Published in *Proceedings of the 6th International Conference on Signal Processing Applications and Technology*, pp. I-860–I-865, October 1995.

# Machine-Independent Compiler Optimizations for the UofT DSP Architecture

*Sanjay M. Pujare,* Corinna G. Lee,** and Paul Chow***

* STD/CLO California Language Lab
Hewlett-Packard Company
11000 Wolfe Road, Mailstop 42UD
Cupertino, CA, U.S.A. 95014-9804

** Department of Electrical and Computer Engineering
University of Toronto
10 Kings' College Road
Toronto, Ontario, Canada M5S 1A4

## ABSTRACT

There are two kinds of compiler optimizations: processor-specific optimizations and machine-independent optimizations. For specialized DSP architectures, processor-specific optimizations play an important role because these optimizations seek to exploit the special hardware features of the target processor to maximize performance. Using these processor-specific optimizations alone in a compiler, however, requires that programmers write their programs in a specific style. Fortunately, incorporating certain machine-independent optimizations in the compiler allows programmers to realize the benefits of processor-specific optimizations without imposing overly restrictive rules on programming style.

This paper describes our implementation of some machine-independent compiler optimizations, collectively called the INDVAR optimizations, for the UofT DSP architecture. In addition we present performance results that demonstrate the effectiveness of the INDVAR optimizations. For our performance evaluation, we used an existing DSP benchmark suite that was written in a specific DSP style dictated by our earlier compiler and wrote two new versions (*C-style* and *C-style with SWP*) that use a more natural style of C programming. Our performance data shows that our machine-independent optimizations are reasonably effective: for seven out of twelve kernel benchmarks and for five out of six application benchmarks, the performance of the *C-style with SWP* version is equal to or better than the performance of the original version, when both versions are compiled by the augmented compiler. The few performance exceptions demonstrate the importance of having the compiler implement both machine-independent and processor-specific optimizations rather than having the programmer share some of the responsibility.

## 1. INTRODUCTION

DSP processors are specially designed or tuned to enable fast and efficient execution of DSP applications. A variety of requirements that characterize a DSP architecture are as follows [1]: MAC instructions, low overhead looping, increased memory bandwidth, addressing support for modulo and bit-reversed addressing, hardwired control circuitry (as opposed to micro-

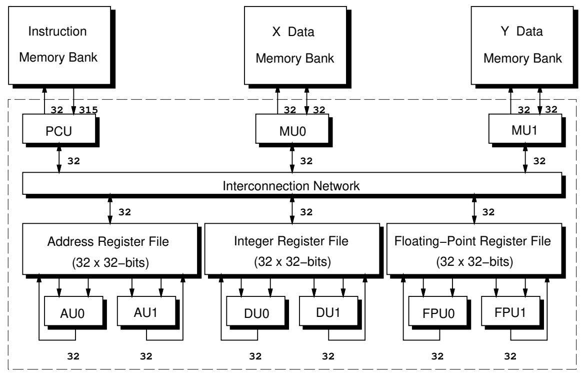


Figure 1: VLIW Model Architecture

programmed), and multiple functional units to exploit DSP parallelism. However it is not enough to have only a specialized architecture for DSP applications. For any machine architecture feature to be useful, it has become necessary to use an optimizing compiler that can exploit these features in the code that it generates. As optimizing compilers become more effective, programmers can become less concerned about the details of the underlying processor architecture and can employ higher-level, more succinct, and more intuitive programming constructs and algorithm optimizations.

Our research group at the University of Toronto has been exploring these issues about processors and HLL compilers for DSP applications [1][2][5]. As part of our prior research, we have designed a general-purpose DSP processor that is a Very Long Instruction Word (VLIW) architecture. A VLIW architecture uses multiple functional units and has been used by others for DSP applications [7]. Figure 1 shows the U of T DSP architecture with its nine functional units. We are currently at the exploratory design stage and use an instruction-set simulator to execute our benchmark programs and to generate performance statistics rather than use actual hardware. Although an implementation of our architecture is too expensive for DSP applications, we have chosen it because it is a good target architecture for an optimizing compiler.

Future research includes exploring mechanisms for implementing a VLIW architecture in a less costly manner.

In addition to designing the VLIW architecture, we also developed an optimizing HLL compiler that can effectively utilize the DSP-specific features of the target processor. This system performed almost three times better than the best commercial system of that time [1]. The HLL C language compiler consisted of a front-end ported from the GCC C compiler and a post-optimizer phase that performed processor-specific optimizations such as instruction scheduling for the target VLIW processor.

Although the processor-specific optimizations demonstrated their benefits, the compiler failed to apply certain machine-independent optimizing transformations. This resulted in some constraints being imposed on the application programmers. These constraints came to be called the "DSP-style" of writing code. DSP-style programming typically requires the use of pointers instead of array elements and careful coding of loops to make them more efficient. As an example, Figure 2 shows a code fragment written in a more natural C-style while Figure 3 shows a semantically equivalent fragment written in the DSP style. Clearly, the C-style fragment is more readable; however, to use the processor-specific features of the DSP processor and to achieve good performance, our GCC-based compiler requires that the source program be written in the DSP style.

As part of our research project, we decided to

```
for (i = 0; i < N; i++) {
  for (j = 0; j < i; j++) {
    a[i][j] = sqrt(i) + j * 2;
    b[j] = 5 * j;
  }
}
```

Figure 2: C-style code fragment

```
pa1 = a;
for (i = 0; i < N; i++) {
  t1 = sqrt(i);
  pa = pa1;
  j1 = 0;
  j2 = 0;
  pb = b;
  for (j = 0; j < i; j++) {
      *pa++ = t1 + j1;
      *pb++ = j2;
      j1 = j1 + 2;
      j2 = j2 + 5;
   }
   pa1 = pa1 + M;
}
```

Figure 3: DSP-style code fragment

investigate how to remove the programming constraints of the DSP style without sacrificing performance. Our primary objective was to develop a compiler that would not impose the constraints of the 'DSP-style' on the users, as the GCC-based compiler did, but would deliver similar performance. In addition to the programming constraints, our GCC-based compiler also lacked certain features, such as a dependence analysis phase and an ability to annotate the output code, that we need for other aspects of our research on the DSP architecture and compiler. In view of the limitations of the GCC-based compiler, we decided to retarget a new compiler for our DSP architecture and chose the SUIF (Stanford University Intermediate Format) compiler.

In this paper, we highlight aspects of our new SUIF-based compiler and present performance data demonstrating its ability to generate good quality code for programs written in a natural C style. Section 2 describes the optimizations that we implemented while Section 3 presents the performance results. The work described in this paper was part of a larger effort to incorporate the SUIF compiler into our software tools. This paper describes the work that is most relevant to eliminating the DSP-style programming. The interested reader is referred to [8] for details on other aspects of the retargeting effort.

## 2. MACHINE-INDEPENDENT OPTIMIZATIONS

Ideally, we wanted the new SUIF-based compiler to generate code for programs written in the natural C style that would be about as fast as the code generated for the same programs written in the DSP style. As we will see next, this requires the use of some powerful scalar optimizations in the SUIF compiler.

### 2.1. Need for Scalar Optimizations

DSP-style programming clearly points to a lack of some scalar optimizations in the GCC-based compiler. Optimizations missing in the GCC-based compiler are:

i) induction variable creation (IVC): This involves replacing linear expressions of the loop index variable with an induction variable (see Section 10.7 in [9]). This optimization would allow the use of array notation inside loops as well as the use of multi-dimensional arrays. For example, for the code fragment in Figure 2, the addresses of the array elements are linear expressions of the loop indices, and hence these expressions can be converted into induction variables that will result in code comparable to what is produced for the fragment in Figure 3. For instance, the array access `b[j]` produces the linear address expression `b+4*j` which, in turn, corresponds to the induction variable `pb` in Figure 3.

ii) loop-invariant code motion (LICM): A loop-invariant expression computes the same value in all iterations of the loop and can be moved out of the loop without changing the semantics of the loop. Moving a loop-invariant expression out of a loop improves performance because the expression will be executed only once before control enters the loop. As an example, in Figure 2 the sub-expression `sqrt(i)` is invariant with respect to the inner `for` loop, since `i` itself is invariant in the inner `for` loop. By using LICM, the computation for `sqrt(i)` can be moved out of the inner loop, the result stored in a temporary variable, and the variable used inside the `for` loop as was done with the variable `t1` in Figure 3.

iii) common sub-expression elimination: While performing IVC and LICM if we replace identical linear or loop-invariant expressions with the same induction/temporary variable we achieve common sub-expression elimination for such expressions.

These optimizations can be implemented using dataflow analysis (DFA), a powerful and general technique for expressing scalar optimizations [9]. While the theory of DFA is well-established, the use of it with specialized DSP architectures has not been investigated in detail. As we demonstrate shortly, DFA-based optimizations sometimes work at cross-purposes with processor-specific optimizations if incorrectly used.

### 2.2. INDVAR Optimizations

The SUIF compiler is a flexible framework for advanced compiler research [10], specifically in the area of optimizations for high performance platforms such as our DSP architecture. SUIF includes a scalar optimizer called ***oynk.*** However, when we attempted to use ***oynk***, it did not work as well as we had expected and did not cover all the optimization opportunities in our benchmarks. As a result, we decided to develop our own optimizer to obtain exactly those optimizations that we wanted. We call this phase INDVAR because of the main optimization it does: induction variable creation. The requirements and the resulting design decisions that guided the development of INDVAR are described below. Detailed descriptions of the algorithms and SUIF can be found in [8].

The most important optimization INDVAR performs is IVC. It detects maximal linear expressions inside a `for` loop and replaces them with induction variables. While traversing an expression tree from the leaves to the root, INDVAR takes note of loop constants, index variables and instructions such as `add`, `sub`, `neg`, `cpy` and `mul` that preserve linearity.

For the IVC optimization to be effective for nested loops, INDVAR also detects loop-invariant expressions, and performs code-motion for these expressions. As an example, the array access `a[i][j]` in Figure 2 involves the outer loop index variable `i` which is a constant with respect to the inner loop. Correct LICM will enable IVC for the address expression for `a[i][j]`. For nested loops, INDVAR starts from the inner-most loop, and goes towards the outer loop for the IVC and LICM optimizations.

INDVAR detects common sub-expressions, which are optimized by IVC or LICM, inside a `for` loop, so that it can use the same induction variable, thus giving us common sub-expression elimination for such expressions. This can be achieved by maintaining a linked list of linear expressions for which an induction variable has been created. If an instance of an expression from this linked list is found later in the loop, that instance can be eliminated with the use of the associated induction variable.

INDVAR tries to reuse the symbols that it creates for the induction variables. This was necessary because the retargeted SUIF compiler does not have a register allocator for our target DSP architecture. This means that if there are two `for` loops, where neither nests the other, they should share an induction variable to minimize the number of local variables in the function. This is achieved by maintaining a linked list of all induction variable symbols already created for the function. Every time a new induction variable is required, INDVAR attempts to reuse an existing induction variable symbol which is no longer live.

## 3. PERFORMANCE

This section summarizes the results of the performance testing we did on the code generated by the new retargeted SUIF compiler. Due to space constraints, we present only the main results; more performance data as well as a more detailed analysis are given in [8].

### 3.1. Experimental Framework

To gather the performance data, we compiled various programs using our GCC-based and SUIF-based compilers and executed the generated code on an instruction-set simulator we developed for our VLIW architecture.

The programs we used consist of 18 benchmarks divided into 12 kernels and 6 applications. The kernels are based on algorithms commonly used in DSP applications and the applications are from the areas of speech and image processing. These were obtained from previous work done on the DSP architecture[1][2] and were written in the DSP style to get better performance out of the GCC-based compiler. We call this original version of the benchmark *DSP-style with SWP* where SWP is an abbreviation for software pipelining. Software pipelining is a form of instruction scheduling where a loop is rearranged so as to exploit parallelism across loop iterations. It is to be noted that the original benchmarks were hand-coded to achieve some amount of software pipelining in their loops.

We created two more versions of these benchmarks: the first version is called *C-style* and the second one is called *C-style with SWP*. Both the *C-style* and *C-style with SWP* versions are semantically equivalent to the *DSP-style with SWP* version. In our performance testing, we ensured that, given the same input, these new versions of the benchmarks produced the same output as the *DSP-style with SWP* version. The *C-style* benchmarks do not have the constraints of the DSP-style and look more like natural C language code, whereas the *C-style with SWP* version, as the name suggests, is the software pipelined version of the *C-style* benchmarks. By using these different versions of our benchmark suite, we were able to quantify how well the INDVAR optimizations perform by comparing the performance of the new versions of the benchmarks to the that of the original version.

For each benchmark, we established a Baseline Execution Time which is the number of cycles needed to execute the code generated by the original GCC-based compiler for the original *DSP-style with SWP* version of that benchmark. In our testing, any absolute execution time number we obtained for any version of the benchmark was converted to a Relative Execution Time by dividing the absolute execution time by the Baseline Execution Time for that benchmark. Thus, the ideal Relative Execution Time of 1.0 for a particular situation signifies that we were able to match the Baseline Execution Time in that situation.

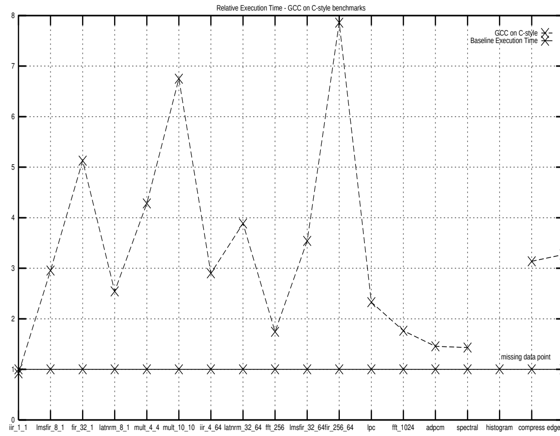

Figure 4: RET - GCC on *C-style* benchmarks

We have used the Relative Execution Time (RET) in all our graphs here. Also, in all these graphs, different symbols represent the RET associated with code generated by different compilers while different line styles correspond to different versions of the benchmark suite. For example, RETs associated with the code generated by the GCC-based compiler are indicated with cross symbols (✕), whereas RETs associated with the SUIF compiler are indicated with various rectangular symbols such as the diamond symbol (◊). Solid lines represent RETs corresponding to the *DSP-style with SWP* version, dashed lines indicate RETs associated with the *C-style* version, and the dotted lines indicate RETs for the *C-style with SWP* version.

### 3.2. Results and Analysis

Figure 4 shows the Relative Execution Time for all benchmarks when the original GCC-based compiler is used on the new *C-style* version. We can see that the relative execution time varies from the best of almost 1.0 for **iir_1_1**, to the worst of almost 8.0 for **fir_256_64**. Note that the *C-style* benchmarks were compiled with the GCC-based compiler version 1.39 and the GCC optimization option **-O** turned on. Also note that there is a missing data point for the benchmark **histogram** because the GCC-based compiler

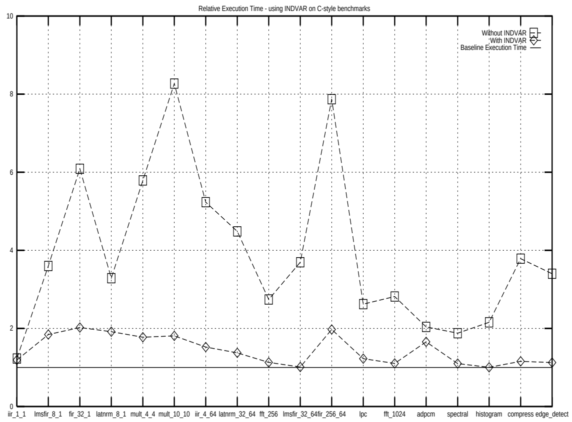


Figure 5: RET - SUIF on *C-style* benchmarks

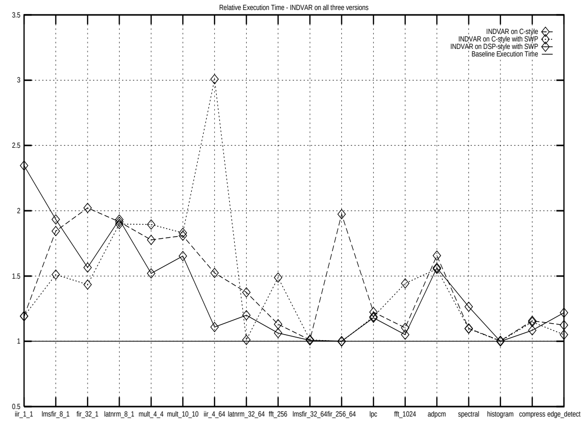


Figure 6: RET - SUIF on all 3 versions

was unable to compile that benchmark successfully. It is this poor performance of the GCC-generated code for the *C-style* benchmarks that motivated us to work on a new compiler such as SUIF. We wanted the new compiler to generate efficient code for the *C-style* benchmarks, thereby eliminating the constraints of DSP-style programming.

Figure 5 shows the Relative Execution Time when the SUIF compiler is used with and without INDVAR to compile *C-style* version benchmarks. The impact of INDVAR on the execution time of the *C-style* can be considered as truly remarkable. We can see that the relative execution time of the code optimized by INDVAR ranges from 1.0 to about 2.1. We can see up to four times improvement in the execution time of the code due to the optimizations performed by INDVAR. Note that the induction variable creation optimization of INDVAR enables the hardware do-loop optimization by the post-optimizer. In short, INDVAR serves the same purpose as the 'DSP-style'. The cumulative effect of induction variable creation and the hardware do-loop generation produces a dramatic improvement in the performance of the SUIF-generated code for the *C-style* benchmarks.

Notice that we haven't yet reached our ideal Relative Execution Time of 1.0 with the exception of a few benchmarks such as **lmsfir_32_64** and **histogram**. One of the main reasons for this is the fact that the *DSP-style with SWP* version benchmarks have been manually software pipelined. To eliminate the performance gap caused by software-pipelining, we decided to modify the *C-style* benchmarks to create another version of benchmarks called *C-style with SWP* which are identical to the *C-style* ones except that their loops are software pipelined in the same fashion as was done for the *DSP-style with SWP* version.

Figure 6 shows the Relative Execution Time of all 3 versions of the benchmarks when compiled with the SUIF+INDVAR compiler under identical conditions. What we see here can be aptly described as mixed results. By using the *C-style with SWP* version, we expected to get better performance or at least no worse performance than with the *C-style* version. This is true for most of the benchmarks with six achieving more than a 6% improvement in performance (maximum speedup of about 2.0 in **fir_256_64**) and eight benchmarks showing little or no difference in performance. Unfortunately, using the *C-style with SWP* version degrades performance for the remaining four programs by more than 5% with the worst being a 100% performance degradation for **iir_4_64**.

We have traced the performance degradation to a complex interaction between the induction variable creation, hardware do-loop and manual software-pipelining optimizations. Manual software pipelining involves rearranging loops as explained earlier. In most cases this requires using additional variables in those loops, which again can result in additional linear or constant expressions in those loops. IVC converts these into induction variables. These extra variables can cause register spilling which we could not tolerate because of other constraints in our DSP architecture. Hence we put a limit on the number of induction variables added by the IVC optimization. When this limit is reached, INDVAR stops further optimization. This prevents one of our processor-specific optimizations, hardware do loop generation, from taking place. The performance degradation we see is the cumulative effect of all these factors.

With the exception of the programs that suffer from the unfortunate interaction between the IVC, hardware do-loop, and manual software-pipelining optimizations, our INDVAR optimizations are successful at achieving performance for C-style programming that is comparable to the performance of DSP-style programming. From Figure 6, we can see that, for seven of the kernel benchmarks and five of the applications, the performance of the INDVAR-generated code for the *C-style with SWP* version is equal to or better than the performance of the INDVAR-generated code for the *DSP-style with SWP* version.

## 4. SUMMARY AND CONCLUSIONS

By using the INDVAR optimizations, we have been able to match, with the exception of a few programs, the performance of the *C-style with SWP* version benchmarks with that of the *DSP-style with SWP* version when both are compiled with the SUIF compiler. In other words, we have been reasonably successful at implementing machine-independent optimizations that enable the use of processor-specific optimizations in the compiler with minimal help from the programmer.

The optimization remaining for the programmer to do is software pipelining, a processor-specific optimization. However as the few performance exceptions indicate, doing so can sometimes be counter-productive. In these cases, applying a processor-specific optimization at the source level limited the application of the machine-independent optimization in the compiler which in turn prevented the compiler from using another processor-specific optimization. To avoid this rather complex and performance-degrading sequence of events, the compiler should perform all the optimizations including software pipelining.

Although the INDVAR optimizations result in good performance, we have not reached our ideal goal of a Relative Execution Time of 1.0 for all of our benchmarks using our retargeted SUIF compiler. To reach this goal would require implementing a software-pipelining optimization in the compiler and further investigation to ensure effectiveness of existing optimizations. We believe that these solutions are orthogonal to our optimizations and when these are implemented, we will most likely be able to bridge the performance gap between the SUIF-generated code for the *C-style* version and the GCC-generated code for the *DSP-style with SWP* version of the benchmarks.

## 5. ACKNOWLEDGEMENTS

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

## 6. REFERENCES

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

[2] Mazen A.R. Saghir. *Architectural and Compiler Support for DSP Applications*. M.A.Sc. Thesis, University of Toronto, 1993. Selected sections published as [3] and [4].

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

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

[5] Wen-Yen Lin. *An Optimizing Compiler for Texas Instruments' TMS320C25*. M.A.Sc. Thesis, University of Toronto, 1995. An abridged version published as [6].

[6] Wen-Yen Lin, Corinna G. Lee, and Paul Chow. An Optimizing Compiler for Texas Instruments' TMS320C25. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages I-689–I-694, October 1994.

[7] Brian Case. Phillips Hopes to Displace DSPs with VLIW. *Microprocessor Report*, pages 12-15, December 5, 1994.

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

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

[10] SUIF Compiler System. Available from http://Suif.Stanford.edu/suif/suif.html.