Published in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing,* pp. II-437 to II-440, April 1994.

APPLICATION-DRIVEN DESIGN OF DSP ARCHITECTURES AND COMPILERS

Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee

Department of Electrical and Computer Engineering, University of Toronto
10 King's College Road, Toronto, Ontario M5S 1A4
CANADA

ABSTRACT

Current DSP architectures are designed to enhance the execution of computationally-intensive, kernel-like loops. Their peculiar architectural features are often difficult for high-level language compilers to exploit. Moreover, their tightly-encoded instruction sets usually restrict the exploitation of instruction-level parallelism beyond a few instances. The quality of compiler-generated code is therefore poor when compared to hand-coded assembly language. In this paper we argue for an application-driven approach to designing flexible DSP architectures and effective compilers. We show that the run-time behavior and architectural characteristics of DSP kernels are different from those of DSP applications. We also show that when given a sufficiently flexible target architecture, a compiler is capable of effectively exploiting instances of instruction-level parallelism and DSP-specific architectural features. Finally, we show that a suitable DSP architecture is one that provides the functionality to support digital signal processing requirements, and the flexibility that enables a compiler to generate efficient code.

1. INTRODUCTION

The current architectures of general-purpose, programmable digital signal processors (DSPs) are tuned to efficiently execute the computationally-intensive loops that typically characterize digital signal processing algorithms. To fully exploit these architectural features, and to enhance overall performance and code size, DSPs have traditionally been programmed using assembly language. As digital signal processing applications become more sophisticated, however, coding, debugging, maintaining and porting large assembly language programs become less desirable. Optimizing high-level language (HLL) compilers are therefore seen as a better alternative for programming DSPs. However, the quality of the code that is generated by current compilers is often poor when compared to hand-coded assembly language programs. This is due to the inability of compilers to utilize the peculiar architectural features of DSPs. Furthermore, compilers are limited in their exploitation of instruction-level parallelism by the tightly-encoded instruction sets of current DSPs [1][2][3]. Hence, there is a need for an architecture that provides the *functionality* needed to satisfy digital signal processing requirements, and the *flexibility* that enables HLL compilers to effectively utilize the architecture and exploit instruction-level parallelism.

Contemporary architectural design stresses the simultaneous development of an architecture and its compiler, based on cost/performance trade-off studies that rely on run-time data gathered from benchmark programs [4]. These programs should represent typical, envisioned applications and work loads. Current DSP architectures, however, are designed to optimize the execution of short, computationally-intensive loops [3]. These loops constitute kernels that are often used to benchmark the performance of DSP processors [1]. Using these kernels as a basis for designing new architectures may therefore lead to specious conclusions.

In this paper, we investigate the differences in the run-time behavior and architectural characteristics of DSP kernels and applications. We also study the ability of a HLL compiler to expose and exploit instances of instruction-level parallelism, and to make use of a flexible DSP target architecture.

2. EXPERIMENTAL METHODOLOGY

We wrote several DSP kernels and applications in the C programming language [5]. These were used as benchmarks to study the availability of instruction-level parallelism, and the impact of different compiler optimizations on execution performance.

Twelve kernels, based on six common DSP algorithms, were implemented. For each algorithm, two kernels of different problem sizes were used. The DSP algorithms that were used are:

- radix-2, in-place, decimation-in-time fast Fourier transform;
- finite impulse response (FIR) filter;
- infinite impulse response (IIR) biquad filter;
- normalized lattice filter;
- least mean squared adaptive FIR filter; and
- matrix multiplication.

Six typical DSP applications, from the areas of speech and image processing, were also used:

- ADPCM speech encoder;
- LPC speech encoder;
- spectral analysis using periodogram averaging;
- image compression using the discrete cosine transform;
- edge detection using Sobel operators and 2D convolution; and
- image enhancement using histogram equalization.

Figure 1 shows the DSP model architecture used for this study. The model is based on a Very Long Instruction Word (VLIW)

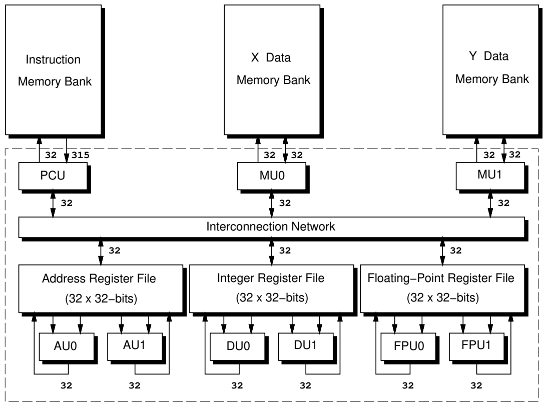

Figure 1. Model DSP Architecture

architecture [6], and was chosen for the flexibility it offers the compiler in exploiting parallelism. In VLIW architectures, the compiler is responsible for specifying the parallel execution of operations, by appropriately encoding the corresponding fields of a long machine instruction. The parallel execution of these operations is similar to the execution of horizontal microcode [4].

The model consists of nine functional units: two floating-point arithmetic units (FPU0 and FPU1); two integer arithmetic and logic units (DU0 and DU1); two address arithmetic units (AU0 and AU1); two memory access units (MU0 and MU1), used for accessing the dual data memory banks (X and Y); and a program control unit (PCU), used to fetch and decode the long machine instructions, and to execute control-type operations. Three separate, multi-ported register files are also used to store the operands of the floating-point, integer, and address units respectively. Other features, typically found in DSP architectures, are also supported. These include multiply-accumulate operations, support for low-overhead looping and specialized addressing modes.[1] All operations are assumed to execute in a single clock cycle.

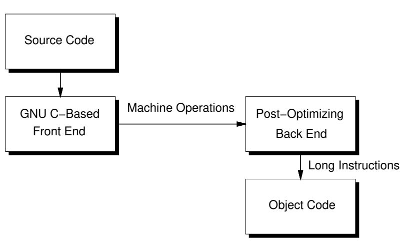

Figure 2. Optimizing C Compiler

Figure 2 shows a block diagram of the optimizing C compiler developed at the University of Toronto [7], and used in this study. The compiler front-end is based on the GNU project optimizing C compiler [8], and is used to compile source programs and generate a sequence of machine *operations*. The back-end then applies a number of DSP- and architecture-specific optimizations on these operations. Of interest to this study are the low-overhead looping (LOL), memory partitioning (MP), and operation compaction (OC) optimizing passes.

In the LOL pass, the set-up and test operations associated with the loops of a program are replaced, whenever possible, by a single operation that controls the body of the loop. The MP pass attempts to expose available parallelism in memory access operations by assigning simultaneously executable *load* or *store* operations to different memory units. The data being accessed is also partitioned to their appropriate data memory banks. Finally, the OC pass attempts to exploit the parallelism in the basic blocks of a program by packing simultaneously executable operations into a single, *long* instruction. The long instructions are then generated by the compiler.

After compiling a source program, the resulting code is executed on an instruction-set simulator that we developed to gather data on a program's run-time characteristics. This data is also used to assess the impact of the model architecture and the different compiler optimizations on execution performance. In the remainder of this paper, data gathered from the benchmark programs are presented and analyzed.

3. RESULTS AND ANALYSIS

3.1. Distribution of Operation-Level Parallelism

Table 1 summarizes the average distribution of operation-level parallelism among the instructions of the kernel and application benchmarks. While parallelism in the kernels is concentrated around 2, 3, and 5 operations per instruction, parallelism in the applications is concentrated around 1, 2, and 3 operations per instruction. The kernels therefore exhibit an average parallelism of 3.48 operations per instruction, while the applications exhibit an average parallelism of 2.11 operations per instruction. This shows that there is generally more parallelism in the kernels.

The difference between the levels of parallelism found in the kernels and the applications is to be expected. Kernels consist mostly of short loops where high levels of parallelism are available. Application programs contain other sections of code, such as control code or the intervening code between loops, where less parallelism is available. This reduces the average parallelism in application programs.

Moreover, since parallelism is exploited at the basic block level, little average parallelism, beyond *two* or *three* operations, can typically be found [9]. The low levels of parallelism found in the applications are therefore typical, while the high levels of parallelism found in the kernels are not common in more general code.

	Available Parallelism				
	1	2	3	4	5
Kernels	9.1%	27.0%	15.3%	3.7%	44.9%
Applications	39.6%	28.0%	21.6%	3.3%	7.5%

Table 1. Distribution of Operation-Level Parallelism

1. Modulo and bit-reversed addressing modes are supported at the instruction set level.

3.2. Functional Unit Utilization

Table 2 summarizes the average run-time utilization of functional units by the benchmarks, which also describes their dynamic functional behavior.

For the kernels, the utilization of functional units is almost evenly distributed between the memory units, the address units, and the floating-point unit that executes the multiply-accumulate operation (FPU1). This distribution shows that DSP kernels spend most of their execution time in loops performing operand fetch, address update, and multiply-accumulate-type operations. The even distribution also shows that the compiler is successful in finding, exposing and exploiting the high levels of parallelism available in these loops.

For the applications, the utilization of the memory and address units is also dominant. However, there is also significant utilization of the integer arithmetic units and the program control unit. This shows that in addition to kernel-like loops, applications perform a significant amount of integer arithmetic and logic operations, as well as control-type operations. The less even distribution in the utilization of the memory and address units shows that the compiler is generally less successful in exposing and exploiting parallelism in memory access and address arithmetic operations.

Functional Unit	Kernels	Applications
MU0	21.2%	25.1%
MU1	15.4%	5.4%
AU0	22.9%	24.3%
AU1	14.2%	8.2%
DU0	N/A	8.1%
DU1	N/A	2.5%
FPU0	3.9%	4.2%
FPU1	19.2%	9.8%
PCU	3.2%	12.4%

Table 2. Functional Unit Utilization

3.3. Overall Impact of Optimizing Passes

Table 3 summarizes the individual and overall impact of the different compiler optimizing passes on the execution performance of the benchmarks. Performance is measured by the average *speedup* in execution time. The speedup attained by the application of an optimizing pass *P* is the ratio of the number of cycles needed to execute a benchmark with all optimizing passes *except P* activated, to the number of cycles needed to execute the benchmark with all optimizations active.

	Optimizing Passes			
	LOL	MP	OC	Overall
Kernels	1.78	1.49	3.48	4.86
Applications	1.23	1.07	2.11	2.83

Table 3. Impact of Optimizing Passes on Benchmark Performance

The overall impact of the optimizations on performance is an average speedup of 4.86 for the kernels and 2.83 for the applications. This shows that the compiler is generally successful in exposing and exploiting parallelism, and in enhancing the run-time performance of the benchmarks. The difference in average speedup between the kernels and the applications is mainly due to the higher levels of parallelism available in the kernel benchmarks.

3.4. Impact of Low-Overhead Looping

Table 3 also shows that the average speedup attained by the use of the LOL pass is 1.78 for the kernels, and 1.23 for the applications.

The higher speedup attained for the kernels is mainly due to a significant reduction in loop overhead. When the LOL pass is not applied, only a few instructions inside the kernel loops can be used by the OC pass to pack loop test operations. In most cases, these operations require separate, dedicated instructions. The overhead of executing these additional instructions is significant, especially since kernels are usually short programs.

In the application benchmarks, there are generally more instructions inside the loops, and this provides a better opportunity for the OC pass to pack loop test operations within other loop instructions. The overhead of executing additional loop test operations is therefore less pronounced, and the speedup from the LOL pass is relatively low.

3.5. Impact of Memory Partitioning

Table 4 summarizes the distribution of data memory accesses made to the X and Y memory banks. The kernels show a more even distribution of data memory accesses, and Table 3 shows a speedup of 1.49 attained from the application of the MP pass. These are mainly due to the memory access patterns of the kernels, which easily lend themselves to parallelization.

The applications exhibit a less even distribution of data memory accesses, and a speedup of only 1.07 is attained from the application of the MP pass. This shows that the compiler is generally less successful in exploiting the parallelism in memory access operations.

	X	Y
Kernels	57.9%	42.1%
Applications	82.3%	17.7%

Table 4. Data Memory Access Distribution

3.6. Impact of Operation Compaction

Table 5 summarizes the impact of the LOL and MP passes on average operation-level parallelism. The parallelism available in a program is the ratio of the total number of operations executed, to the total number of instructions executed. Recall that instructions consist of operations that are packed together to execute in one clock cycle. Thus, the impact of an optimizing pass *P* on parallelism is the parallelism measured when all optimizing passes *other* than *P* are activated.

For the kernels, the LOL pass increases parallelism from 2.95 to 3.48 operations per instruction. This increase is due to the elimination of the additional instructions needed to execute the loop set-up and test operations, which often exhibit very low parallelism. Similarly, the MP pass increases parallelism by exposing the parallelism in memory access operations to the OC pass.

For the applications, the LOL pass *decreases* parallelism from 2.24 to 2.11 operations per instruction. When LOL is not applied, the compaction pass generally succeeds in packing the loop set-up and test operations within the instructions of a loop. Since the LOL pass eliminates these operations, the parallelism in the instructions of the loop is reduced. This reduction in parallelism, however, does not result in reduced performance. Table 3 shows that the use of the LOL pass increases performance by 23%. The use of the MP pass slightly increases parallelism by exposing the parallelism in memory access operations.

The high levels of speedup attained by the use of the OC pass, for both the kernels and the applications, shows its importance in exploiting operation parallelism. The higher average speedup attained by the kernels again shows that they have more parallelism.

	Average Parallelism		
	w/o LOL	w/o MP	Fully Opt.
Kernels	2.95	2.38	3.48
Applications	2.24	1.96	2.11

Table 5. Impact of Optimizing Passes on Parallelism

4. CONCLUSIONS

In this study, we have shown that the dynamic behavior of DSP kernels differs from that of DSP application programs. Kernels are typically short loops that repetitively perform computationally-intensive arithmetic operations, address calculations and operand fetches. Application programs, on the other hand, contain larger loops and perform more integer and control-type operations. Furthermore, kernels exhibit almost 1.6 times more parallelism than applications. Thus, kernels can achieve almost *twice* the speedup achieved by typical applications, due to their smaller loop sizes and their higher levels of parallelism.

Using DSP kernels as benchmarks can therefore be misleading, especially when evaluating the performance of DSP architectures. DSP applications are better suited for assessing the *true* performance of DSP architectures, while kernels are better suited for assessing *peak* performance. Thus, when performing architectural and compiler trade-offs, it is necessary to rely on data gathered from DSP *application programs*. This shows the need for a standardized set of public domain applications that can be used as benchmarks for comparing various DSP architectures. However, until a standard programming language is available, and the reliance on assembly language programming decreases significantly, distributing and evaluating such benchmarks will be difficult.

Our study has also shown that a HLL compiler can generate efficient code for DSP applications when given a flexible target architecture. This was demonstrated by the compiler's ability to exploit an average operation parallelism of 3.48 operations per instruction in kernel-like loops, and 2.11 operations per instruction outside inner loops. It was also demonstrated by the compiler's ability to effectively exploit such DSP architectural features as low-overhead looping, multiple memory banks, and multiple functional units. The application of the LOL pass resulted in a speedup of 1.78 for the kernels, and 1.23 for the applications. The MP pass resulted in a speedup of 1.49 for the kernels, and 1.07 for the applications. This shows the compiler's difficulty in exposing memory access parallelism outside kernel-like loops. Finally, the OC pass resulted in a speedup of 3.48 for the kernels, and 2.11 for the applications. The overall impact of these optimizing passes was a speedup of 4.86 for the kernels, and 2.83 for the applications. It is therefore possible to program DSPs efficiently using a high-level language. However, to achieve the maximum benefit, DSP architectures that are more flexible than what are currently offered are required.

Finally, our study has shown that a VLIW-based architecture is suitable for DSP applications. This is due to its ability to provide the functionality that is necessary for supporting DSP requirements, and the flexibility that enables the exploitation of available instruction-level parallelism. One drawback of a VLIW architecture is its high instruction bandwidth requirements, which place a heavy demand on the memory system. Implementing the model architecture used in this study is therefore not presently feasible. We are currently exploring solutions to the instruction bandwidth problem.

5. REFERENCES

[1] Edward A. Lee, "Programmable DSP Architectures: Part I," *IEEE ASSP Magazine*, pages 4-19, October, 1988.

[2] Edward A. Lee, "Programmable DSP Architectures: Part II," *IEEE ASSP Magazine*, pages 4-14, January, 1989.

[3] Nelson R. Manohar Alers, "The Architecture of VLSI Signal Processors," *Proceedings of the 22nd Asilomar Conference on Signals, Systems and Computers*, pages 626-630, 1988.

[4] John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[5] Mazen A. R. Saghir, *Architectural and Compiler Support for DSP Applications*, M.A.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 1993.

[6] Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10th Symposium on Computer Architecture*, pages 140-150, IEEE, June, 1983.

[7] Vijaya K. Singh, *An Optimizing C Compiler for a General Purpose DSP Architecture*, M.A.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 1992.

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

[9] David W. Wall, "Limits of Instruction-Level Parallelism," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176-188, ACM, April, 1991.