Compiler Feedback in ASIP Design

Frederick Onion, Alexandru Nicolau, and Nikil Dutt

Technical Report 94-2
September 6, 1994

Department of Information and Computer Science
University of California
Irvine, CA 92717-3425
(714) 856-4625

fonion@ics.uci.edu

Abstract

The recent trends toward software/hardware system solutions have placed new emphasis in the role of the compiler. As the systems become more and more complex, the role of the compiler becomes increasingly important. In spite of this fact, however, the compiler is rarely used during the development of the system. This paper presents a framework for providing feedback from an optimizing compiler into the design of an ASIP. The optimizing compiler is used to assess the hardware needs of a suite of applications to which the ASIP is to be tuned. By incorporating the compiler into the design process, the design space is increased as more information is provided at an earlier stage during the design process. Our initial study involves detecting potentially chainable operation sequences using scheduling techniques developed for exploiting instruction-level parallelism. Results of this study are included.

1 Introduction

As designers migrate toward the more economical software/hardware co-design paradigm, the role of the compiler in application-specific systems becomes more and more prominent. The increasingly popular ASIP (Application Specific Instruction-Set Processor), for example, becomes a much more effective tool if it is accompanied by a compiler capable of taking advantage of its application-specific properties. The ASIP offers a balance between the two extremes of ASICs (Application Specific Integrated Circuits), and general programmable processors. It offers the advantage of custom hardware for certain tasks (like a multiply-adder found in many DSP chips) as well as the flexibility of an instruction set. With a compiler capable of generating efficient code for its customized instruction-set, the ASIP becomes a highly flexible application-specific tool, capable of being reconfigured in a short turnaround time.

The typical design of ASIP systems begins with the processor. Application specific hardware extensions, which the designer anticipates will improve the performance of a system are added to extend a base processor. Next, the compiler is developed (if the ASIP is even to have a compiler), using various techniques to try and take advantage of the new hardware extensions. There are two problems with this process. The first problem is that the designer must decide what hardware extensions will be most beneficial for the system without knowing how well the compiler is going to be able to take advantage of those extensions. The second problem is that the compiler writer is left with the difficult task of translating a high-level language program into efficient code that takes advantage of the dedicated features of the ASIP - a task that may not always be possible to a satisfactory degree.

We are advocating a different kind of relationship between an ASIP and its compiler. This relationship should be a symbiotic one, both hardware and compiler working together cooperatively, to produce the most efficient system possible for the least cost. To this end, it is imperative that the compiler become integrated into the design process, assisting the designer in determining how to best customize the processor for a given set of applications. This is particularly true for optimizing compilers today, which may be capable of much more than simple translation. Optimizations like loop-pipelining and beyond basic-block scheduling are capable of altering the program graph in non-obvious ways, and potentially exposing opportunities for performance enhancement that weren't visible before.

In this paper, we present a framework for relating advanced compiler optimizations to the design of an ASIP. The compiler is used to assess the hardware needs of a suite of applications to which the ASIP is to be tuned, providing feedback to the designer to aid in the selection of hardware extensions to the processor. By incorporating the compiler into the design process, the designer is provided with more information about the potential performance of the system at an earlier stage in the design process. This means that better decisions can be made earlier in the design, resulting an a more efficient system in the end.

2 Related Work

There have been several projects which incorporated the compiler into the design process. In [1], the high level synthesis component (called Piper) of the ADAS design system generates a re-order table for use by the compiler of the system. This reorder table is a way to communicate information about the application-specific properties of the chip being designed to the compiler in an automatic fashion. This system helps make the compiler development an easier task (even automatic), but the communication is one way - there is no feedback from the compiler provided in the design process. One research project whose goals are similar in concept to those pursued here is presented in [2]. In this project, an analytical model is presented for predicting the program parallelism detectable when varying the scope of the concurrence detection. Its goals are similar to ours because it uses results of a parallelizing compiler (the Multiflow TRACE SCHEDULING compiler) as input to the model, which in turn can be used in assessing the effectiveness of an architecture with respect to a particular application type. The PEAS-I system, presented in [3], begins with a set of candidate instructions, and based on results of an application program analyzer, chooses the subset of instructions which provides the best performance for the applications under consideration, under the constraints of chip area and power consumption. This system uses the compiler as part of the design process, but only in a limited way - it restricts the candidate instructions to be those that could be generated by the compiler as it was written.

There have also been several different approaches proposed recently on how best to automatically customize an instruction set processor to a particular application type. One approach, taken by Holmer in [4] and [5], is to begin with a completely defined data path, and then design the instruction set by combining the microoperations defined by the datapath into instructions to optimize the cycle count and code size of a set of benchmarks. In [6], an instruction-set matching and selection methodology for DSP and ASIP code generation is presented. This methodology provides a way for the compiler to take advantage of the application-specific properties of an ASIP, but does not look at communication from the compiler to the ASIP design. Finally, in [7], a technique called 'bundling' is presented for generating an instruction set for an ASIP based on the results of an analysis tool.

3 Overview of the proposed approach

Our approach is based on incorporating the compiler into the design process. Using modern optimization techniques, compilers are capable of program graph altering transformations which may expose properties of an application that are not immediately obvious. These exposed properties can then potentially be exploited by providing application specific hardware to take advantage of them. The complete process, as depicted in Figure 1, starts with an optimizing compiler gathering information about a

sample benchmark set. This information is provided to the ASIP design stage, where application specific hardware is synthesized based on the information provided by the compiler. The final product is the customized ASIP, as well as an optimizing compiler - customized to the ASIP by incorporating the optimizations that were used in the analysis phase.

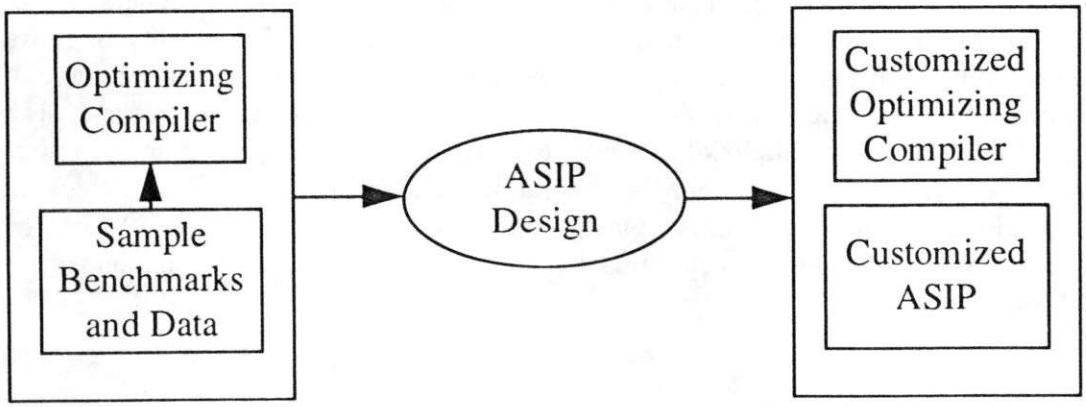

Figure 1 - The role of the compiler in ASIP design

This general scheme provides a framework for the integration of the compiler into the design process. The next step is to define more precisely the flow of information between the compiler and the ASIP design. Because of the diversity of compiler optimizations and customized hardware for ASIPs, it is difficult to define this information flow in a general fashion. We will start, instead, by looking at one hardware optimization in particular, and exploring how that optimization can be used more effectively with input from the compiler.

4 Operator chaining - an initial study

As an initial study into the effectiveness of incorporating compiler feedback into the ASIP design process, we have chosen to look to take a common application-specific optimization, operator chaining [8], and use a compiler to aid in the detection of operation sequences which would be best implemented as chained operations. The MAC (multiply and accumulate) instruction found in many DSP processors (like the TMS320C5x from Texas Instruments [9]) is an example of a chained instruction. Data is passed directly from one operation to the next, avoiding the overhead of storing the intermediate result back to a register file, as well as the fetching and decoding of an additional instruction.

4.1 Operation sequences

In order for operator chaining to be an effective optimization for a given application, the application must have operation sequences with data flow between each operation, matching the chained operations implemented in the ASIP. For example, DSP

applications often have a frequently occurring sequence consisting of a multiply operation whose result is used as an operand to an addition operation. Thus, DSP applications can often take advantage of the MAC instruction available in many DSP processors.

Since the chaining of operations depends on the ordering of the instructions, we have chosen to (initially) relate advanced parallelizing compiler optimizations to the detection of chainable operation sequences. By taking advantage of the compiler's ability to move operations around in the program, we provide the designer with a much broader range of possibilities when selecting which operations to implement as chained sequences. The approach will be to use a parallelizing compiler to compile a suite of application programs, and then, perform an analysis on the operation sequences that are available for implementation as chained sequences. Once the analysis is complete, the results can be used by the system designer to determine which operation sequences to implement as chained instructions.

4.2 Sequence analysis

The concept of exploiting commonly occurring sequences for performance enhancement is not new. It was studied by several researchers in the late 70's and early 80's, mostly in the context of generating new microprograms to execute the operation sequences [10, 11, 12, 13]. This enhanced the performance by eliminating the fetch and decode for the operations in the sequences, and by decreasing the size of the object code. Our approach has similar goals, with the exception of how the performance enhancement is achieved. Instead of reducing the fetch and decode overhead, we are interested in synthesizing specialized hardware (chained functional units) to execute these sequences more efficiently than they could be with the original data path.

By using parallelizing compiler optimization techniques we are able perform much more extensive sequence analysis. Previous efforts to identify frequently occurring sequences in programs were restricted to the operation ordering created by the compiler, which is derived from the sequential statements in the high-level language in which the application was programmed. The analysis was either conducted dynamically, on an execution trace of a program [13], or statically, on the code generated by the compiler (with profile information, in some cases) [10, 11, 12].

Our analysis adds the ability to alter the program graph of the compiled application through the use of advanced instruction-level parallelizing scheduling techniques. In particular, we utilize a technique called percolation scheduling [14], which provides a set of semantic-preserving transformations allowing the movement of operations both within and across the basic blocks of a program (constrained, of course, by the data dependencies in the program). This allows us to search a much broader set of possibilities for potential sequences because we are no longer constrained by the sequential nature of the source program.

5 Experiments

5.1 Sequence detection analysis

Our initial experiments to test the effectiveness of incorporating compiler feedback into the ASIP design process concentrated on relating parallelizing compiler optimizations to the detection of chainable sequences. Using feedback from the optimizing compiler, we were able to uncover a large number of potential sequences with relatively high frequencies which would be suitable for implementation as chained operations. For an operation sequence to be suitable, it must have data flow from the result of a preceding operation to an operand of a succeeding operation. Only those sequences exhibiting this property were considered during these experiments.

The analysis of the benchmark programs was conducted as shown in Figure 2. The benchmark source programs were first compiled by a front end compiler - a version of the Gnu C Compiler (gcc) which was modified to generate a 3-address code. This 3-address code was then used as input to a simulator, in addition to the sample data, to provide profile information for each of the applications. The resulting 3-address code with profile information was then optimized in step 3 using the UCI VLIW compiler. Three levels of optimization were performed - 1) no optimization, 2) full optimization with loop pipelining and percolation scheduling but without register renaming, and 3) full optimization with loop pipelining, percolation scheduling, and register renaming. Each of the resulting optimized program graphs was then fed to the sequence analyzer, which performed a branch and bound search on the graph to detect all potential operation sequences.

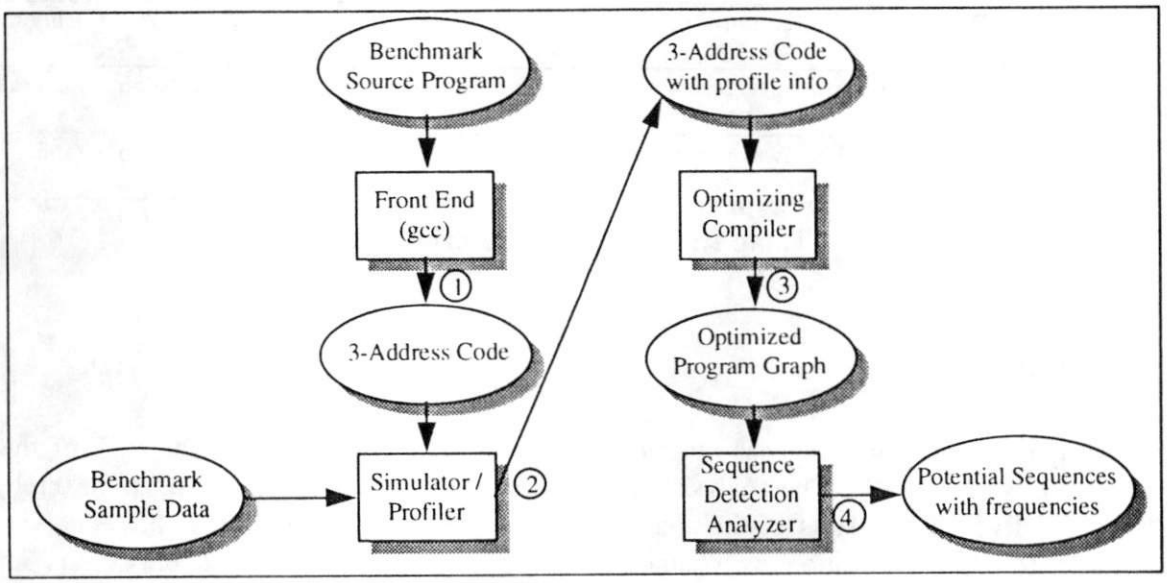

Figure 2 - The sequence detection process

5.2 Benchmarks

For our initial experiments, we selected a set of DSP benchmarks on which to perform our analysis. These benchmarks constitute a wide range of DSP applications, ranging from a complete implementation of edge detection using two-dimensional convolution, to a simple bspline stream filter. Table 1 lists each benchmark along with its description and the data used as input to the benchmark. Several of these benchmarks were adapted from examples presented in [15].

Benchmark	Description	Data Input
fir	35-point lowpass floating point FIR filter with cutoff at 0.2.	Random array of 100 floating point values
iir	Infinite impulse response filter - a 3-section (6th order) Chebyshev filter with 1dB passband ripple and cutoff frequency of 0.3*fs.	Random array of 100 floating point values
pse	Power spectral estimation using FFT.	Random array of 256 floating point values.
intfft	Interpolate 2:1 using FFT and inverse FFT.	Random array of 100 floating point values
compress	Discrete cosine transformation used to compress an image by a factor of 4:1.	24x24 8-bit image
flatten	histogram flattening using gray level modification	24x24 8-bit image
smooth	3x3 Gaussian blur lowpass filter	24x24 8-bit image
edge	Edge detection using two-dimensional convolution	24x24 8-bit image
sewha	Sewha's finite impulse response (FIR) filter.	Stream of 100 random integer values
dft	Discrete fast fourier transform	Stream of 256 random integer values
bspline	B Spline finite impulse response (FIR) filter	Stream of 256 random integer values
feowf	Fifth order elliptic wave filter	Stream of 256 random integer values

Table 1 : Benchmark descriptions

6 Results

The sequence detection analysis outlined in section 5.1 was performed for each of the benchmarks in Table 1. The analysis was performed for sequences of length two, three, four, and five. The results of the analysis for each benchmark were a set of sequences suitable for implementation as chained operations. Each of the sequences has an associated dynamic frequency which is the percentage of execution time for which that sequence accounts as calculated from the profile information collected in step 2 of the sequence detection process (see Figure 2).

6.1 Combined benchmark sequence analysis results

This first set of figures, shows the frequencies of all sequences detected across all of the benchmarks combined. This information was collected by performing sequence detection for each individual benchmark, and then combining the results of all the benchmarks together. The analysis was performed using three levels of optimization in the compiler so that the effects of the optimizations on sequence detection could be assessed. The figures below show the effects of the three levels of optimization which were used during the sequence analysis.

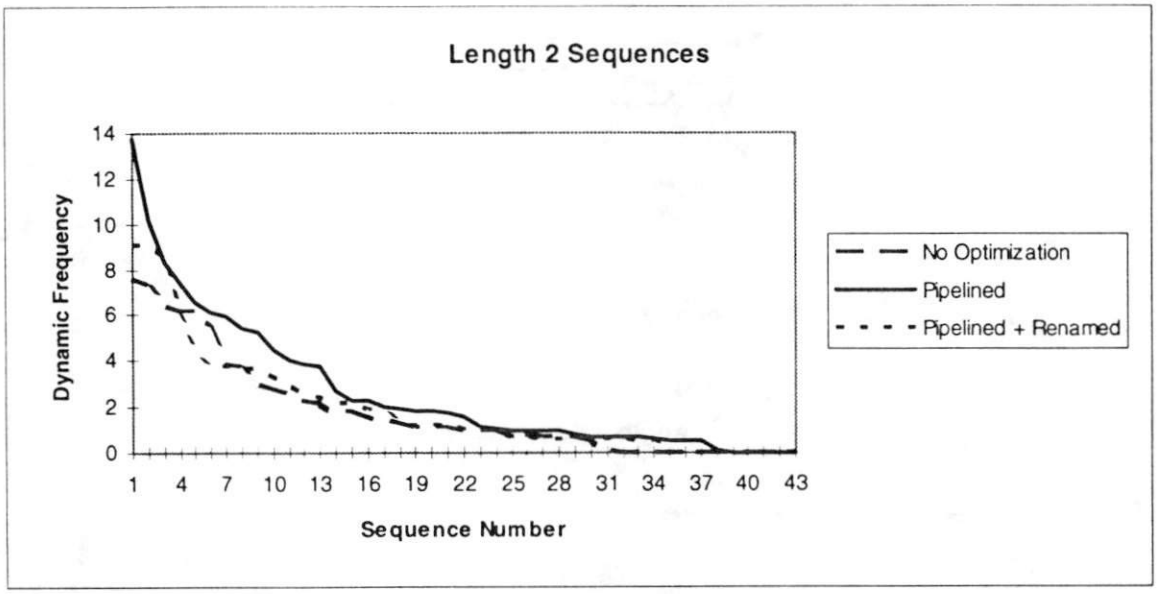

Figure 3 : Length 2 sequences detected using three levels of optimization

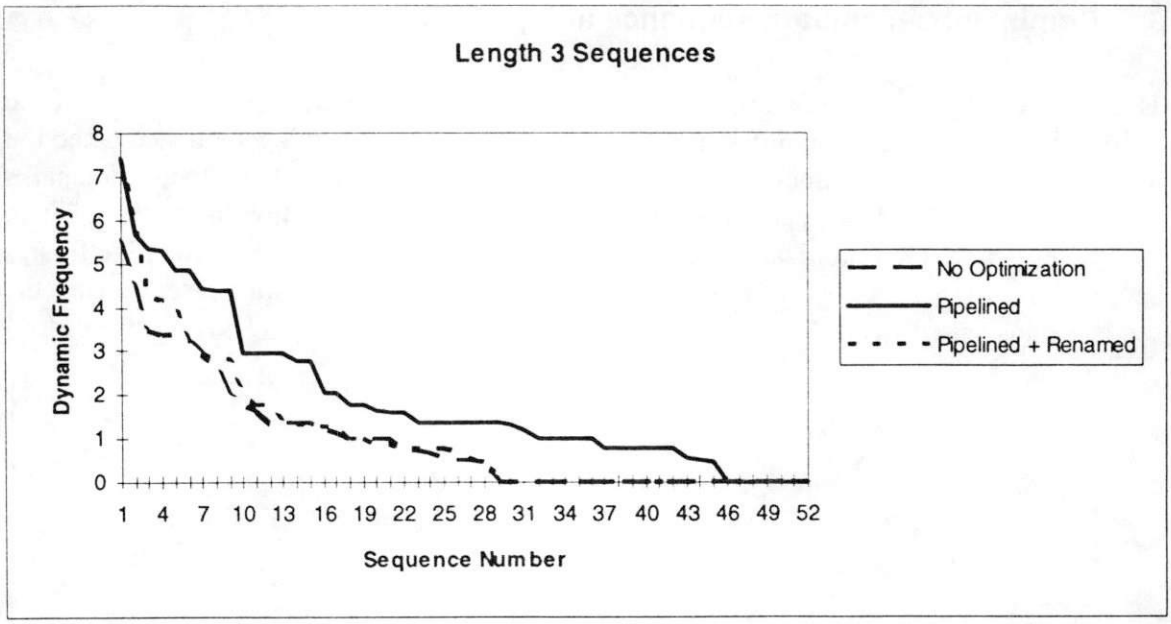

Figure 4 : Length 3 sequences detected using three levels of optimization

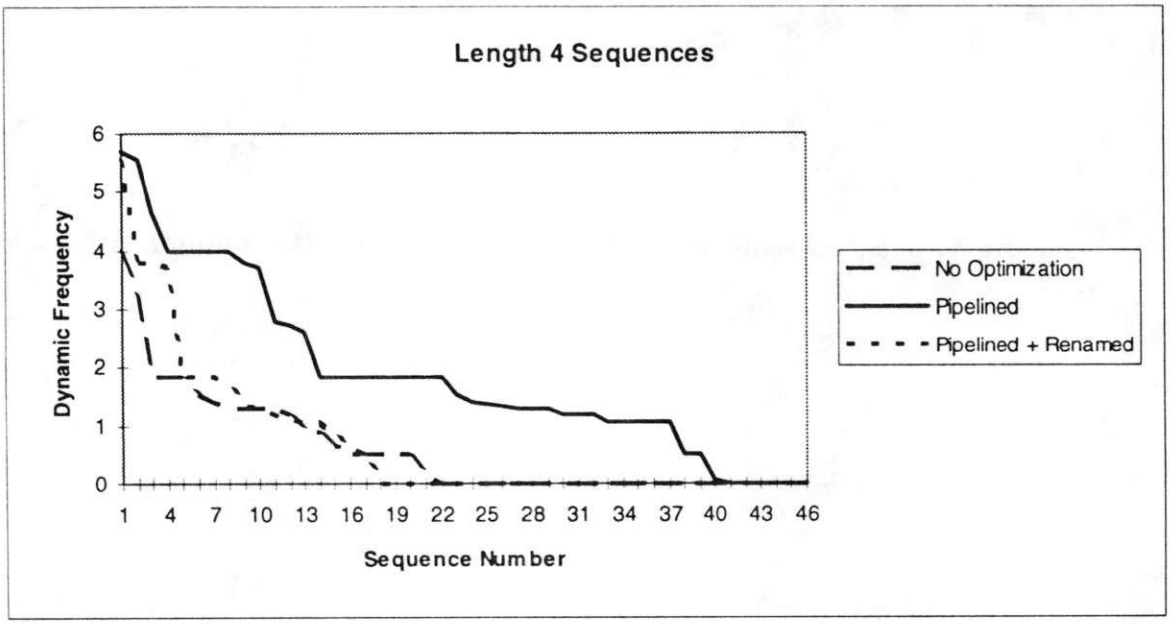

Figure 5 : Length 4 sequences detected using three levels of optimization

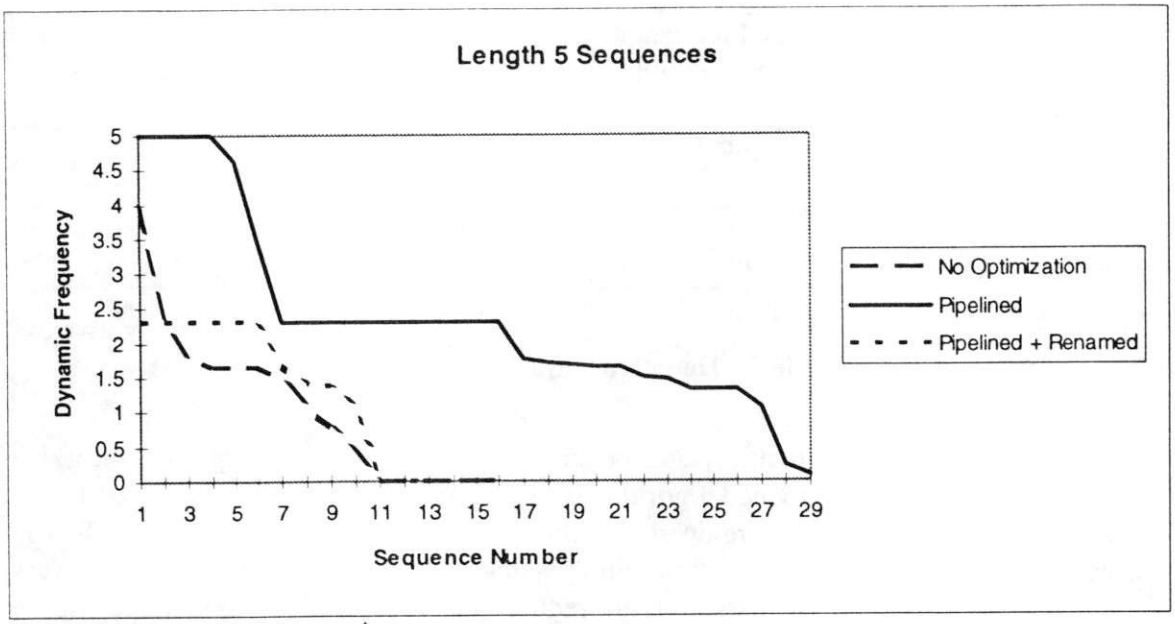

Figure 6 : Length 5 sequences detected using three levels of optimization

6.1.1 Impact of optimizations on sequence detection

We found that performing percolation scheduling and loop pipelining during the analysis phase significantly improved the detection of operation sequences. The code motions allowed us to see data flow occurring across the boundaries of basic blocks and detect that data flow as potential sequences. We also found, however, that the optimization of register renaming tended to have a negative effect on the detection of sequences. Register renaming is an effective optimization for moving operations as high as possible in a program, so that they may be scheduled earlier and thus take advantage of any parallel resources available. When this optimization was used during sequence analysis, however, it tended to move operations which had data flow between them (i.e., those operations suitable for chained operation implementation) away from each other, communicating only through the renamed register, thus eliminating the potential operation sequence.

6.1.2 Something old, something new...

The actual sequences which were detected for the complete set of benchmarks during sequence analysis confirmed some widely held beliefs regarding what operation sequences are most beneficial for DSP applications, in addition to uncovering some surprises which may not have been considered before as potential chained instructions for DSP applications. Table 2 shows several example sequences and their corresponding frequencies using the three levels of optimization.

Operation Sequence (with result to operand data flow)	Dynamic Frequency (No Optimization)	Dynamic Frequency (Pipelined)	Dynamic Frequency (Pipelined + Renamed)
multiply-add	5.6%	8.33%	9.10%
add-multiply	2.25%	13.78%	9.06%
add-add	7.64%	10.15%	8.67%
add-multiply-add	3.38%	7.42%	5.95%
multiply-add-add	2.03%	4.86%	7.40%

Table 2 - Detected sequence examples

As expected, the sequence multiply-add occurred in relatively high frequency, verifying that the MAC instruction is indeed a good choice for DSP processors. In addition to the MAC instruction, however, there were several other operation sequences which also occurred in high frequency. The add-multiply sequence, although it did not occur very often naturally in the code, did exist in high frequency after using code motions to expose it. The majority of these sequences were found in loops which had been pipelined. There was often an addition in one iteration of the loop, whose result was then used by a multiply in the next iteration of the loop. This data flow was not detected by the straight-forward analysis, but was uncovered by using loop pipelining.

6.2 Individual benchmark results

The next set of results being presented, shows the number and frequencies of the sequences detected for each benchmark individually. The results for each benchmark are listed separately within each graph, showing the individual sequences detected for that benchmark. Sequences whose dynamic frequency was less than 5% were not reported (again, results from length three and five sequences are omitted to save space).

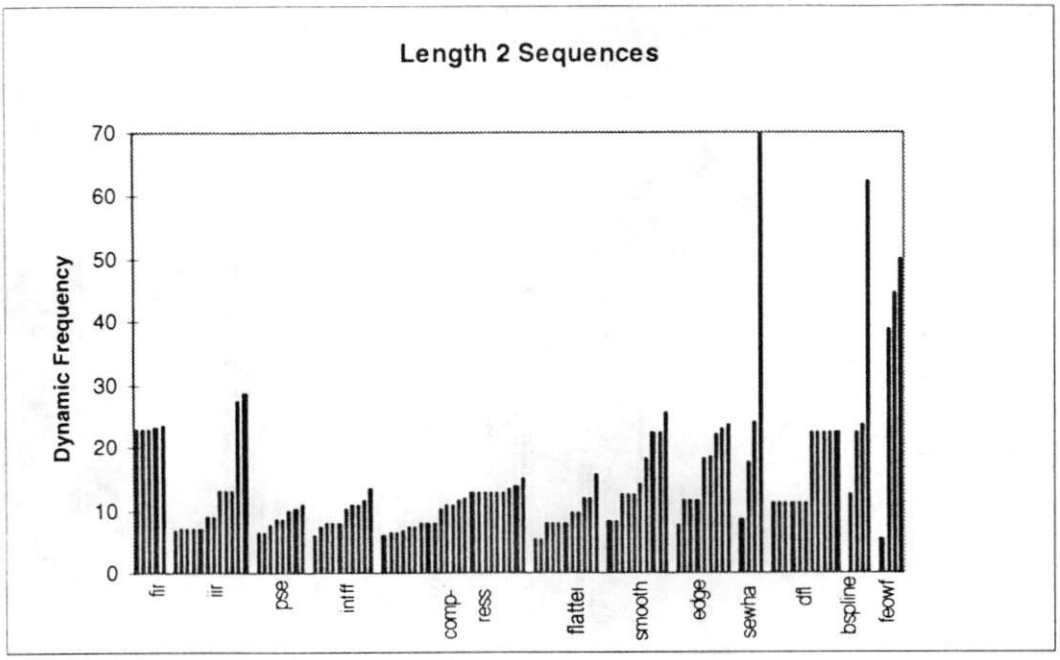

Figure 7 : Detected chainable sequences of length 2

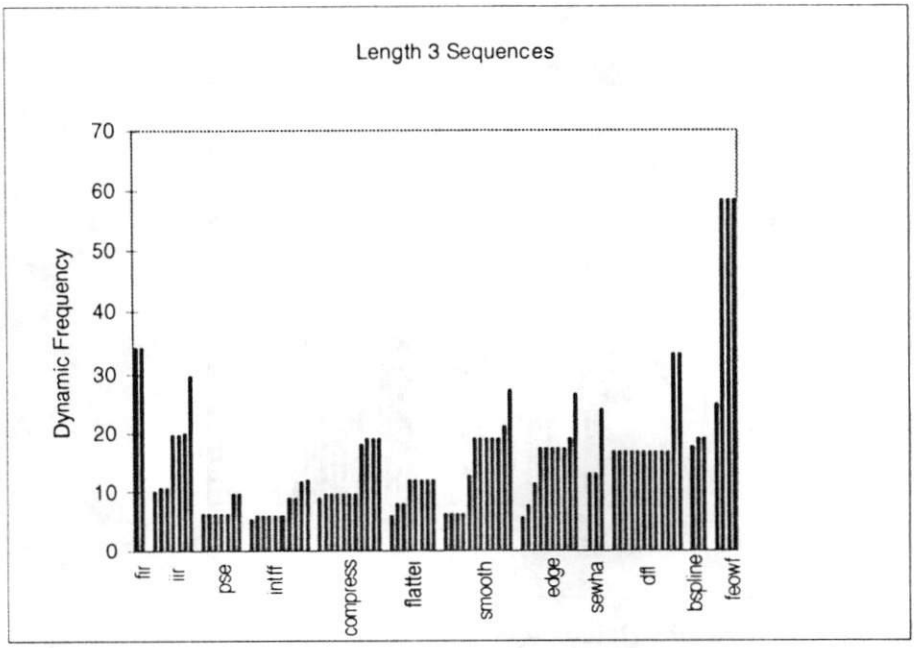

Figure 8 : Detected chainable sequences of length 3

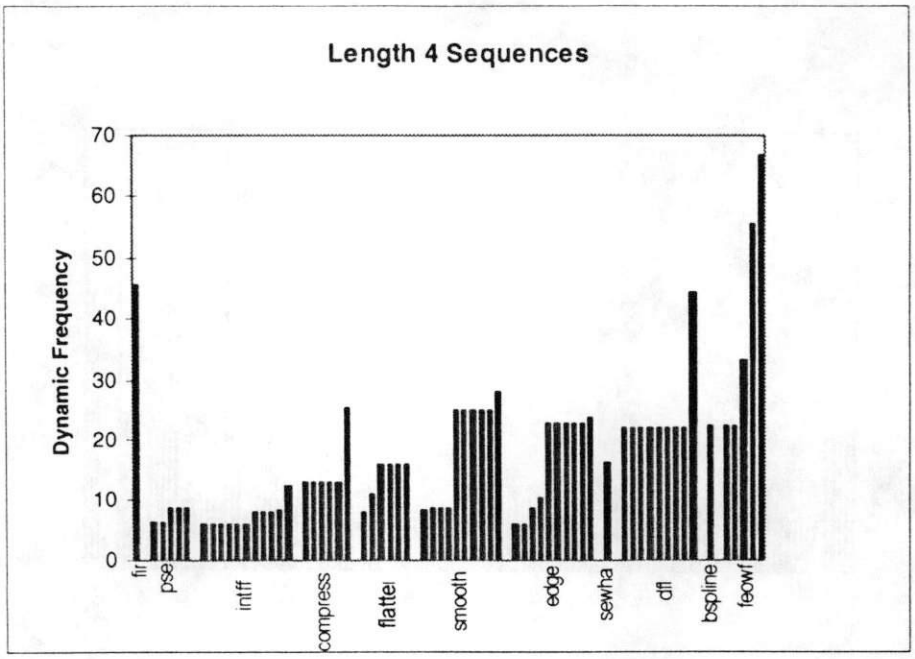

Figure 9 : Detected chainable sequences of length 4

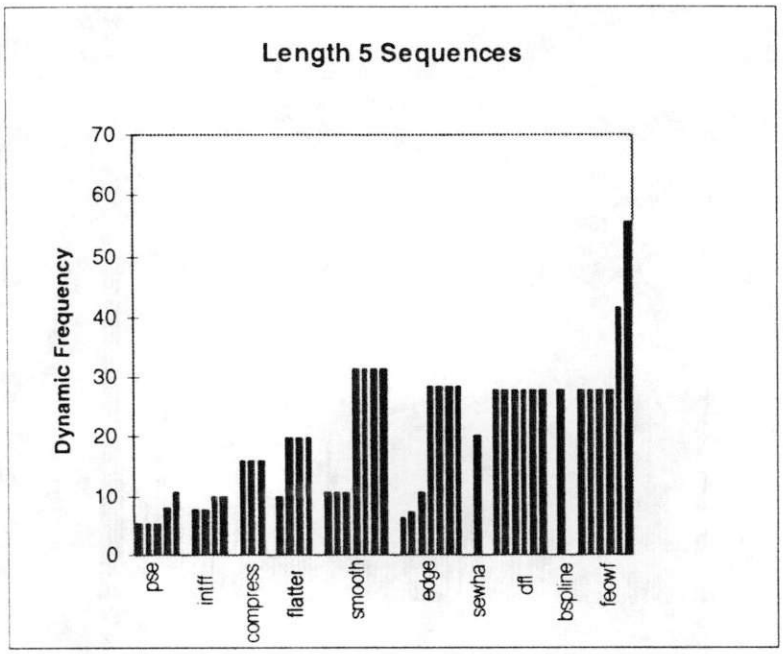

Figure 10 : Detected chainable sequences of length 5

6.3 Sequence examples

The following tables give examples of some of the actual sequences that were detected along with their corresponding frequencies using the three levels of optimization in the compiler.

Operation Sequence (with result to operand data flow)	Dynamic Frequency (No Optimization)	Dynamic Frequency (Pipelined)	Dynamic Frequency (Pipelined + Renamed)
add-add	0%	13.91%	13.91%
add-add-shift	0%	20.87%	20.87%
shift-mul-add	0%	0%	12.52%
add-add-shift-add	0%	27.83%	19.48%
add-shift-add-sub-add	0%	0%	10.43%

Table 3 - Operation sequences detected in smooth

Operation Sequence (with result to operand data flow)	Dynamic Frequency (No Optimization)	Dynamic Frequency (Pipelined)	Dynamic Frequency (Pipelined + Renamed)
add-add	2.11%	14.81%	18.61%
add-add-shift	0%	19.04%	19.04%
shift-mul-add	0%	0%	11.42%
add-add-shift-add	0%	23.69%	15.23%

Table 4 - Operation sequences detected in edge

Operation Sequence (with result to operand data flow)	Dynamic Frequency (No Optimization)	Dynamic Frequency (Pipelined)	Dynamic Frequency (Pipelined + Renamed)
add-mul	0%	63.99%	69.53%
mul-add-add	0%	0%	12.99%
add-mul-add	0%	12%	13.04%

Table 5 - Operation sequences detected in sewha

Operation Sequence (with result to operand data flow)	Dynamic Frequency (No Optimization)	Dynamic Frequency (Pipelined)	Dynamic Frequency (Pipelined + Renamed)
mul-add	0%	11.09%	0%
mul-mul	0%	11.09%	0%
mul-mul-add	0%	16.64%	0%

Table 6 - Operation sequences detected in dft

Operation Sequence (with result to operand data flow)	Dynamic Frequency (No Optimization)	Dynamic Frequency (Pipelined)	Dynamic Frequency (Pipelined + Renamed)
mul-add	0%	12.5%	23.48%
add-mul	0%	62.48%	0.23%
mul-add-add	0%	0%	17.58%
add-mul-add	0%	18.75%	0.07%

Table 7 - Operation sequences detected in bspline

Operation Sequence (with result to operand data flow)	Dynamic Frequency (No Optimization)	Dynamic Frequency (Pipelined)	Dynamic Frequency (Pipelined + Renamed)
add-mul	27.02%	38.88%	38.91%
add-add-add	0%	25.0%	25.0%
mul-add-add	24.32%	58.33%	58.33%
add-add-mul	8.11%	58.33%	58.33%
add-mul-add	40.54%	58.33%	58.33%
mul-add-add-add	0%	22.22%	22.22%
add-add-add-mul	0%	33.33%	22.23%
add-add-mul-add	10.81%	55.55%	44.46%
mul-add-add-mul	0%	22.22%	22.22%
add-mul-add-add	10.81%	66.66%	66.66%
add-add-add-mul-add	0%	41.66%	27.79%
add-add-mul-add-add	0%	55.55%	27.79%
add-mul-add-add-add	0%	27.77%	27.79%
mul-add-add-mul-add	0%	27.77%	27.79%
add-mul-add-add-mul	0%	27.77%	27.79%
mul-add-add-add-mul	0%	27.77%	27.79%

Table 8 - Operation sequences detected in feowf

7 Sequence Coverage

Another way to measure the effectiveness of sequence detection, is to determine the coverage obtained by implementing a set of chained operation sequences. The highest coverage using the fewest number of operation sequences will be the best solution, both in terms of area and speed. In order to compare the coverage that could be obtained both with and without compiler optimizations, we used the sequence detection analyzer tool to iteratively uncover the sequences with the highest frequency. Once the sequence with the highest frequency was found for a given benchmark, the sequence detection analyzer tool was run again, this time ignoring any occurrences of the high-frequency sequence already found. This process continued iteratively until no sequences of any significant percentage

were left to uncover. The analysis was performed both with and without the parallelizing optimizations in order to assess the impact of the optimizations on sequence detection.

We found that by using feedback from our optimizing compiler, we were able to achieve higher coverage rates with fewer operation sequences than could have been achieved without the compiler input. The results of these analyses (on a subset of the benchmarks) are presented in Table 9.

8 Conclusion

We have presented a framework for providing feedback from the compiler to the design of ASIPs. This framework entails relating individual compilation techniques to hardware optimizations and extensions in the development of ASIPs. By using the compiler in the design process, the ASIP designer is presented with a wider range of possibilities, and the assurance that the hardware extensions chosen will be used effectively by the compiler. We also presented results of an initial study on using parallelizing compiler techniques to detect operation sequences suitable for implementation as chained instructions. This study showed that the use of the compiler in assessing the hardware needs of an application can be particularly effective.

We are currently exploring the relationship between an ASIP and its compiler by looking at additional compiler optimizations and how they can impact the choice of hardware extensions for ASIP designs. In particular, we are interested in providing feedback on the use of multiple-issue instruction-set architectures by characterizing the instruction level parallelism of an application suite using compiler optimizations.

Benchmark	Optimization	Sequences	Frequency	Coverage
Sewha	yes	add-multiply	69.53%	91.31%
		add-add-add	13.09%	
		add-compare	8.69%	
	no	add-add	23.99%	31.99%
		add-compare	8.69%	
Feowf	yes	add-add-multiply	58.33%	97.15%
		add-add	27.70%	
		add-multiply	5.56%	
		add-compare	5.56%	
	no	multiply-add	43.24%	75.66%
		add-add	27.02%	
		add-compare	5.40%	
Bspline	yes	add-multiply	62.48%	97.76%
		add-add	11.76%	
		add-compare	11.76%	
		multiply-add	11.76%	
	no	add-add	22.22%	33.33%
		add-compare	11.11%	
Edge	yes	add-shift-add	24.78%	85.35%
		add-compare	17.46%	
		load-multiply-add	17.46%	
		add-shift	12.94%	
		add-load	8.40%	
		shift-add	4.31%	
	no	add-shift-add	24.68%	66.39%
		add-compare	18.13%	
		load-multiply-add	17.14%	
		shift-add-subtract	4.44%	
Iir	yes	fload-fmultiply	23.22%	60.6%
		add-load	12.77%	
		fmul-fsub-fstore	10.45%	
		add-compare	9.52%	
		fload-fadd	4.64%	
	no	fload-fmultiply	28.50%	38.59%
		add-compare	6.80%	

Table 9 - Sequence Coverage

References

[1] I. -J Huang and A. M. Despain, High level synthesis of pipelined instruction set processors and back-end compilers. In *International Workshop on Computer-Aided Codesign*, June 1992.

[2] Pradeep K. Dubey, George B. Adams, III, and Michael J. Flynn. Instruction Window Size Trade-Offs and Characterization of Program Parallelism. *IEEE Transactions on Computers, Vol. 43, No. 4*, pages 431-442, April 1994.

[3] Alauddin Alomary et al. PEAS-I: A hardware/software co-design system for ASIPs. In *2nd International Conference on Software/Hardware Co-design*, pages 2-7, October 1993.

[4] Bruce K. Holmer and Barry M. Pangrle. Hardware/software codesign using automated instruction set design and processor synthesis. In *International Workshop on Computer-Aided Codesign*, October 1993.

[5] Bruce K. Holmer and Alvin M. Despain. Viewing instruction set design as an optimization problem. In *Proceedings of the 24th Symposium on Microarchitecture,* November 1991.

[6] Clifford Liem, Trevor May, Pierre Paulin. Instruction-Set Matching and Selection for DSP and ASIP Code Generation. *Proceedings of EDAC 1994*, pages 31-37.

[7] J. Van Praet, G. Goosens, D. Lanneer, and H. De Man. Instruction Set Definition and Instruction Selection for ASIPs. *High Level Synthesis Symposium*, 1994.

[8] Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[9] Texas Instruments, "TMS320C5x User's Guide", Texas Instruments, 1992.

[10] Tomlinson G. Rauscher and Ashok K. Agrawala. Dynamic problem-oriented redefinition of computer architecture via microprogramming. *IEEE Transactions on Computers,* C-27:1006-1014, November 1978.

[11] Åmund Lunde. Empirical evaluation of some features of instruction set processor architectures. *Communications of the ACM,* 20(3):143-153, March 1977.

[12] Richard E. Sweet and James G. Sandman, Jr. Empirical analysis of the mesa instruction set. In *Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 158-166, October 1982.

[13] Gene McDaniel. An analysis of a mesa instruction set using dynamic instruction frequencies. In *Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167-176, October 1982.

[14] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *International Conference on Parallel Processing*, July 1985.

[15] Paul M. Embree and Bruce Kimble, *C Language Algorithms for Digital Signal Processing*, Prentice Hall, 1991.