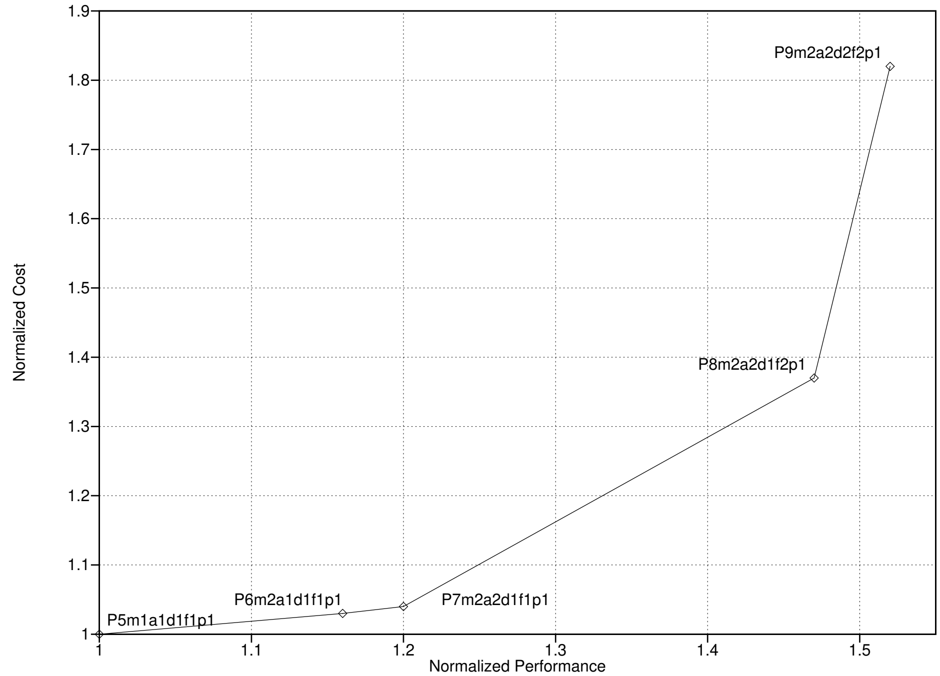


Figure 6.2: Impact of different architectural configurations on performance and cost for the kernel benchmarks.

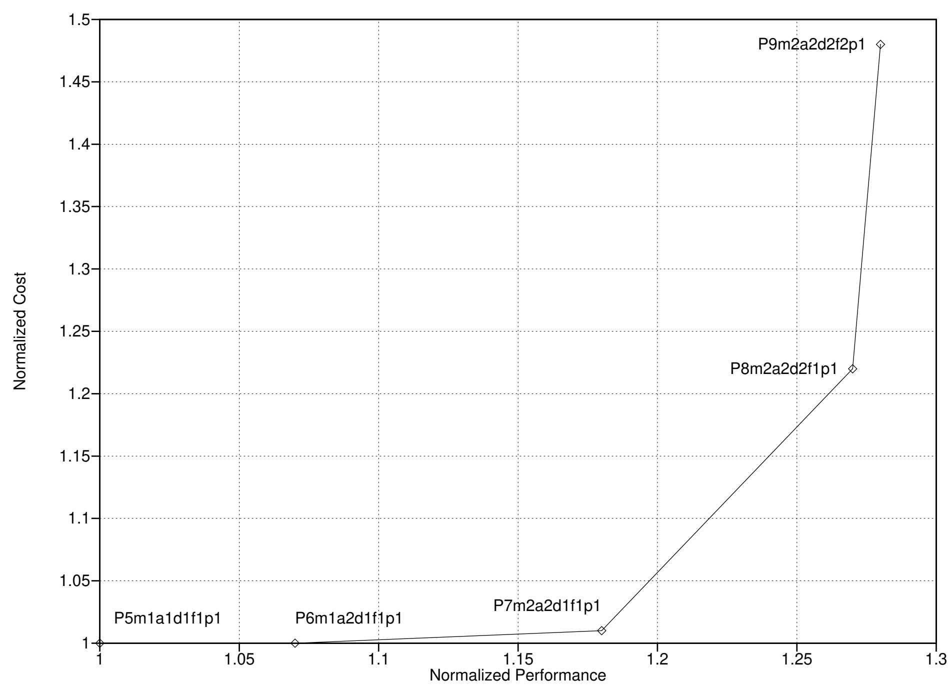


Figure 6.3: Impact of different architectural configurations on performance and cost for the application benchmarks.

for each of the kernel and application benchmarks, respectively. The performance and cost of each configuration is shown relative to the *P5m1a1d1f1p1* configuration. Using these graphs, the designer can make a reasoned choice between different configurations, and assess how each compares in relation to the performance and cost constraints.

6.2 Modifiable Instruction Sets

Similar to classical VLIW instruction sets, the UTDSP instruction set consists of a set of simple, RISC-like machine operations and a long-instruction template that specifies the way operations may be scheduled for parallel execution. The long-instruction template typically consists of several fields that can each be used to specify the execution of a class of operations. This makes the UTDSP a flexible target for HLL compilers, and simplifies the generation and optimization of assembly code, as well as the exploitation of parallelism. It also simplifies the underlying decode and control logic since every field in the instruction template is associated with a specific functional unit, and the exploitation of parallelism is relegated to the compiler.

Unlike classical VLIW instruction sets, however, the UTDSP instruction set is not limited to a fixed set of machine operations or a fixed instruction template. Depending on the functional requirements of the target applications, machine operations can easily be added to, or removed from, the instruction set and marked for execution on specific classes of functional units. The instruction template can also be modified to meet the performance requirements of the target applications. For example, operation fields may be added to, or removed from, the instruction template to exploit different levels of parallelism. The compiler uses the specified set of machine operations and the instruction template to generate multi-op instructions that exploit parallelism and enhance the execution performance of the target applications.

The UTDSP instruction set also achieves higher levels of code density than classical VLIW instruction sets by using single-word, multi-op pointers in instruction memory to represent long, multi-op instructions in decoder memory. Since it may not always be possible to store all the multi-op instructions generated by the compiler in decoder memory, only the most-frequently executed instructions — the ones that have the greatest impact on performance — are stored. As applications change, the set of multi-op instructions stored in decoder memory also changes. A multi-op pointer can therefore represent virtually any multi-op instruction in decoder memory. In addition to improving code density, this provides great flexibility in encoding different combinations of multi-op instructions. In this sense, a multi-op pointer may be regarded as a special *meta-instruction* whose semantics change as the contents of the decoder memory change. This also gives rise to the notion of a *modifiable instruction set* that can be customized to meet the functional,

```
MU0:  ld.s     (a2),f2
MU1:  ld.s     (a3),f3
AU0:  inc      a2,a0,a2
AU1:  inc      a3,a0,a3
FU0:  fmadd.s  f2,f3,f4,f4
```

(a)

```
DU0:  div.d    d5,d3,d7
DU1:  movi.d   #1,d8
```

(b)

Figure 6.4: Multi-op instructions generated by the UTDSP compiler. (a) A multi-op instruction commonly found in DSP instruction sets. (b) A multi-op instruction not commonly found in DSP instruction sets.

performance, and cost requirements of a class of target applications. In this sense, the compiler is not only a tool for generating code, but is also a tool for synthesizing application-specific instruction sets.

Figure 6.4 shows two examples of multi-op instructions generated by the UTDSP compiler. In Figure 6.4 (a), the multi-op instruction is taken from an inner loop of the `adpcm` application. The instruction performs two memory loads, two pointer-update operations, and a floating-point multiply-accumulate. This is a classic example of the type of instructions found in DSP instruction sets and used in the inner loops of common DSP algorithms. Like the instructions found in DSP instruction sets, this multi-op instruction is tightly encoded since it may be represented by a multi-op pointer that occupies a single word in instruction memory. However, unlike the instructions found in DSP instruction sets, the different operations may operate on any of their associated registers instead of being limited to a small set of registers or accumulators. This enables the compiler to make better use of the registers and results in more efficient code. Furthermore, each multi-op instruction can specify the execution of any combination of up to nine operations, and this also provides the compiler with great flexibility to exploit available parallelism.

Figure 6.4 (b) shows another multi-op instruction taken from an inner loop of the `edge_detect` application. This instruction performs an integer division and initializes an integer register with a constant value. Typically, such an instruction is not found in DSP instruction sets, and must be executed as two, separate, sequential instructions. However, this particular instruction is used in the body of a loop that accounts for a significant proportion of the execution time. Clearly, then, by adding this instruction to the instruction set, the execution performance of the application is enhanced. In general, adding an application-specific instruction to the instruction set enhances execution performance if the instruction is on the critical execution path, and is executed with high frequency.

One drawback of this approach to synthesizing application-specific instructions is that multi-op instructions are not orthogonal. This means that different instances of the same multi-op instruction, where the same operations but different operands are used, will be encoded differently, and will be stored in different long-instruction words in decoder memory. One way to minimize the effect of this problem on cost is to exploit redundancy. If several identical instances of a multi-op instruction are used in an application, only one instance of the instruction needs to be stored in decoder memory. Similarly, if part of a multi-op instruction — a sub-instruction — is common to several multi-op instructions, and double-cluster encoding is used, only one instance of that sub-instruction needs to be stored in decoder memory.

Finally, a major factor that affects the way both operations and multi-op pointers are encoded is the length of the instruction word used. The next section examines the impact of the instruction word length on performance and cost in more detail.

6.3 Impact of Instruction Word Length on Cost and Performance

In addition to exploring the performance/cost trade-offs of different architectural configurations, the designer can also explore the trade-offs in using a specific instruction word length for encoding operations and multi-op pointers. While Appendix A examines the different ways in which operations can be encoded using different instruction word lengths, this section examines the impact of the instruction word length on the encoding of multi-op pointers and operations. It also examines their resulting impact on performance and cost, assuming the datapath width is the same for the different instruction word lengths.

6.3.1 Cost and Performance Issues

The UTDSP architecture has thus far been assumed to use 32-bit instruction words. As early RISC processors demonstrated, 32-bit instruction words are sufficient for encoding uni-op instructions. The results in Chapter 5 also demonstrated that 32-bit instruction words are sufficient for encoding multi-op pointers for the UTDSP. Thus, provided that sufficient instruction bandwidth is available, using longer instruction words will not improve performance. However, using longer instruction words increases cost since more memory would be required to store instructions in both instruction memory and decoder memory. For these reasons, instruction words that are longer than 32 bits will not be considered in this section.

On the other hand, using shorter instruction words will degrade execution performance for at least three reasons. First, fewer bits would be available for encoding address fields in multi-op pointers. As a result, fewer multi-op instructions can be accessed in decoder memory, and less parallelism can be exploited. Second, uni-op instructions that use immediate operands have to be stored as two-word instructions. This increases execution time since more time would be required to fetch, decode, and execute these instructions. Finally, fewer bits will be available to encode register specifiers, and hence, less registers can be used. As the results in Chapter 4 demonstrated, this degrades execution performance, since more memory-access operations will be needed to move data between memory and the smaller register set. Nonetheless,

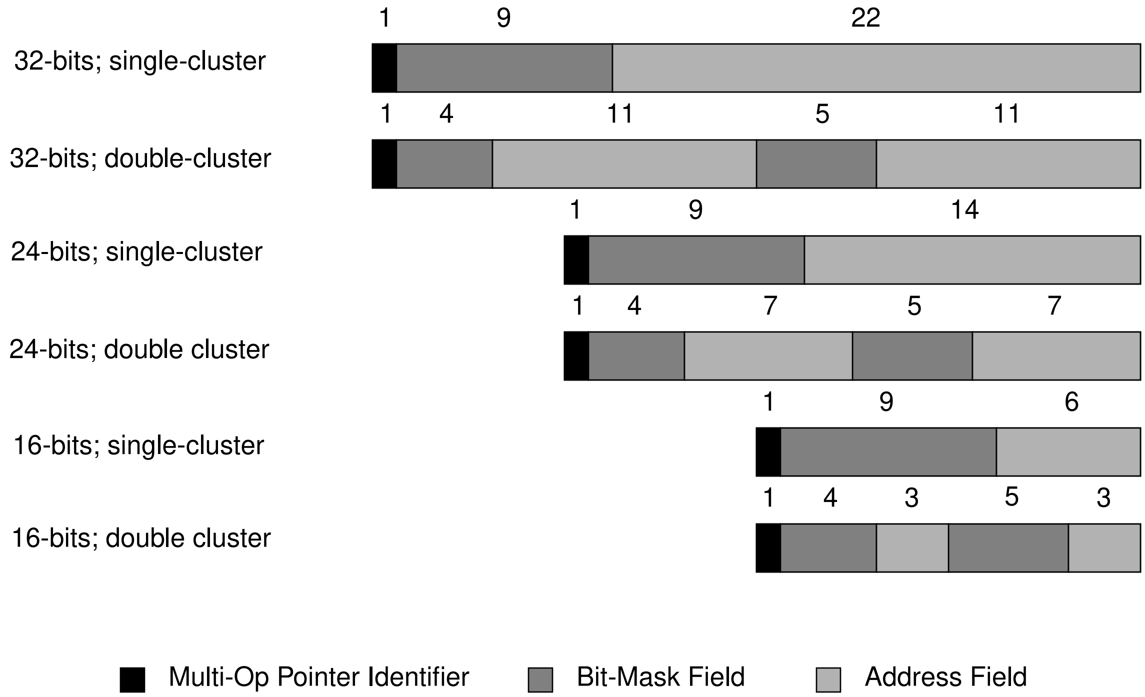


Figure 6.5: Different ways of encoding multi-op pointers using different instruction word

using shorter instruction words reduces instruction bandwidth requirements, and may reduce cost by using less memory. The remainder of this section therefore examines the impact of using 24-bit and 16-bit instruction words on performance and cost, and compares the results to those achieved using 32-bit instruction words.

6.3.1.1 Multi-Op Pointers

Figure 6.5 shows the different ways in which multi-op pointers can be encoded using different instruction word lengths. For each instruction word length, two encoding styles are shown: one for single-cluster encoding, where a single bit mask and a single address field are used, and one for double-cluster encoding, where two bit masks and two address fields are used. The most significant bit of each instruction word is used to identify the word as a multi-op pointer. Since the UTDSP architecture consists of nine functional units, nine bits are also needed for encoding the bit mask in each pointer, regardless of whether single-cluster encoding or double cluster encoding is used. The remaining bits in each pointer are used for encoding the address fields. As the length of the instruction word is reduced, the number of bits available for encoding the address fields is also reduced, and fewer multi-op instructions can be accessed in decoder memory. When double-cluster encoding is used, the available bits must be divided into two address fields, and this further reduces the number of multi-op instructions that can be accessed in decoder memory. By limiting

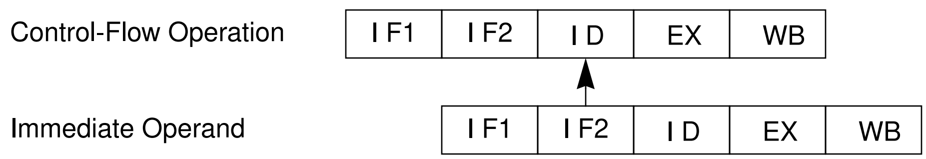


Figure 6.6: Forwarding the immediate operand of a two-word control-flow operation.

the number of multi-op instructions that can be accessed from decoder memory, less parallelism can be exploited. Thus, to limit the impact on performance when the number of multi-op instructions in decoder memory is limited, more selective storage of multi-op instructions is required. One way to achieve this is to use profiling information to store the most-frequently executed instructions in decoder memory.

6.3.1.2 Operations with Immediate Operands

Using short instruction words also affects the encoding of operations that use immediate operands. Typically, such operations must be encoded as two words, where the first word specifies the opcode and any register operands, and the second word specifies the immediate operand. More time is therefore needed to fetch, decode, and execute such operations, and execution performance suffers accordingly.

The UTDSP operations that use immediate operands are divided into three groups: control-flow operations (`beqz.{a|d|s}`, `bnez.{a|d|s}`, `jsr`, and `jmp`), low-overhead looping operations (`do`, `do.d`, `do.a`, and `rep`), and move-immediate operations (`movi.{a|d|s})`. For the control operations, the immediate operand represents a target address. For low-overhead operations, the immediate represents the address of the last instruction in a repeated block (for `do` operations) or the iteration count (for the `rep` operation). Finally, for move-immediate operations, the immediate represents an integer constant.

To minimize the latency associated with a control-flow operation, it is necessary to determine its target address as early as possible in the pipeline. Similarly, to minimize any latency associated with a low-overhead looping operation, the appropriate loop-control registers must be initialized with the address of the last instruction in a repeated block, or with an iteration count, as early as possible in the pipeline. In both cases, the immediate operands should be available during the instruction-decode (ID) stage of their respective operations. Since both operation types have similar pipeline requirements, the term *control-flow operation* will be used to denote both operation types, unless otherwise stated.

Figure 6.6 shows the overlap between the different pipeline stages of a control-flow operation and its corresponding immediate operand. For the immediate operand to become available during the instruction-decode stage associated with the operation, it must be forwarded[1] during the second instruction-fetch stage (IF2). Such forwarding can be done as soon as the operation requiring the immediate is decoded. However, the forwarding can only proceed if the immediate has already been fetched from memory. Since this can

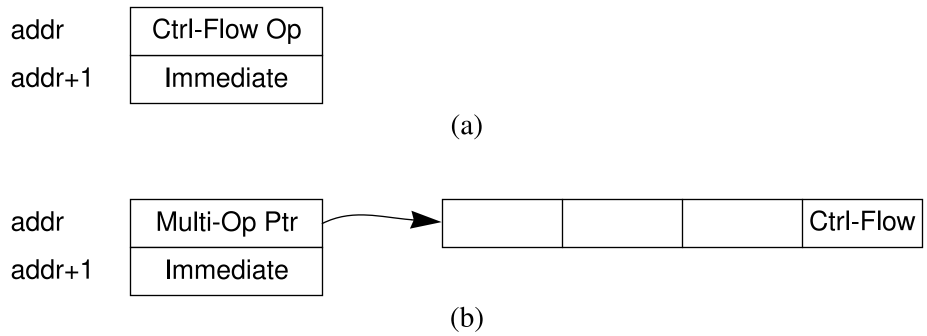


Figure 6.7: Encoding a two-word control-flow operation. (a) Uni-op control-flow instruction. (b) Control-flow operation part of a multi-op instruction.

only be done during the first instruction-fetch stage (IF1), the compiler must always store the immediate operand associated with a control-flow operation in instruction memory.

Figure 6.7 shows the two ways a control-flow operation and its immediate operand can be encoded and stored in instruction memory. Figure 6.7 (a) shows the way a uni-op instruction consisting of a control-flow operation is stored as two words in instruction memory. In this case, the first word is used to store the opcode and any register operands, while the second word is used to store the corresponding immediate operand. On the other hand, Figure 6.7 (b) shows the way a multi-op instruction containing a control-flow operation is represented in instruction memory. Recall that multi-op instructions in decoder memory are represented by a multi-op pointer in instruction memory. However, in this case, the multi-op instruction is represented by two words in instruction memory. While the first word is used to store the multi-op pointer, the second word is used to store the immediate operand associated with the control-flow operation. Since only a single immediate operand is associated with the multi-op instruction, it is assumed that none of the other operations in the multi-op instruction use an immediate field. The case where multiple operations in a multi-op instructions use immediate operands will be examined at the end of this sub-section. Finally, since the immediate operands associated with control-flow operations must always be stored as separate words in instruction memory, instructions that contain a taken branch incur a two-cycle latency, while instructions that contain a non-taken branch incur a one-cycle latency. Similarly, instructions that contain a low-overhead looping operation incur a one-cycle latency.

Unlike the immediate operands associated with control-flow operations, immediate operands associated with move-immediate operations should become available by the execute (EX) stage associated with these operations. Figure 6.8 shows the overlap between the different pipeline stages operating on a move-imme-

1. In the computer architecture literature, the term *forwarding* is commonly used to describe the passing of data from an instruction at a given pipeline stage to another instruction at an earlier pipeline stage. However, for the remainder of this section, the term will be used to denote the passing of an immediate operand to its associated instruction when the instruction is at a later pipeline stage, and the two are stored in different instruction words.

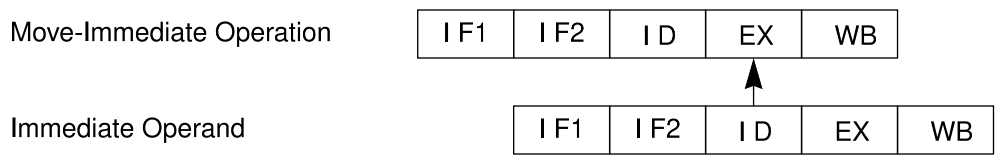


Figure 6.8: Forwarding the immediate operand of a two-word move-immediate operation.

diate operation and its corresponding immediate operand. This Figure shows that the immediate operand must be forwarded from its corresponding instruction-decode (ID) stage. Such forwarding can be communicated once the move-immediate operation has been decoded. However, for the forwarding to proceed, the immediate should have been fetched by the end of its second instruction-fetch (IF2) stage. This makes it possible to fetch the immediate operand from either instruction memory or decoder memory. In the latter case, the immediate operand would be scheduled with other operations, and possibly other immediate operands, into a long-instruction word. The scheduling of immediate operands in long-instruction words provides a great deal of flexibility since it enables the simultaneous fetching of several immediate operands from decoder memory. It also enables the representation of several immediate operands by a single multi-op pointer stored in instruction memory. Finally, scheduling immediate operands into long instruction words may, in some cases, eliminate the latency introduced by having to store an immediate operand in a separate word of memory. This is achieved when the immediate operand is scheduled into an empty operation field, or when scheduling the immediate displaces an operation that can be scheduled into another long-instruction word without introducing a new long-instruction word.

Figure 6.9 shows the four ways a move-immediate operation and its immediate operand can be encoded and stored in instruction memory. Figure 6.9 (a) shows the move operation and its immediate operand encoded as two separate words. This represents the case when the move operation constitutes a uni-op instruction, and the immediate operand cannot be scheduled into a long-instruction word. This encoding also results in a uni-op move-immediate instruction incurring a latency of one clock cycle. Figure 6.9 (b) shows the move operation specified by a multi-op pointer, while its immediate operand is encoded in a separate word of instruction memory. In this case, the move operation is scheduled in a long-instruction word and stored in decoder memory. Since the immediate operand occupies a separate word in instruction memory, the multi-op instruction containing the move operation will also incur a single clock cycle latency. Figure 6.9 (c) shows the move operation encoded as a separate word in instruction memory, and its immediate operand specified by a multi-op pointer. In this case, the move operation is also a uni-op instruction whose immediate is scheduled into a long-instruction word and stored in decoder memory. If scheduling the immediate operand does not introduce a new long-instruction word, there will be no latency associated

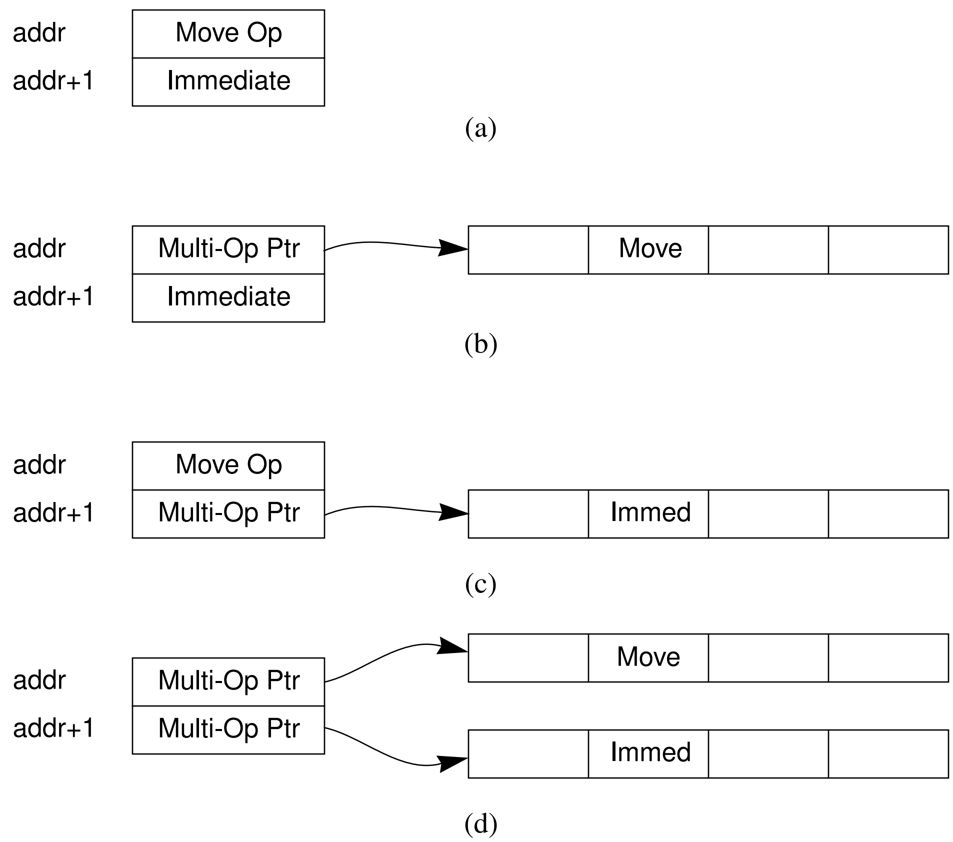


Figure 6.9: Encoding a two-word move-immediate operation. (a) Uni-Op move-immediate instruction. (b) Move operation part of a multi-op instruction. (c) Immediate operand part of a multi-op instruction. (d) Move operation and immediate operand part of multi-op instructions.

with executing the move instruction. This is due to the clock cycle that would have otherwise been wasted being used to execute useful operations scheduled in the same long-instruction word as the immediate operand. Finally, Figure 6.9 (d) shows both the move operation and its immediate operand specified by multi-op pointers. In this case, both the move operation and the immediate operand are each scheduled into different long-instruction words. Once again, if scheduling the immediate does not introduce a new long instruction, there will be no latency associated with the multi-op instruction containing the move operation. Otherwise the multi-op instruction will incur a single clock cycle latency.

An interesting case occurs when a control-flow operation may be scheduled into the same multi-op instruction as one or more move-immediate operations. Since the immediate operand associated with a control-flow operation must be stored in a separate word in instruction memory, it is not possible to schedule it into the same long-instruction word used to specify the immediate operands associated with the move-immediate operations. For this reason, the compiler must never schedule control-flow operations with move-immediate operations. Since control-flow operations delineate the end of a basic block, the

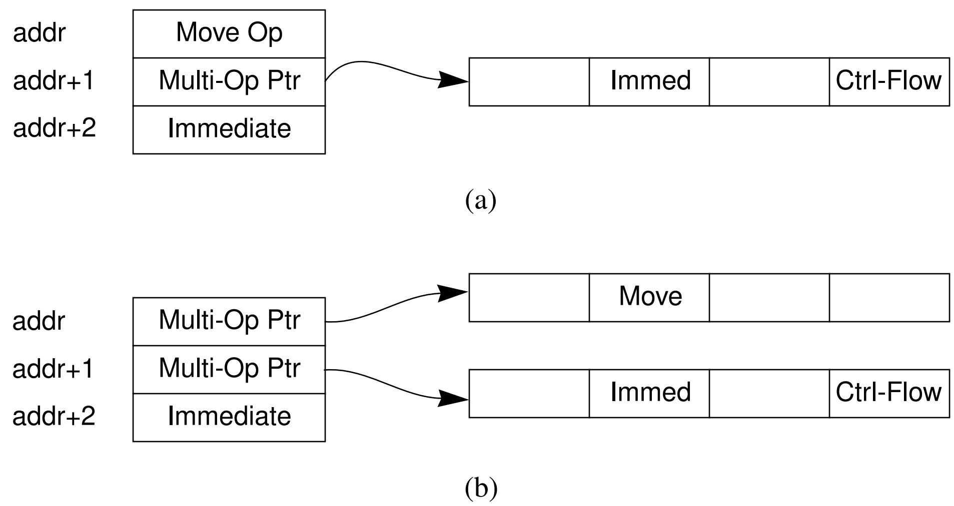


Figure 6.10: Encoding two-word move-immediate and control-flow operations. (a) Uni-op move-immediate instruction. (b) Move operation part of a multi-op instruction.

compiler should therefore schedule the move-immediate operations first, and the control-flow operation second. However, to eliminate the latency associated with the multi-op instruction containing the move-immediate operations, and to reduce the storage requirements of instruction memory, it is possible to schedule the control-flow operation and the immediate operands associated with the move-immediate operations into the same long-instruction word.

Figure 6.10 shows the two ways in which move-immediate operations and control-flow operations can be stored in instruction memory. In Figure 6.10 (a), the move operation is encoded as a separate word of instruction memory. This represents the case where the move operation cannot be scheduled with other operations into the same multi-op instruction. The move operation is followed by a multi-op pointer that specifies the immediate operand associated with the move operation, and a control-flow operation. This pointer is followed by the immediate operand associated with the control-flow operation. In Figure 6.10 (b), the move operation is specified by a multi-op pointer. In this case, the move operation is scheduled with other operations, and possibly other move-immediate operations, into the same long-instruction word. The multi-op pointer is followed by another multi-op pointer that specifies the immediate operand associated with the move operation, and the control-flow operation. Finally, this pointer is followed by the immediate operand associated with the control-flow operation. In both cases, there is no latency associated with the move-immediate operation. On the other hand, a single clock cycle latency is associated with the control-flow operation.

6.3.1.3 Register Specifiers

Appendix A describes the way machine operations can be encoded using instruction words with different lengths. In general, as the length of the instruction word decreases, fewer bits can be used to encode register specifiers, and fewer registers can therefore be used by the compiler for evaluating expressions or storing temporary values. Recall, from Chapter 4, that using a small number of registers can degrade performance significantly. However, when 24-bit instruction words are used, five bits may still be used for specifying source and destination registers. Since this enables 32 registers to be used in each register bank — the same number of registers available using 32-bit instruction words — performance is not degraded. On the other hand, when 16-bit instruction words are used, only three bits can be used for specifying source and destination registers. Since this enables only eight registers to be used in each register bank, performance will be degraded. This shows that the reduction in cost achieved by using shorter instruction words must be traded-off with the performance achieved using a small number of registers.

6.3.2 Experimental Methodology

Section 6.3.1 described the performance and cost issues related to using short-length instruction words. To evaluate the impact of reducing the instruction word length on overall cost and performance, the operation compaction pass in the compiler back-end was modified to ensure that the appropriate instruction fields are scheduled with immediate operands, and that control-flow operations are not scheduled with move-immediate operations. The long instruction encoder was also modified to handle smaller decoder memory banks and use profiling to store multi-op instructions with high execution frequency in decoder memory. Recall that using shorter multi-op pointers provides more limited access to multi-op instructions in decoder memory. This in turn reduces the exploitation of parallelism, and performance suffers accordingly. Profiling reduces the impact on performance by ensuring that only the most frequently-executed instructions are stored in decoder memory.

Using 32-bit words as a baseline, the impact of using 24-bit and 16-bit words on the performance and cost of the UTDSP architecture was assessed. Since the area occupied by the datapath and the data memory banks is not affected by the instruction word length, cost is measured in terms of the area needed to store a program in both instruction memory and decoder memory. This area is proportional to the product of the instruction word count and the instruction word length. On the other hand, execution performance is measured by the number of clock cycles required to execute a program.

6.3.3 Results and Analysis

This section shows the impact of the instruction word length on the instruction word count, the instruction storage requirements, and execution performance of the kernel and application benchmarks, respectively. For each benchmark, results are shown for three instruction word lengths and two multi-op encoding styles. The instruction word lengths are 32 bits, 24 bits, and 16 bits, while the multi-op pointer encoding styles are single-cluster encoding and double-cluster encoding. All results are normalized to the baseline case where 32-bit words are used with double-cluster encoding.

6.3.3.1 Impact on Instruction Word Counts

Figures 6.11 and 6.12 show the impact of using different instruction word lengths and multi-op pointer encoding styles on the instruction word count for the kernel and application benchmarks, respectively. The instruction word count refers to the total number of instruction words, in both instruction and decoder memory banks, required to store a program. Recall that the instruction word count is affected by the length of the instruction word used. For example, the instruction word count increases when immediate operands must be stored in separate instruction words, or when additional long-instruction words must be used for scheduling control-flow and move-immediate operations. The instruction word count also increases when a small number of registers is used, and additional memory-access operations must be used to move data between data-memory and the small set of registers.

For both the kernels and the applications, and independent of the instruction word lengths used, Figures 6.11 and 6.12 show that double-cluster encoding achieves smaller instruction word counts than single-cluster encoding. This result was demonstrated in Chapter 5, and is mainly due to the denser packing of instructions in decoder memory and the smaller number of NOPs stored in it.

When single-cluster encoding is used, Figure 6.11 shows that for the kernels, using 24-bit instruction words results in instruction word counts that are 1.3 – 1.5 times greater (1.4 times greater, on average) than those achieved using the baseline. This increase is mainly due to the additional storage needed for immediate operands, and the long-instruction words introduced by having to reschedule move-immediate and control-flow operations. On the other hand, using 16-bit instruction words results in instruction word counts that are 1.3 – 1.8 times greater (1.5 times greater, on average) than those achieved using the baseline. This is also due to the additional storage needed for immediate operands, the rescheduling of move-immediate and control-flow operations into different long-instruction words, and the additional memory-access operations. Figure 6.11 also shows that using 16-bit instruction words results in instruction word counts that are an average of 1.1 times greater than those achieved using 24-bit instruction words. This is mainly due to the additional memory-access operations that are generated as a result of having a small number of registers.

When double-cluster encoding is used, Figure 6.11 also shows that using 24-bit instruction words results in instruction word counts that are 1.1 – 1.2 times greater (1.1 times greater, on average) than those achieved using the baseline. This is mainly due to the additional overhead associated with storing the

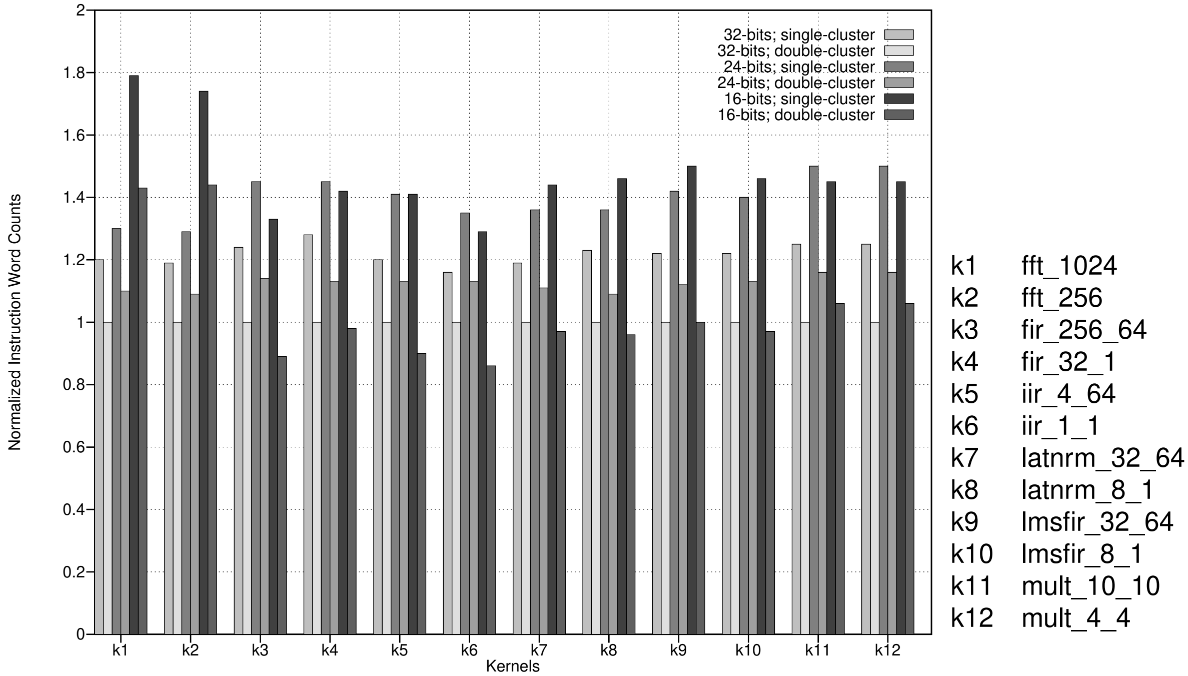


Figure 6.11: Instruction word counts for kernel benchmarks.

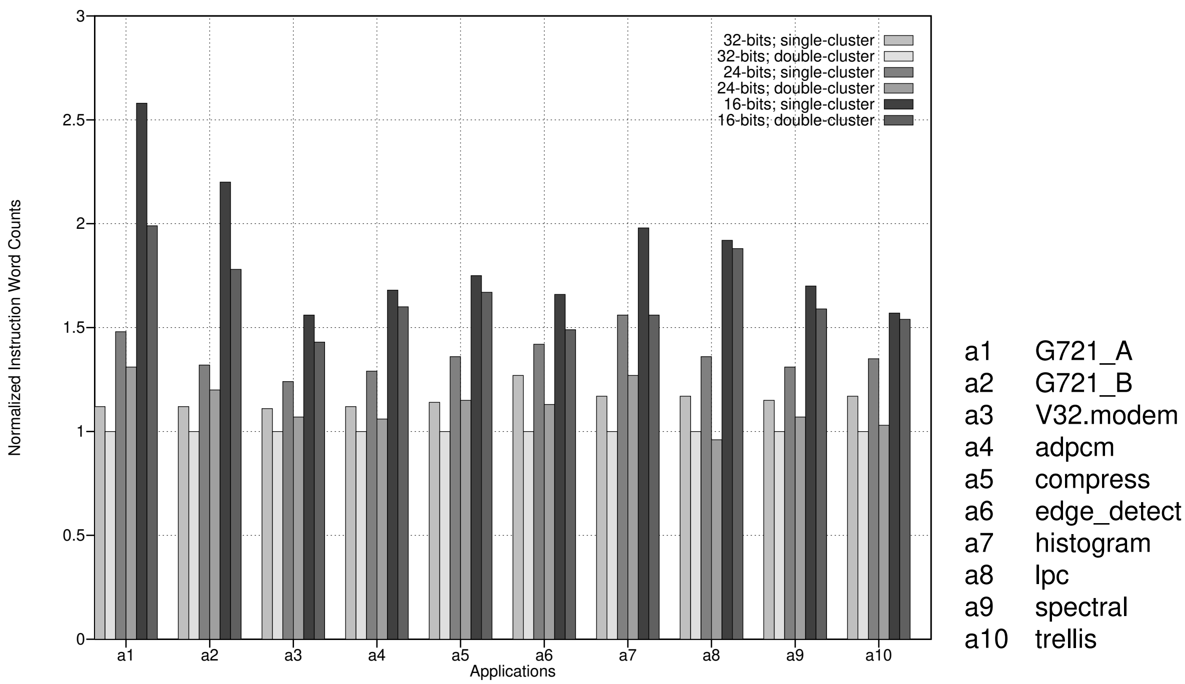


Figure 6.12: Instruction word counts for application benchmarks.

immediate operands of uni-op instructions in separate words, and the rescheduling of move-immediate and control-flow operations into different long-instruction words. On the other hand, using 16-bit words results in instruction word counts that vary between 0.9 – 1.4 times (1.0 times, on average) those achieved using the baseline. The differences in instruction word counts are due to two factors working to offset each other. On the one hand, the overhead associated with storing immediate operands increases the word counts. The introduction of additional long-instruction words to handle the rescheduling of move-immediate and control-flow operations also increases the word counts. On the other hand, the smaller number of multi-op instructions that can be accessed in decoder memory reduces the number of multi-op pointers in instruction memory and the number of NOPs in decoder memory, and this reduces the word counts. Depending on which factor is more dominant for a given kernel, the instruction word counts will vary accordingly. Finally, Figure 6.11 shows that using 16-bit instruction words results in instruction word counts that are a factor of 0.8 – 1.3 times (0.9 times, on average) those achieved using 24-bit instruction words. The lower instruction word counts are mainly due to the smaller number of multi-op instructions that can be accessed in decoder memory, which reduces the number of multi-op pointers stored in instruction memory and the number of NOPs stored in decoder memory. The only exceptions are `fft_1024` and `fft_256` (k1 and k2), where additional memory-access operations offset the reductions due to the smaller number of multi-op instructions in decoder memory. The effects of reducing the number of multi-op instructions stored in decoder memory on execution performance will be discussed in Section 6.3.3.3.

For the applications, when single-cluster encoding is used, Figure 6.12 shows that using 24-bit instruction words results in instruction word counts that are 1.2 – 1.6 times greater (1.4 times greater, on average) than those achieved using the baseline. This is mainly due to the overhead associated with storing the immediate operands of uni-op instructions in separate words. It is also due to having to schedule move-immediate operations and control-flow operations in different long-instruction words. Similarly, and also because of the additional memory-access operations, using 16-bit instruction words results in instruction word counts that are 1.6 – 2.6 times greater (1.8 times greater, on average) than those achieved using the baseline. Figure 6.12 also shows that using 16-bit instruction words results in instruction word counts that are 1.2 – 1.7 times greater (1.4 times greater, on average) than those achieved using 24-bit instruction words. This is mainly due to the additional memory-access operations which offset any reductions in instruction word counts that typically accompany the storage of fewer multi-op instructions in decoder memory.

When double-cluster encoding is used, Figure 6.12 also shows that using 24-bit words results in instruction word counts that are 1.0 – 1.3 times greater (1.1 times greater, on average) than those achieved using the baseline. This, again, is due to the additional words required for storing immediate operands for uni-op instructions, and for accommodating the additional long-instruction words that result from rescheduling move-immediate and control-flow operations. On the other hand, using 16-bit instruction words results in instruction word counts that are 1.4 – 2.0 times greater (1.6 times greater, on average) than those achieved using 32-bit instruction words. This is also due to the additional words required for storing immediates, accommodating additional long-instruction words, and additional memory-access operations. Finally, Figure 6.12 shows that using 16-bit words results in instruction word counts that are 1.2 – 2.0 times greater (1.5 times greater, on average) that those achieved using 24-bit instruction words. This, again, is due to the additional memory-access operations, which offset any reductions in instruction word counts that typically accompany the storage of fewer multi-op instructions in decoder memory.

6.3.3.2 Impact on Instruction Storage Requirements

While Figures 6.11 and 6.12 show the impact of the instruction word length on instruction word counts, they do not show its impact on cost. Recall that cost is related to the area occupied by instruction and decoder memory, and that it is proportional to the product of the instruction word count by the length of an instruction word.

Figures 6.13 and 6.14 show the impact of the different instruction word lengths and multi-op pointer encoding styles on the area occupied by the instruction and decoder memory banks for the kernel and application benchmarks, respectively. These graphs are essentially scaled versions of the graphs in Figures 6.11 and 6.12, where the scaling factors depend on the instruction word length. For 32-bit instruction words, the scaling factor is 1.00; for 24-bit instruction words, it is 0.75; and for 16-bit instruction words, it is 0.50. Thus, for a fixed instruction word count, reducing the instruction word length from 32 bits to 24 bits should reduce the area occupied by instruction and decoder memory by 25%. Further reducing the instruction word length to 16 bits should reduce the area by 50%. However, this represents an ideal case where the instruction word count remains constant, and is not affected by the length of the instruction word or the multi-op pointer encoding styles used. In reality, however, and as demonstrated in Section 6.3.3.1, the instruction word count is affected by a number of factors that are directly related to the length of the instruction word and the multi-op pointer encoding styles used. Thus, although reducing the length of an instruction word is one way of reducing the area occupied by instruction and decoder memory, the effects of reducing the instruction word length on the instruction word count may sometimes lead to less than ideal results. The results in Figures 6.13 and 6.14 are again normalized to the baseline case, where 32-bit words are used with double-cluster encoding.

For the kernels, Figure 6.13 shows that using 32-bit instruction words and single-cluster encoding increases the area occupied by instruction and decoder memory by a factor of 1.2 – 1.3 (1.2 on average). This is mainly due to the less dense packing of operations, and the greater number of NOPs, in decoder

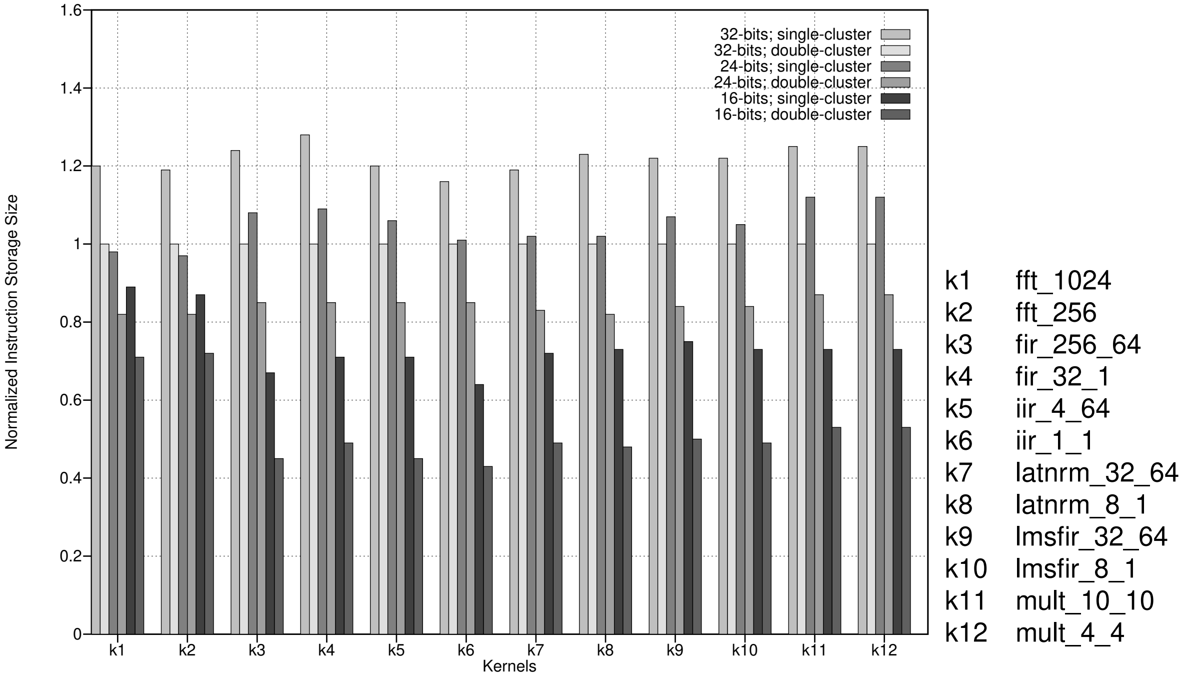


Figure 6.13: Instruction storage requirements for kernel benchmarks.

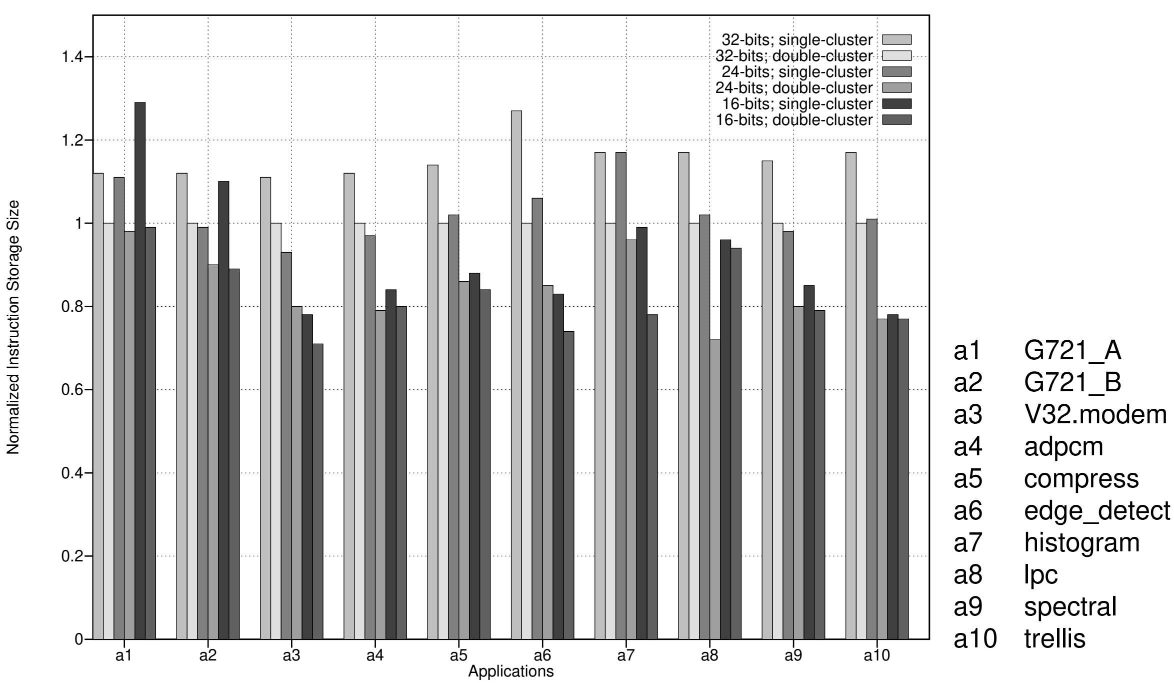


Figure 6.14: Instruction storage requirements for application benchmarks.

memory. Using 24-bit instruction words with single-cluster encoding also increases the area by an average factor of 1.1. In this case, the increase in the instruction word count has a greater impact on area than the reduction in instruction word length. On the other hand, using 24-bit instruction words with double-cluster encoding reduces the area by a factor of 0.1 – 0.2 (0.2 on average). In this case, the reduction in instruction word length has a greater impact on area than the increase in the instruction word count. When 16-bit instruction words are used with single-cluster encoding, the area is reduced by a factor of 0.1 – 0.4 (0.3 on average). Here, the reduction in instruction word length again has a greater impact on area than the increase in the instruction word count. Finally, when 16-bit instruction words are used with double-cluster encoding, the area is reduced by a factor of 0.3 – 0.6 (0.5 on average).

For the applications, Figure 6.14 shows that using 32-bit instruction words with single-cluster encoding increases the area occupied by instruction and decoder memory by a factor of 1.1 – 1.3 (1.2 on average). This is mainly due to the greater number of multi-op pointers stored in instruction memory and the less dense packing of operations in decoder memory. Using 24-bit instruction words with single-cluster encoding has a mixed effect on the area, and causes it to vary by a factor of 0.9 – 1.2 (1.0 on average). This shows that, on average, the increase in the instruction word count is offset by the reduction in instruction word length. On the other hand, using 24-bit instruction words with double-cluster encoding reduces the area by as much as a factor of 0.3 (0.2 on average). In this case, the reduction in instruction word length has a greater impact on area than the increase in the instruction word count. When 16-bit instruction words are used with single-cluster encoding, the impact on area varies by a factor of 0.8 – 1.3 (0.9 on average). This shows that, on average, the increase in the instruction word count is offset by the reduction in the instruction word length. Finally, when 16-bit instruction words are used with double-cluster encoding, the area is reduced by as much as a factor of 0.3 (0.2 on average). This again shows that the increase in the instruction word count is offset by the reduction in the instruction code length.

6.3.3.3 Impact on Execution Performance

The results thus far have demonstrated the effects of reducing the instruction word length on the instruction word count and the area occupied by the instruction and decoder memory banks. However, reducing the instruction word length also affects execution performance. In general, reducing the instruction word length will degrade performance for at least four reasons. First, the need to store the immediate operands of uni-op instructions in separate words of instruction memory introduces a one-cycle latency for each of these instructions. Second, the rescheduling of move-immediate and control-flow operations into different long-instruction words also increases execution time. Third, the shorter address fields in multi-op pointers limit access to multi-op instructions in decoder memory, and this degrades performance by limiting the amount of parallelism that can be exploited. This is especially true when double-cluster encoding is used, since the number of bits available for encoding address fields are reduced even further. Finally, using 16-bit instruction words limits the number of registers that can be used, and this introduces additional memory-access operations.

Figures 6.15 and 6.16 show the effects of reducing the instruction word length and using different multi-op pointer encoding styles on the execution performance of the kernel and application benchmarks, respectively. The results are normalized to the baseline case where 32-bit words are used with double-cluster encoding. This represents an upper bound on execution performance since using 32-bit words enables immediate operands to be encoded into the same instruction word as their corresponding operations. It also enables move-immediate and control-flow operations to be scheduled into the same long-instruction words. Using 32-bit multi-op pointers also provides for sufficiently long address fields to access a greater space in decoder memory. This enables the storage of more multi-op instructions in decoder memory, and hence, the exploitation of more parallelism. Finally, using 32-bit instruction words provides enough bits to specify enough registers, and this eliminates the need for additional memory-access operations.

For both the kernels and applications, using 24-bit and 16-bit instruction words generally degrades performance. For the kernels, Figure 6.15 shows that using 24-bit instruction words, with both single- and double-cluster encoding, degrades performance by as much as a factor of 0.1. However, on average, it achieves a similar level of performance as using 32-bit instruction words with double-cluster encoding. The relatively low impact on performance is mainly due to the small size of the kernels which enables the short address fields of multi-op pointers to access most multi-op instructions, and hence, to maximize the exploitation of parallelism. On the other hand, using 16-bit instruction words has a significant impact on performance. When single-cluster encoding is used, performance is reduced by as much as a factor of 0.8 (0.6 on average). Similarly, when double-cluster encoding is used, performance is reduced by as much as a factor of 0.9 (0.8 on average). The more significant impact on performance is due to the shorter address fields which greatly limit the number of multi-op instructions that can be accessed in decoder memory, and the amount of parallelism that may therefore be exploited. Recall that when the length of the address fields is limited, profiling information can be used to ensure that only the most-frequently used multi-op instructions are stored in decoder memory. Another factor contributing to the degradation in performance is the execution of additional memory-access operations.

For the applications, Figure 6.16 shows that using 24-bit instruction words with single-cluster encoding degrades performance by an average factor of 0.1, while using 24-bit instruction words with double-cluster encoding degrades performance by as much as a factor of 0.2, and an average factor of 0.1. Here, the low impact on performance is mainly due to sufficiently long address fields that support the storage of a significant proportion of multi-op instructions in decoder memory, and the selective storage of the most fre-

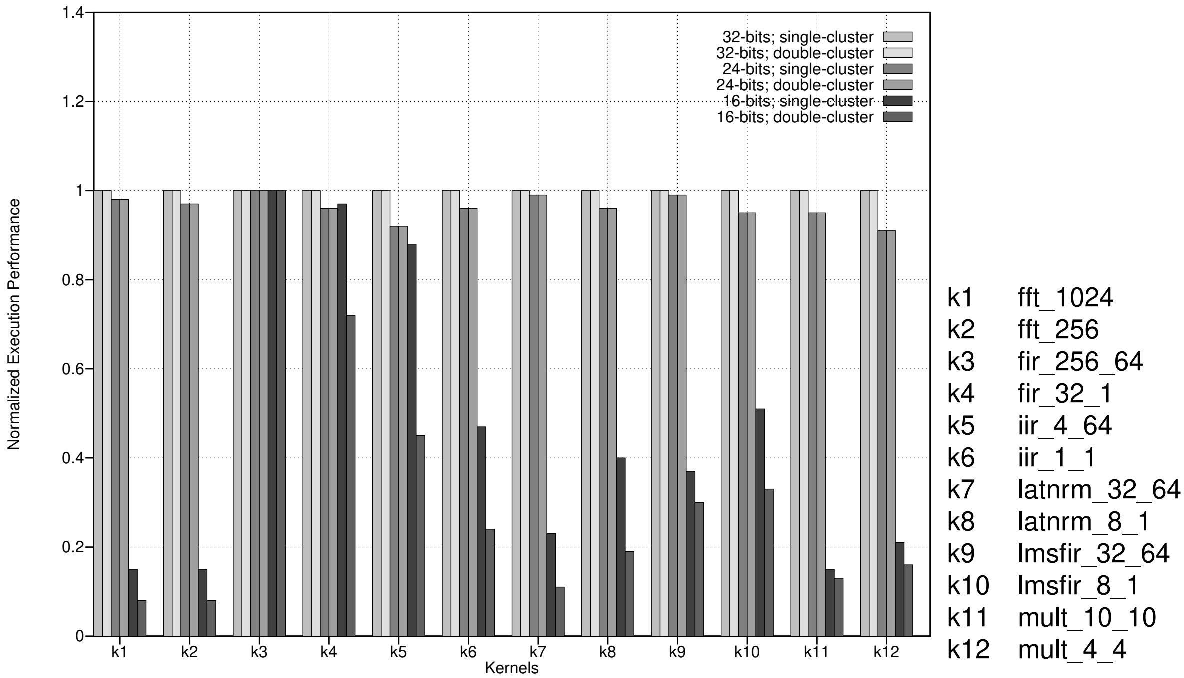


Figure 6.15: Impact of instruction word length on the performance of the kernel benchmarks.

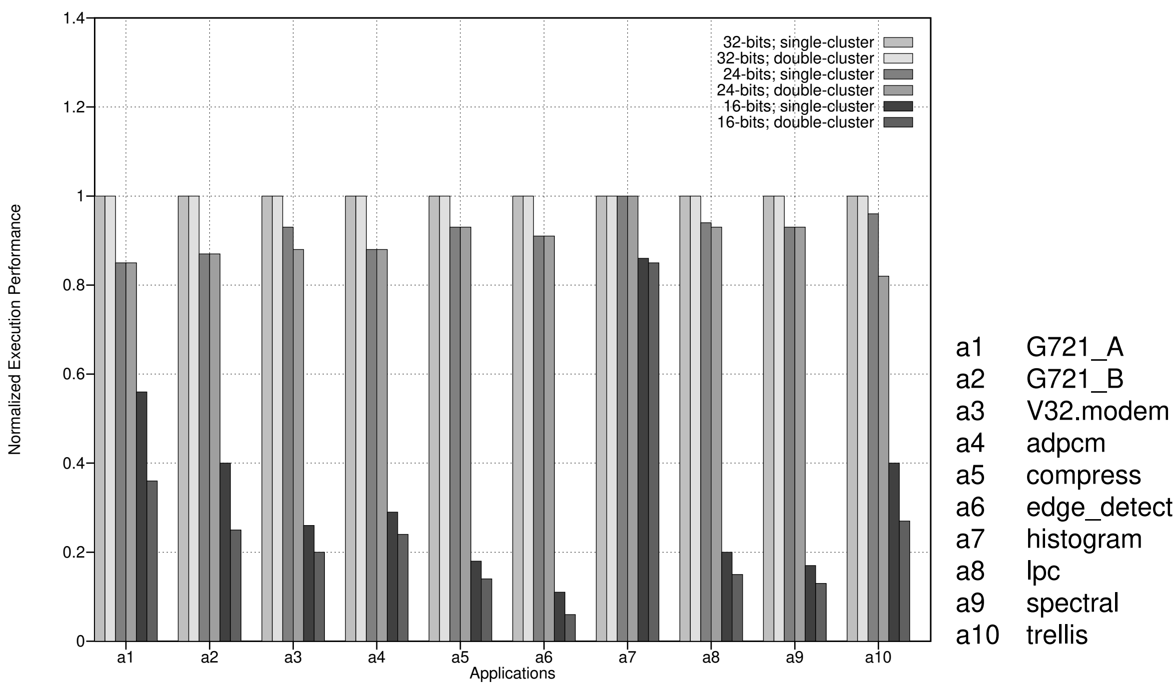


Figure 6.16: Impact of instruction word length on the performance of the application benchmarks.

quently executed multi-op instructions in decoder memory, which minimizes the impact on performance. On the other hand, using 16-bit instruction words with single-cluster encoding degrades performance by a factor of 0.1 – 0.9 (0.7 on average). Similarly, using 16-bit instruction words with double-cluster encoding degrades performance by a factor of 0.1 – 0.9 (0.8 on average). In both cases, the more significant degradation in performance is mainly due to the shorter address fields, which greatly limit the number of multi-op instructions that can be accessed in decoder memory, and the amount of parallelism that may therefore be exploited. Another factor contributing to performance degradation is the execution of the additional memory-access operations.

6.3.3.4 Overall Impact on Performance and Storage Requirements

Tables 6.3 and 6.4 summarize the impact of the instruction word length and the multi-op pointer encoding styles on the performance and storage requirements of the kernels and applications, respectively. The results are normalized to the baseline case where 32-bit words are used with double-cluster encoding. For every instruction word length and encoding style used, the performance gain (PG), cost increase (CI), and performance/cost ratio (PCR) are shown. The performance gain is the ratio of the execution time achieved with a given instruction word length and encoding style to that achieved with the baseline case. Similarly, the cost increase is the ratio of the area occupied by the instruction and decoder memory banks for a given instruction word length and encoding style to that for the baseline case. Finally, the performance/cost ratio for a given instruction word length and encoding style is just the ratio of the performance gain to the cost increase. Although the interpretation of these results depends to a large extent on the application, they can be used by the designer to determine the performance and cost trade-offs between different instruction word lengths and encoding styles. In general, a PCR that is greater than 1.0 indicates that the performance gain outweighs the increase in storage requirements. However, it remains the responsibility of the designer to determine whether the reduction in storage requirements justifies the degradation in performance for a given application. In the following discussion, all results correspond to geometric mean values.

For the kernels, Table 6.3 shows that when 32-bit instruction words are used with single-cluster encoding, there is no degradation in performance, but there is a 22% increase in storage requirements. Similarly, when 24-bit instruction words are used with single-cluster encoding, there is a 4% decrease in performance and a 5% increase in storage requirements. Since the increase in storage requirements outweighs the performance gain, there is no advantage in using either case. When 24-bit instruction words are used with double-cluster encoding, there is a 4% decrease in performance and a 16% decrease in storage requirements. Since the reduction in storage requirements outweighs the reduction in performance, this case is cost-effective. On the other hand, when 16-bit instruction words are used with single-cluster encoding,

Kernels	32-bits; single-cluster encoding			24-bits; single-cluster encoding			24-bits; double-cluster encoding			16-bits; single-cluster encoding			16-bits; double-cluster encoding		
	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR
k1	1.00	1.20	0.83	0.98	0.98	1.00	0.98	0.82	1.20	0.15	0.89	0.17	0.08	0.71	0.11
k2	1.00	1.19	0.84	0.97	0.97	1.00	0.97	0.82	1.18	0.15	0.87	0.17	0.08	0.71	0.11
k3	1.00	1.24	0.81	1.00	1.08	0.93	1.00	0.85	1.18	1.00	0.67	1.49	1.00	0.45	2.22
k4	1.00	1.28	0.78	0.96	1.09	0.88	0.96	0.85	1.13	0.97	0.71	1.37	0.72	0.49	1.47
k5	1.00	1.20	0.83	0.92	1.06	0.87	0.92	0.85	1.08	0.88	0.71	1.24	0.45	0.45	1.00
k6	1.00	1.16	0.86	0.96	1.01	0.95	0.96	0.85	1.13	0.47	0.64	0.73	0.24	0.43	0.56
k7	1.00	1.19	0.84	0.99	1.02	0.97	0.99	0.83	1.19	0.23	0.72	0.32	0.11	0.49	0.22
k8	1.00	1.23	0.81	0.96	1.02	0.94	0.96	0.82	1.17	0.40	0.73	0.55	0.19	0.48	0.40
k9	1.00	1.22	0.82	0.99	1.07	0.93	0.99	0.84	1.18	0.37	0.75	0.49	0.30	0.50	0.60
k10	1.00	1.22	0.82	0.95	1.05	0.90	0.95	0.84	1.13	0.51	0.73	0.70	0.33	0.49	0.67
k11	1.00	1.25	0.80	0.95	1.12	0.85	0.95	0.87	1.09	0.15	0.73	0.21	0.13	0.53	0.25
k12	1.00	1.25	0.80	0.91	1.12	0.81	0.91	0.87	1.05	0.21	0.73	0.29	0.16	0.53	0.30
Min.	1.00	1.16	0.78	0.91	0.97	0.81	0.91	0.82	1.05	0.15	0.64	0.17	0.08	0.43	0.11
Max.	1.00	1.28	0.86	1.00	1.12	1.00	1.00	0.87	1.20	1.00	0.89	1.49	1.00	0.72	2.22
Geometric Mean	1.00	1.22	0.82	0.96	1.05	0.92	0.96	0.84	1.14	0.36	0.74	0.49	0.23	0.52	0.44

Table 6.3: Impact of instruction word length and multi-op pointer encoding style on performance and instruction storage requirements of kernel benchmarks.

there is a 64% decrease in performance and a 26% decrease in storage requirements. Similarly, when 16-bit words are used with double-cluster encoding, there is a 77% reduction in performance and a 48% reduction in storage requirements. Since the reduction in performance outweighs the reduction in storage requirements in both cases, it is clear that neither case is cost-effective. These results suggest that using 24-bit instruction words with double-cluster encoding is the most cost-effective alternative for the kernel benchmarks.

For the applications, Table 6.4 shows that when 32-bit words are used with single-cluster encoding there is no degradation in performance, but there is a 15% increase in storage requirements. Similarly, when 24-bit instruction words are used with single-cluster encoding, there is an 8% decrease in performance and a 2% increase in storage requirements. Since the increase in storage requirements outweighs the performance gain in both cases, there is no advantage to using either case. When 24-bit instruction words are used with double-cluster encoding, there is a 10% decrease in performance and a 16% decrease in storage requirements. Since the reduction in storage requirements outweighs the reduction in performance, this case is again cost-effective. On the other hand, when 16-bit instruction words are used with single-cluster encoding, there is a 71% decrease in performance and an 8% decrease in storage requirements. Similarly,

Applications	32-bits; single-cluster encoding			24-bits; single-cluster encoding			24-bits; double-cluster encoding			16-bits; single-cluster encoding			16-bits; double-cluster encoding		
	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR	PG	CI	PCR
a1	1.00	1.12	0.89	0.85	1.11	0.77	0.85	0.98	0.87	0.56	1.29	0.43	0.36	0.99	0.36
a2	1.00	1.12	0.89	0.87	0.99	0.88	0.87	0.90	0.97	0.40	1.10	0.36	0.25	0.89	0.28
a3	1.00	1.11	0.90	0.93	0.93	1.00	0.88	0.80	1.10	0.26	0.78	0.33	0.20	0.71	0.28
a4	1.00	1.12	0.89	0.88	0.97	0.91	0.88	0.79	1.11	0.29	0.84	0.35	0.24	0.80	0.30
a5	1.00	1.14	0.88	0.93	1.02	0.91	0.93	0.86	1.08	0.18	0.88	0.20	0.14	0.84	0.17
a6	1.00	1.27	0.79	0.91	1.06	0.86	0.91	0.85	1.07	0.11	0.83	0.13	0.06	0.74	0.08
a7	1.00	1.17	0.85	1.00	1.17	0.85	1.00	0.96	1.04	0.86	0.99	0.87	0.85	0.78	1.09
a8	1.00	1.17	0.85	0.94	1.02	0.92	0.93	0.72	1.29	0.20	0.96	0.21	0.15	0.94	0.16
a9	1.00	1.15	0.87	0.93	0.98	0.95	0.93	0.80	1.16	0.17	0.85	0.20	0.13	0.79	0.16
a10	1.00	1.17	0.85	0.96	1.01	0.95	0.82	0.77	1.06	0.40	0.78	0.51	0.27	0.77	0.35
Min.	1.00	1.11	0.79	0.85	0.93	0.77	0.82	0.72	0.87	0.11	0.78	0.13	0.06	0.71	0.08
Max.	1.00	1.27	0.90	1.00	1.17	1.00	1.00	0.98	1.29	0.86	1.29	0.87	0.85	0.99	1.09
Geometric Mean	1.00	1.15	0.87	0.92	1.02	0.90	0.90	0.84	1.07	0.29	0.92	0.31	0.21	0.82	0.25

Table 6.4: Impact of instruction word length and multi-op pointer encoding style on performance and instruction storage requirements of application benchmarks.

when 16-bit instruction words are used with double-cluster encoding, there is a 79% reduction in performance and an 18% reduction in storage requirements. For these two cases, the reduction in performance outweighs the reduction in storage requirements, and it is clear that neither case is cost-effective. These results also suggest that using 24-bit instruction words with double-cluster encoding is the most cost-effective alternative for the application benchmarks.

6.4 Summary

This chapter described the tools and methodology used to customize the UTDSP datapath and instruction set to the functional, performance, and cost requirements of a target application, or group of applications. It also examined the impact of reducing the instruction word length on performance and cost.

The tools include a machine-description file that is used to specify a template architecture consisting of the functional units in the datapath. An operation-definition file is also used to specify a set of machine operations that are mapped for execution on different functional units. Using the specified template architecture and set of machine operations, the compiler can generate matching long instructions with operation fields associated with each functional unit. The instruction-set simulator is used to simulate the execution of compiler-generated code on the template architecture, and measure its execution performance. The code generated by the compiler is also processed by the long-instruction encoder and the architectural template generator. These are used to calculate the corresponding instruction storage requirements, and specify the minimal architectural resources needed to meet the functional requirements of an application, respectively. The architectural template generator also specifies the storage requirements for each data-memory bank, and uses the instruction storage requirements calculated by the long-instruction encoder to specify the instruction storage requirements. Finally, the area estimation model is used to estimate the area, and hence the cost, of the specified datapath.

Using these tools, the designer is able to compare the performance and cost of the specified datapath to the performance and cost constraints of the target application. If the performance requirements are not met, the designer can add more hardware resources to the datapath. The designer can also apply more compiler optimizations to exploit more parallelism. On the other hand, if the cost requirements are not met, the designer can remove under-utilized datapath components. In most cases, however, the designer must find a compromise between performance and cost. For some applications, higher costs may have to be tolerated to meet performance requirements. For other applications, performance may have to be sacrificed to meet cost requirements. To illustrate how the tools and the design methodology can be used to explore the architectural space, different configurations of the UTDSP architecture were customized to execute an aggregate workload of the kernel and application benchmarks, respectively.

In addition to customizing the UTDSP datapath, the designer can customize the UTDSP instruction set to the requirements of the target application. The UTDSP instruction set consists of a set of simple, RISC-like machine operations and an instruction template that specifies the way operations may be scheduled for parallel execution. Machine operations can easily be added to, or removed from, the instruction set to meet the functional requirements of the target application. The instruction template can also be modified by adding or removing operation fields to support different levels of parallelism. The UTDSP instruction set also achieves high code densities by using multi-op pointers in instruction memory to represent multi-op instructions in decoder memory. Since multi-op pointers can represent virtually any multi-op instruction in decoder memory, they are analogous to special meta-instructions whose semantics change with the contents of decoder memory. This not only provides great flexibility in matching the available parallelism in a target application, but is also the basis for a modifiable instruction set that can be customized to the performance requirements of the application.

One factor that greatly affects the way uni-op instructions and multi-op pointers are encoded is the length of the instruction word. Since the area needed to store a program is proportional to the product of the instruction word length by the instruction word count, using short instruction words is one way to reduce system costs. However, when reducing the instruction word length causes uni-op instructions with immediate operands to be encoded in separate words, the instruction word count may actually increase. The instruction word count can also increase when additional long-instruction words are needed to reschedule move-immediate and control-flow operations. Finally, the instruction word count can increase if there are not enough bits in an instruction word to specify enough registers, and additional memory-access operations are needed to move data between data-memory and a small set of registers. Typically, the increase in the instruction word count is offset by the smaller number of multi-op instructions in decoder memory. This reduces the number of multi-op pointers in a program, and hence, the number of multi-op pointers in instruction memory, and the number of NOPs in decoder memory. Depending on which of these conflicting factors is more dominant, system costs will vary accordingly.

Using short instruction words also degrades performance for at least three reasons. First, fewer bits are available for encoding address fields in multi-op pointers. As a result, fewer multi-op instructions can be accessed in decoder memory, and less parallelism can be exploited. This problem is exacerbated when multiple pointers are stored in a single instruction word, since the length of the different address fields is reduced. Second, uni-op instructions that use immediate operands will have to be represented by two words in instruction memory. This increases execution time because of the added latency in fetching, decoding, and executing such instructions. However, for some operations, like the move-immediate operation, this latency can sometimes be hidden by scheduling the immediate operand into a long-instruction word containing other operations. For other immediate operations, such as control-flow operations, this is not possible. Furthermore, having to reschedule control-flow operations and move-immediate operations into different long-instruction words can introduce additional instructions. Finally, using short instruction words reduces the number of bits that can be used to specify registers. With a smaller set of registers, more memory-access operations must be executed to move data between data-memory and the small register set.

To assess the impact of using short instruction words on performance and cost, the compiler back-end was modified to schedule immediate operands, and to avoid scheduling control-flow and move-immediate operations into the same long-instruction word. The long-instruction encoding pass was also modified to handle the shorter address fields in multi-op pointers and the resulting limited access to decoder memory. To minimize the impact on performance due to having fewer multi-op instructions in decoder memory, profiling was used to ensure that the most frequently-executed instructions were stored in decoder memory. Finally, while performance was measured by execution time, cost was measured by the area needed to store a program in both instruction memory and decoder memory. The impact of using 24-bit and 16-bit instruction words, as well as using single-cluster and double-cluster multi-op pointer encoding styles, on performance and cost was measured for the kernel and application benchmarks, respectively. All results were measured relative to the case when 32-bit instruction words were used with double-cluster encoding.

In terms of the impact on cost, using 24-bit instruction words achieved an average of 0.84 times the storage requirements for both the kernel and application benchmarks, respectively. On the other hand, using 16-bit instruction words achieved an average of 0.52 times the storage requirements for the kernels, and 0.82 times the storage requirements for the applications. In terms of the impact on performance, using 24-bit instruction words achieved an average of 0.96 times the performance for the kernels, and 0.90 times the performance for the applications. On the other hand, using 16-bit instruction words achieved an average of 0.23 times the performance for the kernels, and 0.21 times the performance for the applications.

These results show that using 24-bit instruction words helps reduce costs without significantly affecting performance. Since the length of the instruction word has a direct impact on both performance and storage requirements, it is an additional parameter that the designer can use to trade-off performance and cost. However, whether the reduction in cost is worth the degradation in performance remains highly application dependent, and must be resolved by the designer. The results here only illustrate the impact on the design space.

Chapter 7

Summary, Conclusions, and Future Work

This dissertation examined the issues surrounding the use of VLIW architectures in embedded DSP systems. In particular, it examined how the datapath and instruction-set architecture of a VLIW-based DSP architecture can be customized to the functional, performance, and cost requirements of embedded DSP applications.

7.1 Main Contributions

- Making the case for using VLIW-based architectures in embedded DSP systems.
- Developing an iterative, application-driven methodology for customizing the datapath and instruction set of a VLIW-based architecture to the functional, performance, and cost requirements of a target application, or group of applications.
- Developing a new set of tools for supporting the design methodology. Among these tools are an area-estimation model for measuring the cost associated with a datapath, a long-instruction encoding pass for measuring the instruction storage requirements of an application, and an architectural template generator for specifying a datapath with a minimal set of hardware resources that meets the functional requirements of a target application.
- Developing two new compiler algorithms for exploiting the dual data-memory banks of a model DSP architecture.
- Developing a new technique for encoding and decoding long instructions that minimizes instruction-storage and bandwidth requirements, while preserving the flexibility of the long-instruction format.

7.2 Summary and Conclusions

This section summarizes the main ideas and results of the dissertation.

7.2.1 Design Methodology and Tools

Chapter 2 described the application-driven design methodology and the tools used to design VLIW-based application-specific programmable processors. Starting with one or more applications written in the C programming language, and a set of performance and cost requirements, an optimizing compiler is used to compile the applications to a flexible target architecture. The resulting performance and cost are then measured, and the architecture is iteratively tuned to meet the performance and cost requirements of the application. If the performance requirements are not met, more hardware resources can be added to the architecture. Alternatively, different compiler optimizations can be used to expose and exploit more parallelism, or to exploit the underlying features of the target architecture. On the other hand, if the cost requirements are not met, hardware resources that are not used, or that are under-utilized, can be removed from the architecture. This process is repeated until the performance and cost requirements are met.

To support the application-driven design methodology, a number of tools were also developed. These included a suite of DSP benchmarks, consisting of 12 kernels and 10 applications; a VLIW-based model architecture; an optimizing C compiler; an instruction-set simulator for measuring execution performance;, and a datapath area-estimation model for measuring cost.

7.2.2 Exploiting Dual Data-Memory Banks

Chapter 3 described two algorithms for automatically allocating data to the dual data-memory banks of the UTDSP architecture. The algorithms, compaction-based (CB) data partitioning and partial data duplication, were integrated into the data allocation pass of the UTDSP compiler, enabling the automatic exploitation of dual data-memory banks without resorting to the use of compiler pragmas or language extensions.

The performance results indicate that CB partitioning is an effective technique for exploiting dual-memory banks. This is evident in the kernel benchmarks where CB partitioning improves performance by 25% – 70%. When a dual-ported memory is used as an ideal reference for how well the algorithms perform, the performance gains for all kernels were identical, or nearly identical for two of the kernels, to those attained in the ideal case. CB partitioning also improves the performance of the application benchmarks by 2% – 19%. In the ideal case, the performance gain is 3% – 22%. For one application, `lpc`, CB partitioning improves performance by only 4%. By adding partial data duplication, the performance improvement is an ideal 22%.

These results show that a HLL compiler can exploit dual data-memory banks without resorting to techniques such as compiler pragmas or language extensions. Among the many benefits of using the compiler to exploit dual data-memory banks are preservation of code portability, optimizing data accesses across an entire application, and achieving levels of performance similar to those of a dual-ported memory, at a fraction of the cost.

7.2.3 The Case for VLIW DSP Architectures

Chapter 4 examined the performance benefits and cost trade-offs of using the VLIW-based UTDSP architecture compared to more traditional, tightly-encoded DSPs. By constraining the functional units used in the UTDSP model architecture, two commercial DSPs, based on the Motorola DSP56001 and the Analog Devices ADSP-21020, were modeled. The compiler was also modified to only generate combinations of parallel operations that are supported by the instruction sets of both architectures, and to use the number of registers available in each architecture. However, the orthogonality between available registers and machine operations was not constrained, leading to more optimistic performance results than could actually be achieved on these architectures. To enable a fairer comparison to be made between the hardware-limited traditional DSP architectures and the UTDSP model architecture, two instances of the UTDSP architecture were modeled with the same hardware resources as the traditional DSPs. The main differences were that the VLIW-based models were given more registers, they were allowed more flexibility in exploiting parallelism, and they used long instructions instead of tightly-encoded ones.

The results showed that, for the kernel and application benchmarks, the restricted UTDSP architectures achieve 1.3 – 2.0 times the performance of the more traditional architectures. This is mainly due to their more flexible instruction set architecture and their larger set of data registers, which enable the compiler to exploit more parallelism and generate more efficient code. The results of an experiment measuring the effect of restricted register use in traditional DSP architectures also shows that the impact on performance is a factor of 1.4 – 2.0, suggesting that the actual performance achieved by the restricted UTDSP architectures is at least 1.8 – 2.8 times the performance of the more traditional architectures. However, since the experiment involved kernel code, the effect of more restrictive register use is expected to be more significant in larger applications. When the full-fledged, unrestricted UTDSP architecture is used, it achieves 2.0 – 3.0 times the performance of the more traditional architectures. When accounting for the effect of more restricted register use, the actual performance is closer to 2.8 – 4.2 times that of the more traditional architectures.

The results also showed that the restricted UTDSP architectures occupy 2.0 – 2.2 times the area of the more traditional architectures, while the full-fledged, unrestricted UTDSP architecture occupies 3.1 – 5.2 times their area. This was mainly due to the large memory needed to store long instructions and, in the case of the full-fledged UTDSP architecture, to its larger datapath. Thus, in their classical form, VLIW architectures are too expensive to use in cost-conscious, embedded DSP applications. However, the cost of a VLIW architecture can be reduced by efficiently encoding and storing long instructions, and by customizing the datapath to the functional and performance requirements of the target application.

7.2.4 Long-Instruction Decoding and Encoding for the UTDSP

Chapter 5 presented a new method for storing and encoding the long instructions of the VLIW-based UTDSP architecture. The main objective was to devise an inexpensive means of reducing instruction and bandwidth requirements without compromising the flexibility of the long-instruction format.

The UTDSP uses a decoder memory to implement a two-level storage scheme similar to the two-level control stores used in early microprogrammed computers. The first-level *instruction memory* is used to store uni-op instructions and pointers to multi-op instructions. The second-level *decoder memory* is used to store multi-op instructions in a packed format. The decoder memory consists of as many banks as there are functional units, and each bank is used to store the operations that are to be executed on the corresponding functional units. When a multi-op pointer is fetched from instruction memory, it is used to enable the banks that hold the operations in the corresponding multi-op instruction. The pointer also specifies an address that is applied to all enabled decoder-memory banks. While decoder-memory banks that are enabled return the appropriate operations, those that are not return NOPs. This way, operations fetched from decoder memory can be issued to the datapath in a long-instruction format. Since fetching instructions in a two-level instruction store requires accessing both instruction and decoder memory banks, an additional instruction-fetch pipeline stage was introduced to reduce the clock cycle time.

To reduce instruction storage requirements and achieve a dense packing of multi-op instructions in decoder memory, several techniques were used. These included storing multi-op instructions in decoder memory according to the priorities assigned to different operation fields; using the same long-instruction word to store multi-op instructions with mutually exclusive operation fields; using field clustering to increase instruction packing density; and storing unique instances of multi-op instructions or sub-instructions. Using these techniques, the two-level storage scheme was able to preserve the flexibility and performance of the long-instruction format at a fraction of the cost.

To assess its impact on instruction bandwidth, storage requirements, and execution performance, the two-level storage scheme was compared to a lower bound where all instructions are executed sequentially, and the corresponding operations are stored as uni-op instructions. The two-level storage scheme was also compared to an upper bound where all instructions are stored in a long-instruction format, and parallelism is fully exploited. In terms of the bandwidth needed to fetch instruction words from the first-level instruction memory, the two-level storage scheme requires the same amount of bandwidth as the lower bound.

For the UTDSP architecture, this is 32 bits. This represents a great saving when compared to the 288 bits required for the upper bound.

Compared to the lower bound, the two-level storage scheme requires 1.2 – 1.4 times the storage requirements for the kernels, and 1.2 – 1.3 times the storage requirements for the applications. The greater storage requirements of the two-level storage scheme are mainly due to the overhead introduced by the multi-op pointers in instruction memory and the NOPs in decoder memory. However, since the two-level storage scheme enables the exploitation of parallelism, it achieves 2.2 – 3.7 times the performance of the lower bound on the kernels (2.6 times the performance, on average), and 1.2 – 2.6 times the performance of the lower bound on the applications (1.8 times the performance, on average).

Compared to the upper bound, the two-level storage scheme requires 0.4 – 0.5 times the storage requirements for the kernels, and 0.2 – 0.4 times the storage requirements for the applications. This significant reduction in storage requirements is due to the efficiency of the instruction storage and packing techniques. At the same time, the reduction in storage requirements has little impact on performance, and the two-level storage scheme achieves 0.8 – 1.0 times the performance of the upper bound on the kernels (0.98 times the performance, on average), and 0.9 – 1.0 times the performance of the upper bound on the applications (0.95 times the performance, on average). The high levels of performance achieved by the two-level storage scheme is mainly due to its ability to preserve the flexibility of the long-instruction format.

Finally, the impact of selectively packing multi-op instructions in decoder memory was assessed for both the kernel and application benchmarks, respectively. In general, selective packing can be used when the size of decoder memory is restricted by processor area budgets, or when the number of bits available for encoding the address fields in a multi-op pointer is limited. By storing the most frequently executed multi-op instructions in decoder memory, the storage requirements of decoder memory can be reduced without significantly affecting execution performance. In fact, for both the kernels and the applications, the results show that 96% of the performance achieved when all multi-op instructions are stored in decoder memory can be achieved by only storing 46% of the multi-op instructions. Multi-op instructions that cannot be stored in decoder memory are typically serialized, and their constituent operations are stored as uni-op instructions in instruction memory.

7.2.5 Application-Specific Datapaths and Modifiable Instruction Sets

Finally, Chapter 6 described the way the tools can be used to customize the UTDSP datapath and instruction set to the functional, performance, and cost requirements of a target application, or group of applications. It also discussed the impact of the instruction word length on the encoding of multi-op pointers and uni-op instructions, and how these affect performance and cost.

To specify the functional units in the datapath, a machine-description file is used. Similarly, to specify a set of machine operations and map them for execution on the different functional units, an operation-description file is used. Using the specified datapath architecture and set of machine operations, the compiler can generate matching long instructions with operation fields associated with each of the functional units. The compiler also specifies the number of registers available in each register file. Once an initial datapath has been specified, it is used as the target architecture by the compiler. Using the code generated by the compiler the architectural template generator is used to specify the minimal set of architectural resources needed to meet the functional requirements of the application. It also specifies the storage requirements for each data-memory bank, and uses the instruction storage requirements calculated by the long-instruction encoder to specify the instruction storage requirements. The datapath specified by the architectural template generator is used by the area-estimation model to estimate the cost of the datapath. The instruction-set simulator is also used to measure its execution performance.

By comparing the performance and cost of the specified datapath to the requirements of the target application, the datapath can be tuned until it meets the requirements. For example, if the performance requirements are not met, more hardware resources can be added to the datapath. Alternatively, more compiler optimizations can be applied to enhance the run-time behavior of the application. On the other hand, if cost requirements are not met, under-utilized datapath components can be removed. This process is repeated until an architecture is found that meets the performance and cost requirements of the target application. To illustrate how the tools and the design methodology can be used to explore the architectural space, different configurations of the UTDSP architecture were customized to execute an aggregate workload of the kernel and application benchmarks, respectively.

In addition to customizing the UTDSP datapath, the UTDSP instruction set can be customized to the requirements of the target application. The UTDSP instruction set consists of a set of simple, RISC-like machine operations and an instruction template that specifies the way operations may be scheduled for parallel execution. Machine operations can easily be added to, or removed from, the instruction set to meet the functional requirements of the target application. Operation fields can also be added to, or removed from, the instruction template to support different levels of parallelism. The UTDSP instruction set also achieves high code densities by using multi-op pointers in instruction memory to represent multi-op instructions in decoder memory. Since multi-op pointers can represent virtually any multi-op instruction in decoder memory, they are analogous to special *meta-instructions* whose semantics change with the contents of decoder memory. This not only provides great flexibility in matching the available parallelism in a target application, but is also the basis for a *modifiable instruction set* that can be customized to the performance requirements of the application.

One factor that greatly affects the way both uni-op instructions and multi-op pointers are encoded is the length of the instruction word. The impact of using 24-bit and 16-bit instruction words on performance and cost was therefore examined. Cost was measured by the area needed to store a program, which is proportional to the product of the instruction word count by the instruction word length.

Using short instruction words generally reduces system costs because it has a more significant impact on area than the instruction word count. However, for programs with significant amounts of uni-op instructions with immediate operands, that can only be encoded using two, separate instruction words, the instruction word count may actually increase. The instruction word count can also increase due to the introduction of long-instruction words needed to reschedule move-immediate and control-flow operations. Finally, the instruction word count can increase if there are not enough bits in an instruction word to specify a large enough register file, and additional memory-access operations are generated to move data between data memory and the small set of registers. Typically, the increase in the instruction word count is offset by the smaller number of multi-op instructions in decoder memory, which also reduces the number of multi-op pointers in a program, and hence, the number of multi-op pointers in instruction memory, and the number of NOPs in decoder memory. Depending on which of these conflicting factors is more dominant, system costs will vary accordingly.

Using short instruction words also degrades performance for at least three reasons. First, fewer bits are available for encoding address fields in multi-op pointers. Fewer multi-op instructions can therefore be accessed in decoder memory, and less parallelism can be exploited. This problem is exacerbated when multiple pointers are stored in a single instruction word — as is done with double-cluster pointers — since the length of the different address fields is reduced. Second, uni-op instructions that use immediate operands will have to be represented by two words in instruction memory. This increases overall execution time because of the added latency in fetching, decoding, and executing such instructions. However, for some operations, like the move-immediate operation, this latency can sometimes be hidden by scheduling the immediate operand into a long-instruction word containing other operations. For other immediate operations, like control-flow operations, this is not possible. Finally, using short instruction words reduces the number of bits that can be used to specify registers. With a smaller set of registers, more memory-access operations must be executed to move data between data-memory and the small register set.

To assess the impact of using short instruction words on performance and cost, the compiler back-end was modified to schedule immediate operands, and to avoid scheduling control-flow and move-immediate operations into the same long-instruction words. The long-instruction encoding pass was also modified to handle the shorter address fields in multi-op pointers and the resulting limited access to decoder memory. To minimize the impact on performance due to having fewer multi-op instructions in decoder memory, profiling was used to ensure that the most frequently-executed instructions were stored in decoder memory. Finally, while performance was measured by execution time, cost was measured by the area needed to store a program in both instruction memory and decoder memory. The impact of using 24-bit and 16-bit instruction words, as well as using single-cluster and double-cluster multi-op pointer encoding styles, on performance and cost was measured for the kernel and application benchmarks, respectively. All results were measured relative to the case when 32-bit instruction words were used with double-cluster encoding.

In terms of the impact on cost, using 24-bit instruction words achieved an average of 0.84 times the storage requirements for both the kernel and application benchmarks, respectively. On the other hand, using 16-bit instruction words achieved an average of 0.52 times the storage requirements for the kernels, and 0.82 times the storage requirements for the applications. In terms of the impact on performance, using 24-bit instruction words achieved an average of 0.96 times the performance for the kernels, and 0.90 times the performance for the applications. On the other hand, using 16-bit instruction words achieved an average of 0.23 times the performance for the kernels, and 0.21 times the performance for the applications.

These results show that using 24-bit instruction words helps reduce costs without significantly affecting performance. Since the length of the instruction word has a direct impact on both performance and storage requirements, it is an additional parameter that the designer can use to trade-off performance and cost. However, whether the reduction in cost is worth the degradation in performance remains highly application dependent, and must be resolved by the designer. The results here only illustrate the impact on the design space.

7.3 Future Work

Following are some suggestions for future work that can be used to extend the scope of this dissertation.

- **Building a hardware prototype of the UTDSP architecture.** Thus far, the UTDSP has been a model architecture whose execution could only be simulated on the instruction-set simulator. Building a hardware prototype would enable the verification of the main ideas developed in the dissertation, especially those related to encoding and decoding long instructions. It would also enable more accurate measurements of performance, cost, and power consumption to be made.

- **Synthesizing HDL models from the architectural template generator.** Once an architecture that meets the specifications of an application has been found, the next step is to implement the architecture in hardware. One approach to doing this is to generate a behavioral model of the datapath in a hardware-description language (HDL) such as VHDL or Verilog, and then use commercial EDA tools to perform logic synthesis, placement, and routing. The behavioral model can easily be synthesized from the description of the architecture specified in the architectural template generator. The different components of the model can be selected from a library of parameterizable behavioral models corresponding to different datapath components. This not only simplifies the implementation of the hardware, but also results in a synthesizable core that can be reused, as a macro cell, in different system-on-a-chip designs.

- **Supporting specialized functional units.** In addition to standard functional units, such as integer ALUs or floating-point multipliers, specialized functional units can easily be added to the UTDSP datapath. One motivation for using such functional units is to implement time-critical operations in hardware. The specialized functional units can be implemented as reconfigurable logic blocks that execute special operations. Such operations can easily be generated by the compiler by mapping them to special function names, and using regular function calls to invoke them. When the compiler encounters a function call to one of these special functions, it would generate the corresponding special operation. The operation can then be scheduled like any other operation into a multi-op instruction, and issued in the appropriate operation field of the long-instruction word. This approach could be used, for example, to support the execution of SIMD operations. Although compiler technology will eventually enable the generation of SIMD operations directly, this approach can be used as a temporary solution.

- **Optimizing for power consumption.** In addition to high performance and low cost, low power consumption is an important requirement for portable embedded systems. A new tool is therefore needed for optimizing the datapath to meet the power requirements of target applications.

- **Refining the datapath area model.** The current area model is based on using technology-independent bit-area estimates of different datapath components. While this provides a first-order approximation of the area, and hence, the cost of the datapath, it is not very accurate. A more accurate model can be obtained, for example, by using the area associated with the HDL model of the datapath. Such a model would not only account for the datapath components, but would also consider the interconnections between the different components.

- **Upgrading the compiler.** A number of actions must be taken to update the current UTDSP compiler. First, the long-instruction encoding pass should be integrated into the optimizing back-end to enable the packing of multi-op instructions in decoder memory, and generate the associated multi-op pointers. Second, the poor scalar optimizations in the current version of the SUIF compiler front-end must also be fixed. Alternatively, a newer version of the SUIF compiler should be used. Third, the ability to use fixed-point arithmetic, including the automatic scaling of variables, should be added to the compiler.

Finally, new compiler optimizations, such as the generation of SIMD instructions, or the use of predicates, should be investigated. The associated implications on the hardware, as well as the overall impact on performance and cost, should also be studied.

- **Integrating the tools into a single interactive package.** The current set of tools consist of independent components that have mainly been used to validate the design methodology and measure its impact on performance and cost. Integrating current and future tools into a single interactive package will provide the necessary design environment for realizing an application-specific programmable processor that meets the functional, performance, cost, and power requirements of its target applications.

Appendix A

The UTDSP Machine Operation Set

This appendix describes the basic set of machine operations that can be executed on the different functional units of the UTDSP. These operations are initially generated by the UTDSP C compiler front-end before they are packed into long instructions for parallel execution in the optimizing back-end. Although the set of operations can be modified — operations can be added to, or removed from, the set to better match the functional requirements of the target application — the set of operations described in this appendix are common to most programmable processors. The appendix also describes the way operations can be encoded using different instruction word lengths.

A.1 UTDSP Machine Operations

The basic set of UTDSP machine operations is divided into five groups that are each mapped to execute on a different class of functional units. The five groups include memory, addressing, integer, floating-point, and control operations. Tables A.1 – A.5 describe the different operations in more detail.

Each table shows the syntax, functional description, and encoding format used with its associated set of operations. Section A.2 describes the encoding formats in greater detail. The tables also use the following notation: `ai`, `dj`, and `fk` are used to denote address, integer, and floating-point registers, respectively; `#X` is used to denote an immediate value; `(al)` and `(dm)` are used to denote the contents of registers `al` and `dm`, respectively; `D[am]` is used to denote a data-memory location whose address is stored in address register `am`; and `label` is used to denote the target address of a control-flow operation, or the address of the last instruction in a repeated block of instructions.

A.1.1 Memory Operations

Since the UTDSP is a load-store architecture, all memory accesses are made using load and store operations. Table A.1 shows the set of memory operations used in the UTDSP. For each of the address, integer, and floating-point register files, a separate set of memory operations is used. Memory operations are mapped to execute on memory units MU0 and MU1.

A.1.2 Addressing Operations

Table A.2 shows the set of addressing operations used in the UTDSP. All addressing operations operate on address registers, and include special operations that use modulo- and bit-reversed addressing. Although the UTDSP compiler is capable of using the modulo addressing operations, it currently cannot use the bit-reversed addressing operations. The addressing operations are mapped to execute on address units AU0 and AU1.

Having a dedicated set of addressing operations and registers makes it easy for the post-optimizer to perform alias analysis and to disambiguate pointer references. These are especially useful for the data partitioning and partial data duplication algorithms described in Chapter 3.

A.1.3 Integer Operations

Table A.3 shows the set of integer operations used in the UTDSP. In addition to the common set of arithmetic and logical operations, the integer operations include multiply-accumulate (`madd.d`) and multiply-subtract (`msub.d`) operations that are commonly used in DSP algorithms. All integer operations operate on integer registers, and are mapped to execute on integer units DU0 and DU1.

A.1.4 Floating-Point Operations

Table A.4 shows the set of floating-point operations used in the UTDSP. In addition to the common set of arithmetic operations, the floating-point operations include multiply-accumulate (`fmadd.s`) and multiply-subtract (`fmsub.s`) operations. All floating-point operations operate on floating-point registers, and are mapped to execute on floating-point units FU0 and FU1.

A.1.5 Control Operations

Table A.5 shows the set of control operations used in the UTDSP. In addition to a set of operations that change control flow, control operations include low-overhead looping operations that cause a single operation or a block of operations to be repeatedly executed for a specified number of iterations. Control operations also include operations for moving data between different register files while converting them to the proper representation. Finally, the set of control operations includes a `trap` operation that the instruction set simulator uses to perform file I/O, and to implement transcendental C functions. Table A.6 shows the different trap operations and transcendental functions.

Operation	Syntax	Functional Description	Encoding Format
Load address register	`ld.a (ai),aj`	`aj = D[ai]`	Type I
Load integer register	`ld.d (ai),dj`	`dj = D[ai]`	Type I
Load floating-point register	`ld.s (ai),fj`	`fj = D[ai]`	Type I
Store address register	`st.a (ai),aj`	`D[ai] = aj`	Type I
Store integer register	`st.d (ai),dj`	`D[ai] = dj`	Type I
Store floating-point register	`st.s (ai),fj`	`D[ai] = fj`	Type I

Table A.1: Memory Operations

Operation	Syntax	Functional Description	Encoding Format
Address decrement	`dec ai,aj,ak`	`ak = ai - aj`	Type I
Modulo address decrement	`decmod ai,aj,ak,al`	`al = (ai - aj) mod ak`	Type I
Bit-reversed address decrement	`decfft ai,aj,ak`	`ak = ai - aj` `(bit-reversed)`	Type I
Address increment	`inc ai,aj,ak`	`ak = ai + aj`	Type I
Modulo address increment	`incmod ai,aj,ak,al`	`al = (ai + aj) mod ak`	Type I
Bit-reversed address increment	`incfft ai,aj,ak`	`ak = ai + aj` `(bit-reversed)`	Type I
Bit-wise AND	`and.a ai,aj,ak`	`ak = ai & aj`	Type I
Arithmetic shift left	`asl.a ai,aj,ak`	`ak = ai << aj` `(arithmetic)`	Type I
Arithmetic shift right	`asr.a ai,aj,ak`	`ak = ai >> aj` `(arithmetic)`	Type I
Bit-wise inclusive OR	`ior.a ai,aj,ak`	`ak = ai | aj`	Type I
Logical shift left	`lsl.a ai,aj,ak`	`ak = ai << aj` `(logical)`	Type I
Logical shift right	`lsr.a ai,aj,ak`	`ak = ai >> aj` `(logical)`	Type I
Modulo operator	`mod.a ai,aj,ak`	`ak = ai % aj`	Type I
Move register	`mov.a ai,ak`	`ak = ai`	Type I
Move immediate	`movi.a #X,ak`	`ak = #X`	Type II
Bit-wise NOT	`not.a ai,ak`	`ak = ~ai`	Type I
Bit-wise exclusive OR	`xor.a ai,aj,ak`	`ak = ai ^ aj`	Type I
Set equal	`seq.a ai,aj,ak`	`ak = (ai == aj)`	Type I
Set not equal	`sne.a ai,aj,ak`	`ak = (ai != aj)`	Type I
Set greater than	`sgt.a ai,aj,ak`	`ak = (ai > aj)`	Type I
Set less than	`slt.a ai,aj,ak`	`ak = (ai < aj)`	Type I

Table A.2: Addressing Operations

<table>
<tr><th>Operation</th><th>Syntax</th><th>Functional Description</th><th>Encoding Format</th></tr>
<tr><td>Absolute value</td><td>abs.d di,dk</td><td>dk = |di|</td><td>Type I</td></tr>
<tr><td>Add</td><td>add.d di,dj,dk</td><td>dk = di + dj</td><td>Type I</td></tr>
<tr><td>Bit-wise AND</td><td>and.d di,dj,dk</td><td>dk = di & dj</td><td>Type I</td></tr>
<tr><td>Arithmetic shift left</td><td>asl.d di,dj,dk</td><td>dk = di << dj (arithmetic)</td><td>Type I</td></tr>
<tr><td>Arithmetic shift right</td><td>asr.d di,dj,dk</td><td>dk = di >> dj (arithmetic)</td><td>Type I</td></tr>
<tr><td>Bit-wise inclusive OR</td><td>ior.d di,dj,dk</td><td>dk = di | dj</td><td>Type I</td></tr>
<tr><td>Logical shift left</td><td>lsl.d di,dj,dk</td><td>dk = di << dj (logical)</td><td>Type I</td></tr>
<tr><td>Logical shift right</td><td>lsr.d di,dj,dk</td><td>dk = di >> dj (logical)</td><td>Type I</td></tr>
<tr><td>Modulo operator</td><td>mod.d di,dj,dk</td><td>dk = di % dj</td><td>Type I</td></tr>
<tr><td>Move register</td><td>mov.d di,dk</td><td>dk = di</td><td>Type I</td></tr>
<tr><td>Move immediate</td><td>movi.d #X,dk</td><td>dk = #X</td><td>Type II</td></tr>
<tr><td>Bit-wise NOT</td><td>not.d di,dk</td><td>dk = ~di</td><td>Type I</td></tr>
<tr><td>Subtract</td><td>sub.d di,dj,dk</td><td>dk = di - dj</td><td>Type I</td></tr>
<tr><td>Bit-wise exclusive OR</td><td>xor.d di,dj,dk</td><td>dk = di ^ dj</td><td>Type I</td></tr>
<tr><td>Set equal</td><td>seq.d di,dj,dk</td><td>dk = (di == dj)</td><td>Type I</td></tr>
<tr><td>Set not equal</td><td>sne.d di,dj,dk</td><td>dk = (di != dj)</td><td>Type I</td></tr>
<tr><td>Set greater than</td><td>sgt.d di,dj,dk</td><td>dk = (di > dj)</td><td>Type I</td></tr>
<tr><td>Set less than</td><td>slt.d di,dj,dk</td><td>dk = (di < dj)</td><td>Type I</td></tr>
<tr><td>Multiply-accumulate</td><td>madd.d di,dj,dk,dl</td><td>dl = dk + (di * dj)</td><td>Type I</td></tr>
<tr><td>Multiply-subtract</td><td>msub.d di,dj,dk,dl</td><td>dl = dk - (di * dj)</td><td>Type I</td></tr>
<tr><td>Multiply</td><td>mult.d di,dj,dk</td><td>dk = di * dj</td><td>Type I</td></tr>
<tr><td>Divide</td><td>div.d di,dj,dk</td><td>dk = di / dj</td><td>Type I</td></tr>
</table>

Table A.3: Integer Operations

Operation	Syntax	Functional Description	Encoding Format
Add	fadd.s fi,fj,fk	fk = fi + fj	Type I
Move register	fmov.s fi,fk	fk = fi	Type I
Subtract	fsub.s fi,fj,fk	fk = fi - fj	Type I
Set equal	fseq.s fi,fj,fk	fk = (fi == fj)	Type I
Set not equal	fsne.s fi,fj,fk	fk = (fi != fj)	Type I
Set greater than	fsgt.s fi,fj,fk	fk = (fi > fj)	Type I
Set less than	fslt.s fi,fj,fk	fk = (fi < fj)	Type I
Multiply-accumulate	fmadd.s fi,fj,fk,fl	fl = fk + (fi * fj)	Type I

Table A.4: Floating-Point Operations

Operation	Syntax	Functional Description	Encoding Format
Multiply-subtract	`fmsub.s fi,fj,fk,fl`	`fl = fk - (fi * fj)`	Type I
Divide	`fdiv.s fi,fj,fk`	`fk = fi / fj`	Type I
Multiply	`fmult.s fi,fj,fk`	`fk = fi * fj`	Type I

Table A.4: Floating-Point Operations

Operation	Syntax	Functional Description	Encoding Format
Address to integer	`mova2d ai,dk`	`dk = ai`	Type I
Integer to address	`movd2a di,ak`	`ak = di`	Type I
Floating-point to integer	`movs2d fi,dk`	`dk = fi`	Type I
Integer to floating-point	`movd2s di,fk`	`fk = di`	Type I
Instruction block repeat	`do #X, label`	`Repeat instruction block #X times`	Type III
Instruction block repeat	`do.a ai,label`	`Repeat instruction block (ai) times`	Type II
Instruction block repeat	`do.d di,label`	`Repeat instruction block (di) times`	Type II
Single-instruction repeat	`rep #X`	`Repeat following instruction #X times`	Type II
Branch if address register equal to zero	`beqz.a ai,label`	`PC = label if (ai == 0)`	Type II
Branch if integer register equal to zero	`beqz.d di,label`	`PC = label if (di == 0)`	Type II
Branch if floating-point register equal to zero	`beqz.s fi,label`	`PC = label if (fi == 0.0)`	Type II
Branch if address register not equal to zero	`bnez.a ai,label`	`PC = label if (ai != 0)`	Type II
Branch if integer register not equal to zero	`bnez.d di,label`	`PC = label if (di != 0)`	Type II
Branch if floating-point register not equal to zero	`bnez.s fi,label`	`PC = label if (fi != 0)`	Type II
Jump indirect	`jmp.a (ai)`	`PC = (ai)`	Type I
Jump direct	`jmp label`	`PC = label`	Type II
Jump subroutine	`jsr label`	`Save PC and context on the stack; PC = label`	Type II
Return from interrupt	`rti`	`Restore PC and context from the stack`	Type I
Return from subroutine	`rts`	`restore PC and context from the stack`	Type I

Table A.5: Control Operations

Operation	Syntax	Functional Description	Encoding Format
Trap	`trap #X`	`Implement file I/O or C transcendental functions`	Type II

Table A.5: Control Operations

Trap	Function Implemented	Trap	Function Implemented
1 – 4	Not used	15	floor()
5	Input to data-memory bank X	16	log()
6	Output from data-memory bank X	17	log10()
7	abs()	18	sin()
8	fabs()	19	sqrt()
9	fmod()	20	pow()
10	asin()	21	tan()
11	acos()	22	atan()
12	ceil()	23	exit()
13	cos()	50	Input to data-memory bank Y
14	exp()	60	Output from data-memory bank Y

Table A.6: Transcendental Functions

A.2 Encoding UTDSP Operations

This section describes how individual UTDSP operations are encoded, regardless of whether they are part of a multi-op instruction stored in decoder memory, or are uni-op instructions stored in instruction memory. One of the factors affecting the way operations are encoded is the length of the instruction word used. While Chapter 6 discussed the impact of the instruction word length on performance and cost, this section shows how operations are encoded using 32-bit, 24-bit, and 16-bit instruction words.

As with RISC instructions, the main objective of the encoding is to simplify the decoding logic associated with the different functional units. For this reason, only three operation formats, Types I, II, and III, are used. Each format uses fixed operation and operand fields.

The Type I format is used to encode register-register operations. Since most UTDSP operations are register-based, this format is the most commonly used. The Type II format is used to encode operations that use a single, immediate operand, such as `movi.a`, `bnez.d`, and `rep`, and that may also use a single register operand either as a source or a destination. Finally, the Type III format is used for operations that use two immediate operands. The only UTDSP operation that uses this format is the `do` operation. Its two

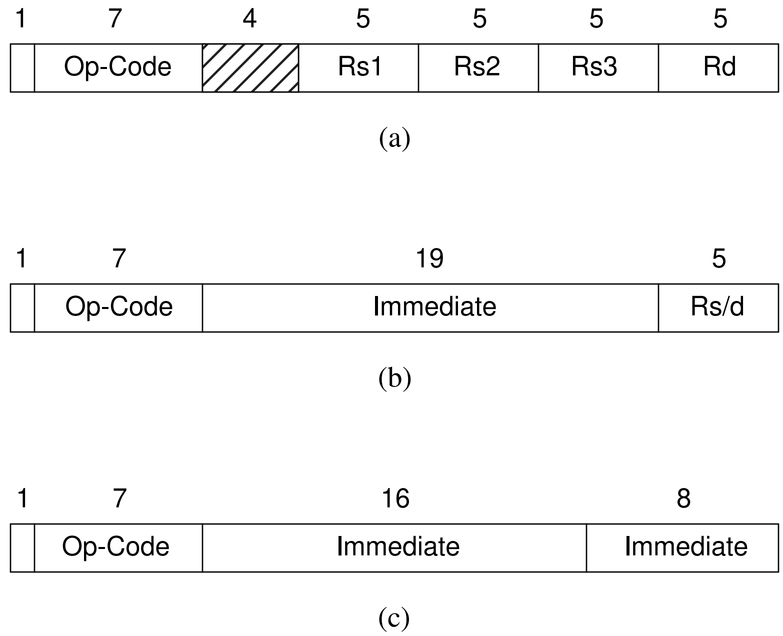


Figure A.1: Operation formats used with 32-bit instruction words.
(a) Type I format. (b) Type II format. (c) Type III format.

immediate operands are the PC-relative offset into the address of the last instruction in a repeated block, and the block iteration counter.

A.2.1 32-Bit Operation Formats

Figure A.1 shows the operation formats used with 32-bit instruction words. All three formats use a seven-bit field to encode the op-code. Furthermore, the most-significant bit in each format is used to identify the operation as a uni-op instruction when it is fetched from instruction memory. This helps the instruction decoding circuitry dispatch the operation to the functional units directly.

While the majority of register-register operations use two source operands, some operations, such as `madd.d` and `incmod`, use three. The Type I format therefore uses four, five-bit fields to specify up to three source registers and a destination register. Each five-bit register field can specify 32 registers. The Type II format uses a 19-bit immediate field and a five-bit register field. Finally, the Type III format uses two immediate fields: a 16-bit immediate field to store the PC-relative offset into the last instruction in the body of a loop; and an eight-bit immediate field to store the block iteration count. The lengths of these fields can be changed to better suite the target applications, and are only used here as an example.

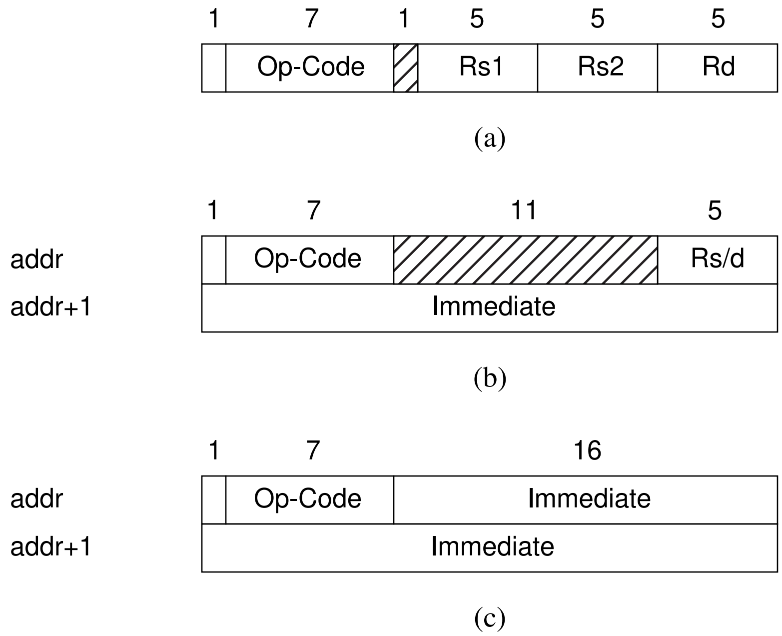


Figure A.2: Operation formats used with 24-bit instruction words.
(a) Type I format. (b) Type II format. (c) Type III format

A.2.2 24-Bit Operation Formats

Figure A.2 shows the operation formats used with 24-bit instruction words. Like the formats used with 32-bit instruction words, the three formats use a seven-bit field to encode the op-code. They also use the most-significant bit to identify an operation as a uni-op instruction when it is fetched from instruction memory.

The Type I format uses three, five-bit register fields that can specify up to two source registers and a destination register. For operations that require three source operands, the presence of only two fields for encoding source registers necessitates that the third source register be implicitly specified in the destination register field. In other words, the third source register and the destination register should be identical. Since the operations that require three source operands involve sum-of-product and modulo update operations, they can be expressed as accumulator-based operations, and are therefore not affected by the more restrictive encoding. The five-bit register fields allow 32 registers to be specified as sources or destinations.

Unlike the one used with 32-bit instruction words, the Type II format uses two instruction words. The first word — the one stored at the lower memory address — is used to specify the op-code and, depending on the operation, either a source register or a destination register. The register is specified using a five-bit field. The second word, on the other hand, is used to specify a 16-bit immediate operand. Since operations

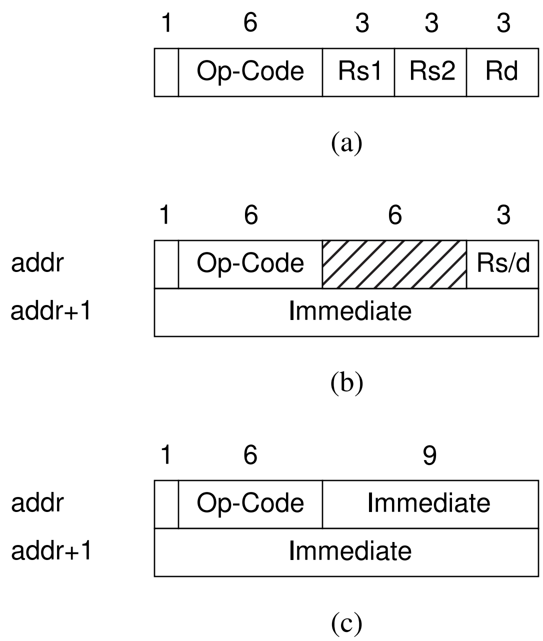


Figure A.3: Operation formats used with 16-bit instruction words.
(a) Type I format. (b) Type II format. (c) Type III format.

that use the Type II format occupy two words in memory, the decoding logic on the functional units should be modified to enable the forwarding of the immediate operand to the appropriate pipeline stage. Section 6.3 in Chapter 6 describes this in more detail, and discusses the performance and cost implications of using two-word operations.

Finally, the Type III format also uses two instruction words. The first word is used to specify the op-code and a 16-bit iteration count. The second word, on the other hand, is used to specify a 24-bit offset to the last instruction in the body of the loop. Since the Type III format also occupies two words in memory, the decoding logic on the appropriate functional units should also be modified to ensure that the 24-bit offset is forwarded to the appropriate pipeline stage. Section 6.3 also describes this in more detail.

A.2.3 16-Bit Operation Formats

Figure A.3 shows the operation formats used with 16-bit instruction words. Unlike the formats used with 32-bit and 24-bit instruction words, all three formats use a six-bit field to encode the op-code. Although this limits the total number of operations that can be specified, a rich mix of operations can still be maintained through a judicious selection of operations. For all three formats, the most-significant bit is still used to identify an operation as a uni-op instruction when it is fetched from instruction memory.

The Type I format, like that used with 24-bit instruction words, uses three fields to encode up to two source registers and a destination register. However, unlike the Type I format used with 24-bit instruction words, each register field consists of only three bits. This limits the number of registers that can be specified as source or destination operands to only eight. As the results from Chapter 4 demonstrated, using a smaller number of registers decreases performance and increases code size due to the additional memory access operations needed to move data between memory and the small set of registers. Chapter 6 describes the impact of using 16-bit instruction words on performance and cost in more detail.

The Type II and Type III operation formats, like those used with 24-bit instruction words, also use two instruction words to encode an operation. For the Type II format, the first word is used to specify the op-code and a source or destination register. The register is specified using a three-bit field. The second word, on the other hand, is used to store a 16-bit immediate operand. For the Type III format, the first word is used to specify the op-code and a nine-bit iteration count. The second word is used to store a 16-bit offset to the last instruction in the body of a loop. Since both formats use two-word operation formats, the decoding logic must also be modified to ensure that the immediate operands are forwarded to the appropriate pipeline stage. Section 6.3 in Chapter 6 describes this in more detail.

Appendix B

Deriving the Bit-Area Estimates of the Datapath Area Model

The model used to estimate the area of the UTDSP datapath is based on estimating the area of its different datapath components, and adding them to obtain an estimate of the total datapath area. In turn, the area of a datapath component is obtained by multiplying a technology-independent bit-area estimate by the data width of the datapath, and scaling it by the appropriate line size. The technology-independent bit-area estimate of a datapath component is an approximate value of the area occupied by a single bit of that component, independent of the line size used. This appendix explains how the bit-area estimates of the different datapath components in the UTDSP datapath area model were derived. Section 2.8 in Chapter 2 describes the datapath area model in more detail.

B.1 Deriving Technology-Independent Bit-Area Estimates

The bit-area estimates of different datapath components used in the UTDSP datapath area model were derived from the die photographs of two research and two commercial microprocessors. These included the MIPS-X [30], SPUR [31], MIPS R5000 [32], and MIPS R10000 [33]. The MIPS-X and SPUR were mainly used because of the familiarity of University of Toronto researchers with their design and implementation details. On the other hand, the MIPS R5000 and R10000 were used because die photographs, with overlays showing different datapath components, and information about their implementation were readily available. Table B.1 shows the technologies used in the design of these microprocessors, the dimensions of their dies, and their data widths.

The first step in deriving the bit-area estimate of a datapath component from a die photograph is to determine the overall scale. The scale can easily by found by dividing the length of any of the sides of the die on the actual chip, by the corresponding length of that side on the photograph.

Microprocessor	Technology	Line Size	Metal Layers	Die Size	Data Width
MIPS-X	CMOS	2.00 μm	2	8.0 mm × 8.0 mm	32 bits
SPUR	CMOS	2.00 μm	2	10.0 mm × 10.0 mm	64 bits
MIPS R5000	CMOS	0.35 μm	3	8.8 mm × 9.5 mm	64 bits
MIPS R10000	CMOS	0.35 μm	4	16.6 mm × 17.9 mm	64 bits

Table B.1: Microprocessors used and their characteristics

Once the scale has been found, the area of the datapath component of interest can be measured. Although some experience is usually required to identify datapath components on a die photograph, most recent die photographs simplify the task by outlining different components, such as ALUs, register files, and memory banks. After measuring the length and the width of a component on the photograph, the derived scale can be used to determine the actual dimensions of the component, and to calculate its actual area.

Once the area of the datapath component has been measured, its bit-area estimate can easily be calculated by dividing the area of the component by the data width used. For example, if a 32-bit ALU has an area of 3.20×10^6 μm^2, its bit-area estimate will be 10^5 μm^2. Using bit-area estimates is a very useful way of estimating the area of a component for different data widths. For example, to estimate the area of a similar, 24-bit ALU, the bit-area estimate of the ALU, 10^5 μm^2, is multiplied by 24 to give an area of 2.40×10^6 μm^2. For some datapath components, such as multipliers and registers, calculating the bit-area estimates involves more than simply dividing the area of the datapath by the data width. For example, the bit-area estimate for a multiplier is derived by dividing the area of the multiplier by the *square* of the data width. On the other hand, calculating the bit-area estimate for a register must take into account the number of its read and write ports.

Although bit-area estimates are very useful, they are still tightly coupled to the specific technology and line size used. To express bit-area estimates in technology-independent terms, the dimensions and areas of datapath components can be measured in terms of the technology-independent factor, f. By definition, f is equal to the line size expressed without dimensions. Thus, any dimension can be expressed in terms of f simply by multiplying it by f and dividing the result by the dimensionless line size. Similarly, any area can be expressed in terms of f by multiplying it by f^2 and dividing the result by the square of the dimensionless line size. For example, if a datapath component has dimensions 100 μm × 40 μm, and is implemented in 2 μm technology, $f = 2$, and its technology-independent dimensions are $(50 \times f)$ μm × $(20 \times f)$ μm. Similarly, the technology-independent expression of its area is $1000 \times f^2$ μm^2. Using technology-independent bit-area estimates makes it easy to approximate the area for different line sizes. For example, if the line size

Datapath Components	Microprocessor	Area (μm^2)	Bit-Area Estimates (μm^2/bit)	Technology-Independent Bit-Area Estimates (μm^2/bit)
Integer ALU	MIPS-X	3.00×10^6	9.2×10^4	$2.30 \times 10^4 \times f^2$
Floating-Point ALU	SPUR	22.88×10^6	35.76×10^4	$8.94 \times 10^4 \times f^2$
Floating-Point Multiplier	MIPS R5000	2.69×10^6	6.57×10^2	$5.36 \times 10^3 \times f^2$
Floating-Point Multiplier	MIPS R10000	5.60×10^6	1.37×10^3	$1.12 \times 10^4 \times f^2$
Floating-Point Divide/ Square-Root Circuit	MIPS R5000	1.53×10^6	2.39×10^4	$19.53 \times 10^4 \times f^2$
Memory Cell	MIPS-X	4.67×10^4	1.46×10^3	$3.65 \times 10^2 \times f^2$
Register Cell (2R, 1W)	MIPS-X	9.28×10^4	2.90×10^3	$7.25 \times 10^2 \times f^2$
Single-Ported Register Cell	MIPS-X	9.28×10^4	2.90×10^3	$7.25 \times 10^2 \times f^2$
Register Cell (2R, 2W)	SPUR	1.25×10^4	1.96×10^3	$4.89 \times 10^2 \times f^2$
Single-Ported Register Cell	SPUR	8.33×10^3	1.30×10^2	$3.25 \times 10^1 \times f^2$

Table B.2: Area, bit-area estimates, and technology-independent bit-area estimates of different datapath components.

in the above example is reduced from 2 µm to 1 µm, it is easy to see that the area of the component will be reduced from 4000 μm^2 to 1000 μm^2.

B.2 Bit-Area Estimates of Key Datapath Components

Table B.2 shows the areas, bit-area estimates, and technology-independent, bit-area estimates of key datapath components measured from the die photographs of the four microprocessors. The first column shows the different datapath components whose areas were measured. These include an integer ALU, a floating-point ALU, a floating-point multiplier, a floating-point divide/square-root circuit, a memory cell, and a register cell. For the floating-point multiplier and the register cell, two measurements were made from different microprocessors. Since the area of a register cell depends on the number of ports used to access the cell, the corresponding areas of single-ported register cells were also derived. Here, it was assumed that each additional write port[1] increases the area of the register cell by 50%. Since the MIPS-X uses a register file with two read ports and a single write port, the area of each register cell is equivalent to

1. Since the circuitry used to write to a register cell can also be used to read two different register cells, each additional write port effectively increases the number of read ports by two. Providing *W* write ports and *R* read ports is therefore equivalent to providing $max\,(W, \lceil R/2 \rceil)$ write ports.

UTDSP Datapath Components	Technology-Independent Bit-Area Estimates (μm^2/bit)
Integer ALU	$2.30 \times 10^4 \times f^2$
Integer Multiplier/Divider	$2.03 \times 10^5 \times f^2$
Floating-Point Adder	$8.94 \times 10^4 \times f^2$
Floating-Point Multiplier	$8.28 \times 10^3 \times f^2$
Floating-Point Divide/Square-Root Circuit	$1.95 \times 10^5 \times f^2$
Single-Ported Register Cell	$3.96 \times 10^2 \times f^2$
Memory Cell	$3.65 \times 10^2 \times f^2$

Table B.3: Technology-independent bit-area estimates of UTDSP datapath components.

the area of a single-ported register cell. On the other hand, since the SPUR uses a register file with two read ports and two write ports, the corresponding single-ported register cell area was obtained by dividing the register cell area by 1.5 ($1 + 1 \times 0.5$).

The second column in Table B.2 shows the microprocessors from which the datapath components were taken, while the third column shows the areas of the different datapaths, expressed in μm^2. The fourth column shows the bit-area estimates for each datapath component, expressed in μm^2/bit. These were obtained simply by dividing the area, shown in the third column, by the data width of the microprocessor. For the floating-point multipliers, the bit-area estimates were obtained by dividing the area by the square of the data width. The data width of each microprocessor is shown in Table B.1. Finally, the fifth column shows the technology-independent, bit-area estimates of the different datapath components, expressed in μm^2/bit. These were obtained by multiplying the bit-area estimates, shown in the fourth column, by f^2 and dividing the results by the square of the dimensionless microprocessor line size. The line size of each of the microprocessors is also shown in Table B.1.

The UTDSP datapath area model was developed using the technology-independent, bit-area estimates of the different datapath components in Table B.2. Table B.3, which also appeared as Table 2.3 in Chapter 2, shows the bit-area estimates of the different datapath components used in the UTDSP datapath area model. For most of these datapath components, such as the integer ALU, the floating-point ALU, the floating-point divide/square-root circuit, and the memory cell, the model uses the same technology-independent bit-area estimates shown in Table B.2. However, for the other components, the bit-area estimates used in the model were *derived* from the ones in Table B.2. For example, the bit-area estimate of the floating-point multiplier was taken as the average of the two estimates in Table B.2. Furthermore, the bit-area estimate of the integer multiplier/divider was taken as the sum of the bit-area estimates of the floating-point multiplier and the floating-point divide/square-root circuit. Finally, the bit-area estimate of the single-ported register cell was taken as the average of the single-ported register cell bit-area estimates of the MIPS-X and the SPUR microprocessors.

B.3 Limitations of Bit-Area Estimates

While the technology-independent, bit-area estimates provide a convenient way of estimating the area of individual datapath components, and the overall area of a datapath, they are not without their limitations. For example, the bit-area estimates are only expressed in terms of the technology-independent constant, k, which, in turn, depends on the line size, λ. Although the line size is the major factor affecting the area of a component, it is not the only one. Other factors, such as the number of metal layers used, for example, also affect the area of a component — generally, the greater the number of metal layers, the easier it is to connect transistors, and the more compact the area of a circuit can be. However, since it is difficult to model the impact of all these factors on area, and since the line size is the major factor affecting area, the technology-independent, bit-area estimates provide good, first-order approximations.

Since the bit-area estimates are first-order approximations, the resulting component and datapath area estimates cannot be guaranteed to be accurate. However, since they provide a means of quantifying the differences in area, and hence cost, of different architectural configurations, they are a useful design tool that can help designers perform the necessary performance and cost trade-offs required to meet the specifications of a target application.

Bibliography

[1] Pierre G. Paulin, et al., “Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends,” *Proceedings of the IEEE*, Vol. 85, No. 3, pp. 419–435, IEEE, March, 1997.

[2] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee, “Application-Driven Design of DSP Architectures and Compilers,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. II-437–440, IEEE, 1994.

[3] Sanjay Pujare, Corinna G. Lee, and Paul Chow, “A Performance Evaluation of Machine-Independent Compiler Optimizations,” *Proceedings of the Sixth International Conference on Signal Processing Applications and Technology*, pp. I-860–865, DSP Associates, October, 1995.

[4] Monica S. Lam, “Software Pipelining: An Effective Scheduling Technique for VLIW Machines,” *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 318–328, ACM, June, 1988.

[5] Joseph Fisher, “Very Long Instruction Word Architectures and the ELI-512,” *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 140–150, IEEE, 1983.

[6] John R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, 1985.

[7] Brian Case, “Philips Hopes to Displace DSPs with VLIW,” *Microprocessor Report*, Vol. 8, No. 16, pp. 12–15, MicroDesign Resources, December 5, 1994.

[8] Selliah Rathnam and Gert Slavenburg, “Processing the New World of Interactive Media: The Trimedia VLIW CPU Architecture”, *IEEE Signal Processing Magazine*, Vol. 15, No. 2, pp. 108–117, IEEE, March, 1998.

[9] Dave Epstein, “Chromatic Raises the Multimedia Bar,” *Microprocessor Report*, Vol. 9, No. 14, pp. 23–27, MicroDesign Resources, October 23, 1995.

[10] Dave Epstein, “IBM Extends DSP Performance with Mfast,” *Microprocessor Report*, Vol. 9, No. 16, pp. 1, 6–8, MicroDesign Resources, December 4, 1995.

[11] Jim Turley and Harri Hakkarainen, “TI’s New ‘C6x DSP Screams at 1,600 MIPS,” *Microprocessor Report*, Vol. 11, No. 2, pp. 14–17, MicroDesign Resources, February 17, 1997.

[12] Nat Seshan, “High VelociTI Processing,” *IEEE Signal Processing Magazine*, Vol. 15, No. 2, pp. 86–101, 117, IEEE, March, 1998.

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[14] Richard M. Stallman, *Using and Porting GNU C*, Free Software Foundation, Inc., 1990.

[15] Vijaya K. Singh, *An Optimizing C Compiler for a General Purpose DSP Architecture*, M.A.Sc. Thesis, University of Toronto, 1992.

[16] Mazen A. R. Saghir, *Architectural and Compiler Support for DSP Applications*, M.A.Sc. Thesis, University of Toronto, 1993.

[17] Mark G. Stoodley, *Scheduling Loops with Conditional Statements*, M.A.Sc. Thesis, University of Toronto, 1995.

[18] Robert P. Wilson, et al., “SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers,” http://suif.stanford.edu/suif/suif1/suif-overview/suif.html, 1994.

[19] Sanjay M. Pujare, *Machine-Independent Compiler Optimizations for the U of T DSP Architecture*, M.Eng. Thesis, University of Toronto, 1995.

[20] Stan Liao, et al., “Code Generation and Optimization Techniques for Embedded Digital Signal Processors,” *Proceedings of the First SUIF Compiler Workshop*, pp. 36–39, Stanford University, 1996.

[21] Mike Tien-Chien Lee, "SUIF-Based Retargetable DSP Code Generation for Telecommunication Application," *Proceedings of the First SUIF Compiler Workshop*, pp. 40–45, Stanford University, 1996.

[22] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett, "Local Microcode Compaction Techniques," *ACM Computing Surveys*, Vol. 12, No. 3, pp. 216–294, ACM, September, 1980.

[23] Motorola, *DSP56002 Digital Signal Processor User's Manual*, http://www.mot.com, 1990.

[24] Motorola, *DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual*, http://www.mot.com, 1989.

[25] Texas Instruments, *TMS320C5x User's Guide*, http://www.ti.com, 1993.

[26] Texas Instruments, *TMS320C3x User's Guide*, http://www.ti.com, 1989.

[27] Analog Devices, *ADSP-2100 Family User's Manual*, Third Edition, http://www.analog.com, 1995.

[28] Analog Devices, *ADSP-21020 User's Manual,* Second Edition, http://www.analog.com, 1995.

[29] Joseph A. Fisher, Paolo Fraboschi, and Giuseppe Desoli, *Custom-Fit Processors: Letting Applications Define Architectures*, Technical Report HPL-96-144, HP Laboratories, 1996.

[30] Paul Chow, *The MIPS-X RISC Microprocessor*, Kluwer Academic Publishers, 1989.

[31] Mark Hill, et al., "Design Decisions in SPUR," *Computer*, Vol. 19, No. 11, IEEE, November, 1986.

[32] MIPS Technologies, Inc., *MIPS R5000 Microprocessor*, http://www.sgi.com/MIPS/, January, 1996.

[33] MIPS Technologies, Inc., *MIPS RISC Technology R10000 Microprocessor Technical Brief*, http://www.sgi.com/MIPS/, October, 1994.

[34] Vojin Zivojnovic, "Compilers for Digital Signal Processors: The Hard Way from Marketing- to Production-Tool," *DSP & Multimedia Technology Magazine*, Vol. 4, No. 5, pp. 27–45 July/August, 1995.

[35] Ashok Sudarsanam and Sharad Malik, "Memory Bank and Register Allocation in Software Synthesis for ASIPs," *Proceedings of the International Conference on Computer-Aided Design*, pp. 388–392, IEEE/ACM, 1995.

[36] Jose Luis Pino, Soonhoi Ha, Edward A. Lee, and Joseph T. Buck, "Software Synthesis for DSP Using Ptolemy," *Journal of VLSI Signal Processing*, Vol. 9, No. 1–2, pp. 7–21, January, 1995.

[37] P. G. Lowney, et al., "The Multiflow Trace Scheduling Compiler," *Journal of Supercomputing*, Vol. 7, Issue 1-2, pp. 51–141, May, 1993.

[38] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee, "Towards Better DSP Architectures and Compilers," *Proceedings of the Fifth International Conference on Signal Processing Applications and Technology*, pp. I-658–664, DSP Associates, October, 1994.

[39] Alan Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1985.

[40] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[41] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[42] R. M. Keller, "Look-Ahead Processors," *ACM Computing Surveys*, Vol. 7, pp. 66–72, ACM, 1975.

[43] Special issue on multimedia extensions for general-purpose processors, *IEEE Micro*, Vol. 16, No. 4, IEEE, August, 1996.

[44] Gerald Cheong and Monica Lam, "An Optimizer for Multimedia Instruction Sets: A Preliminary Report," *Proceedings of the Second SUIF Compiler Workshop*, Stanford University, 1997.

[45] Derek DeVries, *A Vectorizing SUIF Compiler*, M.A.Sc. Thesis, University of Toronto, 1997.

[46] Paul Hellyar, *Automatic Compilation for Short Vector Extensions*, M.A.Sc. Thesis, University of Toronto, in progress.

[47] Texas Instruments, *TMS320C80 (MVP) Parallel Processor User's Guide*, http://www.ti.com, 1995.

[48] Markus Levy, "C Compilers for DSPs Flex Their Muscles," *EDN Magazine*, pp. 93–107, June 5, 1997.

[49] Subrata Dasgupta, "The Organization of Microprogram Stores," *ACM Computing Surveys*, Vol. 11, No. 1, pp. 39–65, ACM, March, 1979.

[50] E. W. Reigel, U. Faber, and D. A. Fisher, “The Interpreter — A Microprogrammable Building Block System,” *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 40, pp. 705–723, 1972.

[51] Robert F. Rosin, Gideon Frieder, and Richard H. Eckhouse, Jr., “An Environment for Research in Microprogramming and Emulation,” *Communications of the ACM*, Vol. 15, No. 8, pp. 748–760, ACM, 1972.

[52] R. P. Colwell, et al., “A VLIW Architecture for a Trace Scheduling Compiler,” *IEEE Transactions on Computers*, Vol. 37, No. 8, pp. 967–979, IEEE, August, 1988.

[53] B. R. Rau, et al., “The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs,” *Computer*, Vol. 22, No. 1, pp. 12–35, IEEE, January, 1989.

[54] D. R. Ditzel, H. R. McLellan, and A. D. Berenbaum, “The Hardware Architecture of the CRISP Microprocessor,” *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 309–319, IEEE, 1987.

[55] Richard Crisp, “Direct Rambus Technology: The New Main Memory Standard,” *IEEE Micro*, Vol. 17, No. 6, pp. 18–28, November/December, 1997.

[56] Steve Purcell, “The Impact of Mpact 2,” *IEEE Signal Processing Magazine*, Vol. 15, No. 2, pp. 102–107, IEEE, March, 1998.

[57] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, “Approximation Algorithms for Bin-Packing — An Updated Survey,” *Algorithm Design for Computer System Design*, Springer Verlag, 1984.

[58] J. Gary Augustson and Jack Minker, “An Analysis of Some Graph Theoretical Cluster Techniques,” *Journal of the Association for Computing Machinery*, Vol. 17, No. 4, pp. 571–588, ACM, October, 1970.

[59] Paul Fortier, Louis E. Pouliot, Louis Bélanger, and Jean-Sébastien Dion, *Report on DSP Design Tools and Methodologies*, Report IC 95-12, Canadian Microelectronics Corporation, 1995.