Towards Better 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

High-level language (HLL) DSP compilers continue to generate poor quality code because of their inability to exploit DSP-specific architectural features. We believe that this is chiefly due to compilers being developed after an architecture has been established and architectures being designed with little, if any, consideration to HLL programming and compiler issues. We also believe that better DSP architectures and compilers may be obtained by adopting an application-driven design methodology, whereby an architecture and its compiler are developed in parallel and where dynamic statistics assessing the impact of architectural and compiler trade-offs on the execution performance of typical DSP applications are analyzed to iteratively fine-tune the architecture and the compiler.

This paper describes the research that we are conducting at the University of Toronto to demonstrate the validity of an iterative, application-driven design methodology for DSP architectures and compilers. We describe our VLIW-based DSP architecture and optimizing C compiler as well as the instruction-set simulator and suite of DSP benchmarks we used to evaluate the effectiveness of our DSP system. We also present performance results showing that our compiler is able to successfully use DSP-specific features in our model VLIW architecture as well as expose and exploit instruction-level parallelism. The performance results also indicate areas of improvement, thus highlighting the iterative aspect of our design methodology.

1. INTRODUCTION

DSP applications typically have stringent real-time constraints. These, in turn, have motivated architects to design highly-tuned processors with special features that can efficiently execute programming constructs used frequently in DSP applications. Such hardware features include fast multiply-accumulate units, dual data memory banks, simultaneous execution of multiple operations, support for low-overhead looping, and modulo- and bit-reversed-addressing modes [1][2]. To take advantage of these special hardware features and to produce code that satisfies real-time constraints, DSP programmers typically develop their applications in assembly language. Assembly code, however, is difficult to code, debug, maintain, and port especially as applications become larger and more complex. Because programming in a high-level language (HLL) can alleviate these problems, many vendors currently provide HLL compilers for their DSP processors [3].

A 1992 survey [4], however, indicated that DSP-specific hardware features were rarely used by several commercial DSP C compilers. Included in the survey were compilers for Motorola's DSP56001 and DSP96002, Texas Instruments' TMS320C25 and TMS320C30, AT&T's DSP32, and Analog Devices' ADSP-21020. All compilers surveyed, with the exception of the AT&T compiler, were able to generate low-overhead looping instructions. On the other hand, none of the compilers supported modulo addressing nor were they able to emit code that could utilize two data banks simultaneously. Additionally, none with the exception of the Motorola compiler were able to generate code that would access memory in parallel with another operation.

The inability to use DSP-specific features results in compiler-generated code that executes significantly slower than handcrafted assembly code does. We believe that the poorer quality of the code is due to two reasons. The DSP compilers were developed after the target DSP architecture had already been established, and existing DSP architectures have been designed for human assembly-language programmers rather than for compilers.

Instead of designing the architecture and compiler separately and sequentially, our approach is to design the compiler and the architecture in parallel, in an iterative manner. In this way, the design of one component can influence the design of the other with the overall goal of achieving greater performance [5]. Figure 1 shows a block diagram of this design methodology. We start by defining a baseline architecture and an optimizing HLL compiler for this architecture. We then compile a number of DSP applications, measure their execution performance and other dynamic statistics using an instruction-set simulator, and analyze this data to iteratively fine-tune the architecture and the compiler to enhance execution performance. Since architectural and compiler trade-offs are guided by the execution performance of DSP applications, we refer to this development process as one that is *application-driven*.

Although no single HLL is particularly well-suited for developing DSP applications, we have chosen the *C programming language* for several reasons. C is a popular choice among DSP programmers because of its flexibility

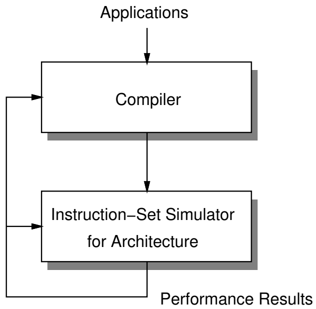

Figure 1: Application-Driven Design Methodology

and assembly-language-like features. C is also portable, making it more attractive to use than other DSP-specific languages, such as DSP/C [6], or signal-flow-graph languages, such as Gabriel [7]. Finally, the availability of the public-domain, GNU-C compiler [8] allows us to modify an existing compiler to generate code for a new architecture rather than invent a new DSP language and create a new compiler from scratch.

This paper describes the research we are conducting at the University of Toronto to demonstrate the validity of our design methodology [4][9]. Included are descriptions of our model DSP architecture, its instruction-set simulator, and our optimizing C compiler. We implemented the compiler to use the DSP-specific features of our DSP architecture as well as to expose and exploit instruction-level parallelism in DSP applications. The paper also presents experimental results showing the effect our compiler has on the execution performance of a suite of DSP benchmarks that we developed.

2. MODEL DSP ARCHITECTURE

As a point of departure for our research exploration, we are determining how a compiler can exploit DSP-specific features and parallelism in DSP applications with little concern for the cost involved in achieving greater performance. Consequently, we developed a model DSP architecture with three main objectives in mind. First, the architecture had to be a flexible target for the optimizing compiler. Features such as simple, highly-orthogonal, register-register operations make it easier for the compiler to generate more efficient code [5]. Second, the architecture had to facilitate the compiler's efforts to exploit parallelism. Finally, the architecture had to contain features that are commonly found in general-purpose, programmable DSP architectures.

To satisfy these objectives, we chose a Very Long Instruction Word (VLIW) architecture [10] as our model DSP architecture. A VLIW architecture consists of multiple functional units whose operations are controlled by corresponding fields of a very long instruction. Each field in a long instruction can be used to specify an operation that is executed on the associated functional unit. In this

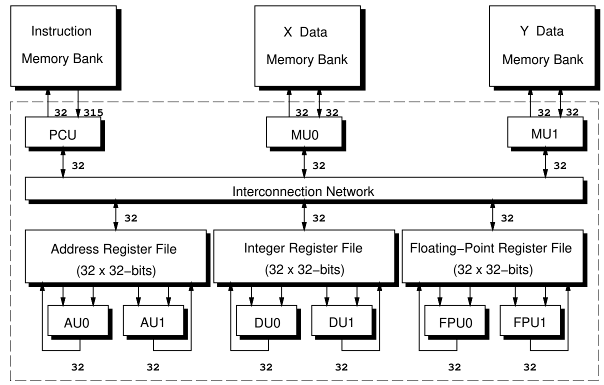

Figure 2: VLIW Model Architecture

way, multiple operations may be executed in parallel.

Our model DSP architecture is intended to be used only during the first stage of our research exploration. Obviously such an architecture can be impractically expensive to implement. Nonetheless, it is an easy target for compilers, thus expediting the initial compiler work.

2.1. Main Architectural Features

Figure 2 shows a block diagram of the model architecture we are currently using. It 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 to access the dual data memory banks; and a program control unit (PCU) used to fetch and decode the long machine instructions and to execute control-type operations. Operations that execute simultaneously in these nine functional units are specified in one long instruction.

The model is a load/store architecture. All arithmetic operations operate on register operands and store their results back to registers. Three separate multi-ported register files are used to store the operands of the floating-point, integer and address arithmetic units, respectively. Data memory is only accessed through *load* and *store* operations.

The model uses a Harvard memory architecture. The instruction-memory space is separate from the data-memory space to enable the access of instructions and data in parallel. On each clock cycle, we assume that a 315-bit instruction may be fetched from the instruction-memory bank. Furthermore, since the data memory space is divided into two data banks (X and Y), two simultaneous data-memory accesses may also be made each clock cycle.

The model architecture also uses a simple, three-stage, "fetch, decode, execute" pipeline. To eliminate the effect of pipeline hazards and to enhance performance, we assume that all operations execute in a single clock cycle.

We also assume that operands are read from registers before registers are written during a clock cycle.

Other features commonly found in DSP architectures are also supported at the instruction-set level. These include multiply-accumulate operations, support for low-overhead looping, and specialized addressing modes such as modulo and bit-reversed addressing. Low-overhead looping is supported through a number of control registers and a system stack. The control registers include a loop-count register, a loop-address register and a loop flag. The system stack is used to support nested loops. Modulo addressing is supported in a slightly different manner than what is provided by DSP processors such as the Motorola 96002 [11], or the Texas Instruments TMS320C30 [12]. The model architecture includes *modulo-increment* or *-decrement* operations similar to those provided by other DSP processors. However, unlike the aforementioned processors, the model architecture uses address registers rather than specialized registers to store the lengths of different arrays.

2.2. Instruction Set Simulator

To model the behavior of the architecture and to gather run-time statistics on the architectural characteristics of DSP applications, we developed an instruction-set simulator. The gathered statistics are used to guide the development of the architecture and the compiler by measuring the impact of architectural and compiler trade-offs on the execution performance of DSP applications.

Execution performance is measured by the *clock-cycle count* corresponding to a program's execution time, and the *amount of operation-parallelism* exposed and exploited by the compiler and the architecture. By monitoring these performance metrics, we can assess the impact of architectural and compiler trade-offs on execution performance.

Other statistics that may be used to guide the development of the architecture and its instruction set include functional-unit utilization, dynamic operation counts, dynamic instruction counts, addressing-mode utilization, register-set utilization, branching behavior and system stack depth.

3. OPTIMIZING C COMPILER

We developed our optimizing compiler to exploit the DSP-specific features of the model architecture and to expose and exploit instruction-level parallelism. To simplify the design of the compiler, we decoupled the compilation process from the optimization process. Figure 3 shows a block diagram of our optimizing C compiler with its GNU-C compiler front-end and its optimizing back-end. The optimizing passes that are currently implemented include modulo addressing, low-overhead looping, data partitioning and operation compaction. These passes may be invoked either individually or in various combinations at compile-time.

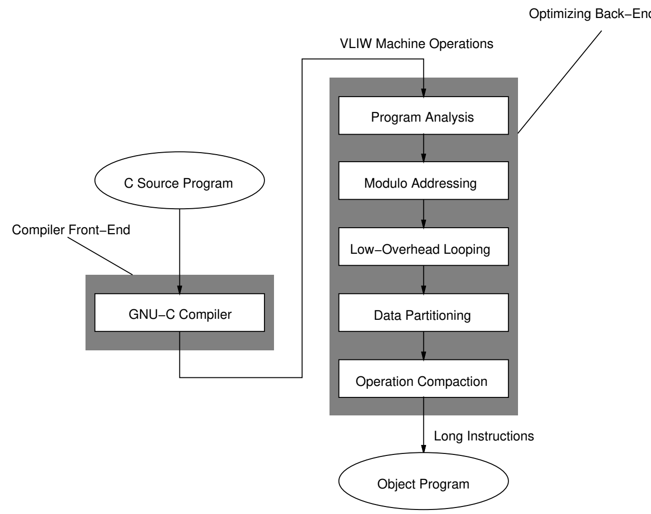

Figure 3: Optimizing C Compiler

3.1. The GNU-C Compiler Front-End

The front-end is responsible for translating a C source program into a sequence of VLIW machine *operations*. We modified the GNU-C compiler to generate code for our model VLIW architecture. GNU-C performs traditional code optimizations such as common subexpression elimination, strength reduction and register allocation [13]. However, the generated code does not exploit any form of parallelism, nor does it use any underlying features of our model architecture.

3.2. The Optimizing Back-End

The optimizing back-end is responsible for enhancing the execution performance of the code generated by the front-end. Its main functions are to apply target-specific optimizations and to exploit instruction-level parallelism.

To assist the optimizing back-end, programs should be written in a specific style of C. These programming constraints are required by specific optimizations and are mentioned in the following descriptions of the optimizing passes. If this programming style is not used, the resulting code will execute correctly, albeit more slowly. In the long term, we plan to make this coding style more transparent to the programmer and thus, allow more diverse programming styles. Known optimizing techniques can be used to do this.

3.2.1. Program Analysis

Prior to the application of the main optimizing passes, the code is divided into basic blocks and the control-flow graph of the program is constructed. Information on the program's control-flow, data-flow and alias characteristics is then extracted [13]. This information is used by the subsequent optimizing passes.

3.2.2. Modulo Addressing

The modulo addressing (MA) pass enables one-dimensional arrays to be accessed in a modulo fashion. Modulo addressing can be difficult for HLL compilers to support because this type of addressing is not normally specified in the syntax of the language. To overcome this problem, we changed the *semantic meaning* of arrays in the C programming language. Thus, whenever the modulo addressing pass is invoked, *all* arrays are treated as 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 accessed in a modulo fashion.

The algorithm for generating modulo addressing code to access an array is fairly straightforward. The compiler first determines the length of the array and assigns an address register to store the array length. It then uses aliasing information to identify any pointers that access the array. Finally, the compiler generates code to initialize the assigned address register with the array length and generates the appropriate modulo-increment or -decrement operations.

To simplify the modulo-addressing hardware, an array should be stored at a memory address that is a multiple of the smallest power-of-two greater than or equal to its own length. Thus, another responsibility of the compiler, and not the programmer's, is to determine the appropriate addresses at which to store arrays. To help the compiler determine the starting address of an array as well as its length, arrays should be statically defined. This restriction, however, is also imposed by other commercial DSP processors that provide hardware support for modulo addressing.

3.2.3. Low-Overhead Looping

The low-overhead looping (LOL) pass enhances the execution performance of a loop by removing the overhead operations associated with the loop and replacing them by a single operation that controls the body of the loop.

Overhead operations that typically degrade the execution performance of a loop include loop initialization, testing for the loop termination condition and branching to the beginning of the loop. Since these overhead operations are generated by the front-end of the compiler in a consistent fashion for all loops, this optimizing pass can identify these operations easily by using pattern matching.

Information from data-flow analysis is then used to determine whether any operations within the body of a loop are dependent upon the overhead operations. Such dependencies may prevent the use of low-overhead looping. For example, if the loop counter is used within the body of a loop to scale the elements of an array, the operations that initialize and update the value of the loop counter cannot be removed. This problem can easily be solved, however, by using a *dummy variable* instead of the loop counter to scale the elements of the array.

Information from data flow analysis is also used to determine the number of times a loop is to be executed. The actual number does not need to be known at compile time; only the location of the iteration count is needed. Thus, low-overhead looping can also be applied to loops that iterate a variable number of times.

Once the iteration count of a loop is determined and dependences on the loop counter do not exist, the loop-overhead operations are replaced by a single, low-overhead looping operation that controls the execution of the loop.

3.2.4. Data Partitioning

The data partitioning (DP) pass exposes parallelism in *load* operations to exploit the dual data-memory banks of the model architecture. By appropriately partitioning the global variables of a program between both data-memory banks, the parallelism in operations that access these variables may be exposed.

To represent the parallelism available in *load* operations, an undirected *interference graph* is first constructed. The nodes of the graph represent the global variables of the program. An edge connecting a pair of nodes in the graph indicates that the corresponding variables may be accessed in parallel and that they should be allocated to different data-memory banks.

Edges are added to the graph by examining the operations of each basic block. If a pair of operands are fetched from data memory using a pair of *load* operations, the parallelism in fetching these operands can be exposed by allocating the accessed variables to different data-memory banks. Aliasing information is used to determine the variables that are accessed, and an edge that connects the corresponding nodes is added to the graph. A *cost* is also assigned to each edge and is equal to the depth of the block in the control-flow graph. The deeper the block, the more important it is to allocate the corresponding variables to different data-memory banks. Variables are therefore partitioned to expose the most memory-access parallelism in inner loops.

Once the interference graph is constructed, its nodes are partitioned into two sets by finding a *maximal independent set* on the nodes of the graph and its corresponding *complement set* [14]. By definition, a maximal independent set is any subset of the nodes of a graph 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. The complement set consists of those nodes that do not belong to the maximal independent set.

Because the nodes of a maximal independent set are not edge-connected, their corresponding variables may be allocated to the same data-memory bank. However, the nodes of the complement set may be edge-connected and allocating their corresponding variables to the same data-memory bank may hide parallelism. The resulting degradation in performance may be quantified by the sum of the costs associated with the edges connecting nodes of the complement set. Since the maximal independent set is not unique, an *optimal* partitioning may be found by calculating all possible maximal independent sets and their corresponding complement sets, and choosing the partitioning that incurs the least cost. This way, the most parallelism is exposed. The variables corresponding to the maximal independent set are therefore allocated to one data-memory bank, and the variables corresponding to the least-cost complement set are allocated to the other.

After partitioning the nodes of the interference graph, the memory-bank allocations of the corresponding variables are propagated through the program. For each memory-access operation, alias information is used to determine the data-memory bank to which the variable being accessed has been allocated. Depending on which data-memory bank is used, the memory-access operation is assigned to the appropriate memory unit. This assignment is used in the *operation-compaction pass* when scheduling memory operations together with other operations into long instructions.

To simplify the data-partitioning algorithm, stack references are partitioned using a different method. Successive stack locations are assigned to different memory banks, and the *same* stack address is used to access both data-memory banks. Moreover, to keep the partitioning of stack operations simple, arrays must not be defined as local variables. We are working on augmenting the data partitioning algorithm to handle stack references.

3.2.5. Operation Compaction

The operation compaction (OC) pass exploits the parallelism in the basic blocks of a program by packing simultaneously-executable machine operations into long instructions. Recall that a long instruction can specify the simultaneous execution of up to nine machine operations.

To help the compiler expose available parallelism in a basic block, a *dependence graph* is first constructed from the dependence relationships between the operations of the block. In general, two types of dependencies may exist between a pair of operations, i and j, where i precedes j. A *strong* dependence exists between i and j if i writes a register that is read by j, or if both i and j write the same register. Conversely, a *weak* dependence exists between i and j if i reads a register that is written by j. Since registers are read before they are written within a clock cycle, operations with weak dependencies may be simultaneously executed.

The dependence graph is used to calculate a *data-ready set*, which consists of those operations that are ready for scheduling in a long instruction. An operation is *data-ready* if all the predecessor operations with which it has strong dependencies have already been scheduled, or if the predecessor operations with which it has weak dependencies are also data-ready.

After computing the data-ready set, data-ready operations are scheduled into long instructions. Selecting operations from the data-ready set is based on a priority. The larger the number of operations that depend on a data-ready operation, the more important it is to schedule it early. Moreover, scheduling a data-ready operation depends on the availability of a functional unit to execute the operation. When it is no longer possible to schedule any more operations, the long instruction is issued and the data-ready set is updated. This continues until all operations in the block are scheduled. The long instructions are the final output of the optimizing back-end.

4. EVALUATION OF COMPILER AND ARCHITECTURE

To evaluate our optimizing compiler and model architecture, we compiled a suite of DSP benchmarks and executed the resultant code on our instruction-set simulator. While executing the code, the simulator would also gather statistics on the run-time behavior and architectural characteristics of the benchmark being executed. In addition, we verified the functional correctness of our compiler and simulator by checking that a program when executed on our experimental DSP system produces the same output as it produces when executed on a commercial Sun SPARCstation system. By analyzing the run-time statistics produced by the simulator, we were able to assess the impact of our optimizing compiler and model architecture on the execution performance of DSP programs. In this section, we present a summary of our analysis. A more detailed description of the statistics can be found in [15].

The suite of DSP benchmarks consists of twelve kernels and six applications that we wrote in the C programming language. The kernels are based on six algorithms that are commonly used in DSP applications:

- radix-2, in-place, decimation-in-time fast Fourier transform;

- finite impulse response (FIR) filter;
- infinite impulse response (IIR) filter;
- normalized lattice filter;
- least mean squared adaptive FIR filter; and
- matrix multiplication

For each algorithm we implemented two kernels of different problem sizes. The applications are from the areas of speech and image processing and they include:

- ADPCM speech encoder based on CCITT G.721;
- 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

Table 1 summarizes the individual and overall performance impact of the different optimizing passes. 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 clock 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.

With the exception of the speedup numbers for the modulo-addressing pass, the averages are for all the kernels and applications. Because not all the programs require modulo addressing, the average speedups for this optimizing pass are for the three kernels that use it and the ADPCM application.

As mentioned in the introduction, our design methodology is an iterative process. The results we have obtained thus far are being used to identify strengths as well as weaknesses in our experimental system.

The results in Table 1 indicate that our optimizing compiler is able to improve the execution performance of the DSP benchmarks by exploiting the DSP-specific features of the model architecture and by exposing and using parallelism. The success of the compiler in enhancing performance is due, to a large degree, to the flexibility of the model VLIW architecture, which provides the necessary functionality needed to support DSP requirements as well as a simple, regular, and orthogonal instruction set that is a suitable target for our compiler. These results support our belief that a HLL compiler is capable of improving performance, provided a sufficiently flexible target architecture, such as a VLIW-based architecture, is used.

Table 1 also identifies areas that could use some improvement and provides guidance in prioritizing the improvements. For example, the data-partitioning pass has the smallest average speedup for both kernels and applications. This together with statistics on average parallelism and usage frequency of the dual memory banks

	Optimizing Passes				
	MA	LOL	DP	OC	Over all
Kernels	2.72	1.78	1.49	3.48	4.86
Applications	1.08	1.23	1.07	2.11	2.83

Table 1: Impact of Optimizing Passes on Performance

has led us to re-evaluate our algorithm. We are currently modifying our algorithm and believe that these changes will result in better performance. The application speedup for modulo addressing is also small. Nevertheless, we have decided not to investigate this further at the present time. Because so few programs in our suite require modulo addressing, the performance impact of improving the modulo-addressing pass will be more limited in scope than that of improving the data-partitioning algorithm.

Another part of the system that can be improved is the benchmark suite, which contains kernels and applications. Kernels are typically loops that execute a few computationally-intensive arithmetic operations, pointer update operations and operand fetches. On the other hand, application programs execute more integer and control-type operations, and consist of loops with more operations. By separating the run-time statistics for kernels and applications, we can observe how these differences in program characteristics cause differences in performance. Whether the optimizing passes are invoked individually or in combination, the average speedups for the kernels are always 40-70% higher (and 2.5 times higher for the modulo-addressing pass) than those for the applications. Since the trend in DSP programs is towards larger, more complex applications, relying on kernels alone to guide the development of new DSP architectures and compilers may lead to optimistic performance expectations. To better demonstrate that our conclusions are reasonably general, we are currently augmenting our suite with more applications.

5. SUMMARY AND CONCLUSIONS

Contemporary HLL DSP compilers are still unable to exploit DSP-specific architectural features such as dual data-memory banks or modulo addressing modes. Furthermore, the tightly-encoded instruction sets of most DSP architectures greatly restricts a compiler's ability to exploit instruction-level parallelism beyond a few fixed instances. The quality of the resultant compiler-generated code is therefore poor when compared to that of hand-coded assembly language code, and DSP programmers continue to use assembly language to achieve greater performance for DSP applications.

We believe that the poor performance of HLL DSP compilers may be attributed to two reasons. Compilers are developed after a target architecture has been established and DSP architectures are not designed with HLL compilers and programming in mind. We also believe that these problems can be overcome by adopting an application-driven design methodology whereby an architecture and its compiler are developed in parallel, and where dynamic statistics assessing the impact of architectural and compiler trade-offs on the execution performance of typical applications are analyzed to iteratively fine-tune the architecture and the compiler.

To substantiate these beliefs, we developed a VLIW-based model DSP architecture in parallel with developing an optimizing C compiler that is capable of exploiting DSP-specific hardware features as well as instruction-level parallelism. We have developed and implemented compiler algorithms that can use hardware features such as modulo addressing, low-overhead looping, and dual data memory banks. To evaluate our system, we also developed a suite of DSP benchmarks and an instruction-set simulator that can gather statistics on the run-time behavior of DSP programs.

In this paper, we described our model architecture, optimizing compiler, benchmark suite, and instruction-set simulator. We also presented performance results that highlighted the strengths and weaknesses of our system. The statistics showed that our compiler is successful in enhancing performance by exploiting DSP-specific features and by exposing and exploiting instruction-level parallelism. Hence, these results support our belief that a HLL compiler is capable of improving performance provided a sufficiently flexible target architecture is used.

The performance success of the compiler is due, to a large part, to the flexibility of the model VLIW architecture. Now that we are confident that our model VLIW architecture is a good target for our optimizing compiler, we are currently exploring the cost issues of implementing a VLIW architecture. An obvious drawback is the high instruction-memory bandwidth requirements which place heavy demands on the memory system and which can be too expensive and impractical to implement.

The statistics also indicated areas of improvement, thus highlighting the iterative aspect of our design methodology. An analysis of the results suggests that improvements to the algorithm for partitioning data should provide substantial gains in performance. In addition, the results showed a significant difference in performance gains between kernels and applications. Consequently, to better demonstrate that our conclusions are reasonably general, we are currently augmenting our benchmark suite with more applications.

6. ACKNOWLEDGEMENTS

This research has been funded by a grant from the Information Technology Research Centre, a Centre of Excellence supported by Technology Ontario.

7. 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] Ray Weiss and Julie Anne Schofield, "EDN's DSP-Chip Directory," *EDN*, Vol. 38, No. 20, Pages 57-109, September 30, 1993.

[4] 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.

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

[6] Kevin W. Leary and William Waddington, "DSP/C: A Standard High Level Language for DSP and Numeric Processing," *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, Pages 1065-1068, IEEE, 1990.

[7] Edward A. Lee, et. al., "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 37, No. 11, Pages 1751-1989, November 1989.

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

[9] 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.

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

[11] Guy R. L. Sohie and Kevin L. Kloker, "A Digital Signal Processor with IEEE Floating-Point Arithmetic," *IEEE Micro*, Pages 49-67, December 1988.

[12] Panos Papamichalis and Ray Simar, Jr., "The TMS320C30 Floating-Point Digital Signal Processor," *IEEE Micro*, Pages 13-29, December 1988.

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1988.

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

[15] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee, "Application-Driven Design of DSP Architectures and Compilers," *Proceedings of the International Symposium on Acoustics, Speech and Signal Processing*, Pages II-437-II-440, IEEE, 1994.