

Modifying a VLIW Compiler Framework to Implement an Optimizing Compiler for a Fixed Point DSP

Subramanian Rajagopalan and Sharad Malik
Princeton University
{sr,sharad}@ee.princeton.edu

Sreeranga P. Rajan
Fujitsu Laboratories of America
{sree}@fla.fujitsu.com

Guido Araujo and Sandro Rigo
Instituto de Computacao, UNICAMP
{guido,srigo}@ic.unicamp.br

Abstract

A common design methodology for embedded DSP systems is the integration of one or more digital signal processors (DSPs), program memory, and ASIC circuitry onto a single IC. Program memory size being limited in embedded DSP systems, the criterion for optimality is that the embedded software must be very dense. In order to improve the instruction cycle time, it's becoming more common to design DSPs with multiple functional units. Recently, DSPs have been designed to include support for SIMD instructions. Such a design with support for instruction level parallelism (ILP) poses significant challenges to developing optimizing DSP compilers, because existing DSP compilers provide little for exploiting ILP. In this work we show how we can modify a retargetable VLIW compiler infrastructure to efficiently develop an optimizing compiler for DSPs with ILP. Our thesis is that using a VLIW compiler framework is a better approach to develop an optimizing compiler for a DSP with ILP/SIMD than a DSP-specific compiler that is enhanced to exploit ILP/SIMD. We use the IMPACT VLIW framework to develop a compiler for the second generation *Fujitsu Hiperion*, a fixed-point DSP.

1 Introduction

One of the most important requirements for embedded system software is that it has to be sufficiently dense so as to fit within the limited quantity of silicon area dedicated to program memory. Obtaining sufficiently dense software has remained a challenge for optimizing compilers for embedded DSP systems.

An *optimizing compiler* features two sets of code optimization modules, in addition to the components that were described above: *machine independent* optimizations apply various optimizing transformations to the front-end-generated IR; *machine dependent* or *post-pass* optimizations apply various optimizing transformations to the generated assembly code.

Unfortunately, it is common knowledge that existing

compilers for embedded DSPs are generally unable to generate assembly code that is sufficiently dense. It is typical for naive DSP compilers to generate assembly code whose size is more than 5 times greater than the size of the corresponding hand-written assembly code. There are two reasons for this: **first**, most existing embedded DSP compilers make use of traditional machine-independent optimization techniques, in which the primary metric is performance, rather than code density. In some cases such as loop unrolling these two metrics are not closely associated. **Second**, current embedded DSP compilers fail to provide adequate support for the specialized architectural features of DSPs via machine-dependent code optimizations. These features not only allow for the fast execution of common DSP operations, but also allow for the generation of dense assembly code that specifies these operations.

In order to guarantee that all code density and performance requirements are safely met, system designers typically hand-program the embedded software in assembly, which is a very time-consuming, tedious, and error-prone task. This situation unfortunately in turn leads to architecture designs which do not take the compiler into considerations. In order to increase productivity, compilers must be developed that are capable of generating high-quality code for embedded DSPs.

In an earlier work [6], using the SPAM compiler framework [9], we had obtained a high-quality optimizing compiler for *Fujitsu Elixir*, a fixed-point DSP that is primarily used in cellular telephones. Subsequently, we modified the optimizing compiler for the first generation *Fujitsu Hiperion* architecture and obtained results competing with a commercially available optimizing compiler for DSPs. The second generation *Fujitsu Hiperion* which is a replication of the first generation *Hiperion* allows a number of instructions to be executed in parallel. The support for instruction level parallelism (ILP) poses significant challenges to existing optimizing DSP compilers, including the compiler we developed for the first generation *Hiperion* [6]. The reason is that existing DSP compiler frameworks do not provide sufficient support for exploit-

ing ILP, and furthermore it is difficult to enhance them to exploit SIMD. Whereas a VLIW compiler framework has in-built ILP-based optimizations, and can be extended to exploit SIMD.

In this work we modify the IMPACT VLIW [4] compiler framework to develop an optimizing compiler that exploits ILP. The DSP we have chosen for compiler development is the second generation Hiperion. Since the DSP have multiple specialized functional units with restricted ILP and are fully statically scheduled, we use VLIW compilation techniques to generate assembly code. Since VLIW compilers can be made highly re-targetable using machine description languages, we believe that this exercise will help in developing a synergistic architecture-compiler design methodology for DSPs.

The organization of the rest of the paper is as follows. The second generation Hiperion architecture is described in Section 2. The IMPACT compiler framework is described in Section 3. Section 4 describes the development of optimizing compiler for Hiperion using IMPACT. Results of experiments are provided in Section 5. Conclusions are summarized in Section 6.

2 Hiperion Architecture

The Fujitsu Hiperion DSP is a fixed point DSP. Its datapath consists of dual memory banks, an arithmetic and logic unit (ALU) comprising of a shift unit, add unit and a multiply unit and an address generation unit. The two memory banks allow two memory accesses to occur in parallel. In addition to this, restricted Instruction Level Parallelism (ILP) also exists between the ALU and memory units. For example, a MUL (Multiply) and two LOAD operations can be executed in parallel only if the destination registers of the two LOAD operations are the same as the source registers of the MUL operation.

Most of the ILP constraints in Hiperion such as the previous example are because of the way in which the instructions are encoded. All instructions are encoded in 16 bits irrespective of whether an instruction has one, two or three operations.

Although there are enough functional units to perform 2 LOAD operations, an ADD operation and a MUL operation, the Hiperion can perform neither an ADD and MUL in parallel nor an ADD and two LOAD operations in parallel.

We call ILP as Irregular ILP [5] where the combinations of operations that can be issued in parallel are decided not due to physical resource constraints alone but also due to pressure to reduce code size, area or power.

3 IMPACT Compiler

In this section, we give a brief overview of the IMPACT [4] compiler. The IMPACT compiler is a re-targetable compiler for multiple instruction issue processors. The front-end of IMPACT produces as output an Intermediate Representation (IR) called *Lcode* which is then used by the

code generator. Code generation for a given target architecture is performed in three phases, namely,

Phase 1 Opcode Selection for the Lcode IR is performed in this phase. Complex operations such as the Multiply Accumulate (MAC) are generated using peephole optimizations.

Phase 2 In this phase, Machine dependent optimizations, Pre-pass scheduling, Register Allocation [8], Machine dependent optimizations and Final Scheduling are performed.

Phase 3 Assembly Code is generated from the annotated and scheduled Lcode in this phase.

The target processor architecture specification in IMPACT comprises of three parts:

1. The processor details such as function frame layout, access of local variables, function return value register, machine specific registers etc are described in the Machine specification or Mspec section.
2. Machine description or *Mdes* [2, 7]. In this description, the set of all operations supported by the processor architecture, the physical resources such as functional units used by the operations and type of each operand in an operation are specified.
3. Register files corresponding to the supported data types are described in the Register Allocator interface.

The Mspec information is primarily used by the IMPACT Front end to synthesize the Lcode from C source files. Mspec also includes information useful for the optimizer. For example, it has the cost of implementing each Lcode in the actual processor which is used by the optimizer to decide if merging operations is profitable or not. The Mspec is described as C functions. The Mdes is used by the IMPACT scheduler to build the Reservation Table which keeps track of the usage of processor's physical resources with cycle time as and when operations are scheduled. The Mdes is described in a high level text based representation called Hmdes [2].

Re-targetability is also enhanced by having a processor independent scheduler and register allocator. IMPACT also has a wide array of machine independent local, global and loop optimizations.

4 Hiperion Compiler

In this section, we describe the various problems encountered and the solutions we adopted in implementing a compiler for the Hiperion DSP within IMPACT compiler framework. The code generator flow in IMPACT that we used for this purpose is shown in Figure 1.

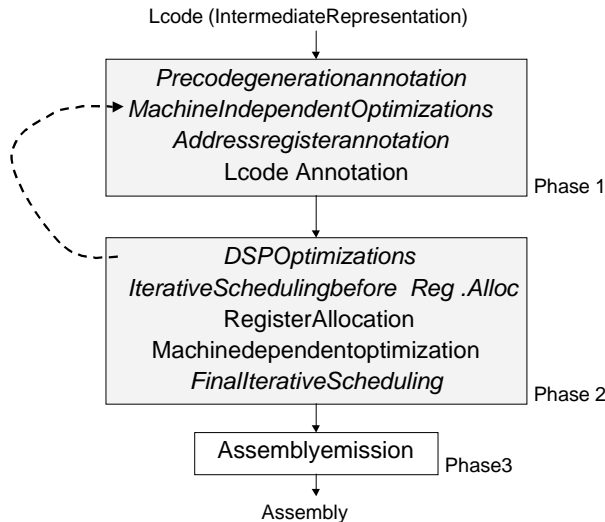


Figure 1: IMPACT Code Generator Flow

4.1 Pre-code generation annotation

The front end of the IMPACT compiler was made to generate Lcode consisting only of simple operations (like ADD, MUL etc) that cannot be broken into smaller operations. Before the code generation phase, a pre-annotation phase is introduced which splits the indirect loads and stores generated by the front-end into direct loads and stores as Hiperion does not support indirect memory addressing. This also provided a better opportunity to the optimizer to optimize the code.

4.2 Optimizations

Machine independent optimizations in IMPACT such as dead code elimination, common sub-expression elimination, copy propagation, operation combining, loop optimizations etc are then performed on the Lcode. MAC operations are also generated as peephole optimizations in this stage. Constant propagation was not performed as most operations in Hiperion with an immediate operand has a one cycle penalty. In order to prevent the loads and stores from being converted back into indirectly addressed memory operations, the cost of the recombination was made high in the Mspec. Loops whose iteration count is known statically and the loop index is used only in loop counter increment operation are optimized to make use of zero overhead looping hardware. This reduces the general register pressure as a machine register is now used in the place of a general register to keep the loop count.

4.3 Address register annotation

After the machine independent optimization phase, data flow analysis is performed to identify types of temporaries. For example, the Hiperion has different sets of operations for manipulating address registers and general registers.

It also restricts the destination register of multiply operations to be only accumulators. Hence to classify temporaries as either general, address or accumulator, two new register types namely, an address register type and an accumulator register type were created.

4.4 Operation Selection

The operation selection phase has the following responsibilities apart from tagging the Lcode with the corresponding processor opcode.

1. The three operand Lcode format is converted to the two operand Hiperion format using MOV operations
2. Identify the type of operation as an address operation or a general operation based on the dataflow analysis.
3. It is possible that one type of operation may end up with operands of several types. Hence MOV operations are necessary to generate correct code.

It is important to note that how the three responsibilities are implemented will affect the total number of MOV operations generated. Hence for each Lcode operation, the operation selection phase looks at all its operands and then decides the best set of Hiperion operations to generate.

4.5 DSP Optimizations

In this phase, some of the typical DSP optimizations like memory bank allocation to make use of the dual memory bank architecture, offset assignment [3] to exploit the auto increment and decrement feature and array reference allocation [1] to optimize array references by reducing the number of address register update operations are performed to decrease code size and improve performance.

4.6 Scheduling and Register Allocation

The Irregular Instruction Level Parallelism (ILP) in the Hiperion DSP described in Section 2 could not be directly described in the Mdes of the IMPACT compiler. This is primarily due to the fact that Hiperion unlike other VLIW (Very Long Instruction Word) processors allows a varying number of operations to be scheduled in the same cycle without paying a penalty in the form of NOPs. Hence an algorithm to convert such Irregular constraints into regular resource based constraints using artificial resources that can easily specified in the Mdes was developed. The input to this Artificial Resource Assignment algorithm is the set of all possible combinations of operations that a processor can execute in parallel. The output is a machine description file which has the minimum set of artificial resources assigned to each operation such that all the ILP constraints are satisfied. The main advantage of this algorithm is that it avoids writing of processor specific schedulers when irregular constraints exist and allows us to use the IMPACT table based scheduler.

Operation versioning problem In order to tackle the problem of operation versioning mentioned in Section 2, we use the ability of the IMPACT scheduler to allow multiple alternatives for an operation.

Alternative An alternative for an operation defines both the set of registers that each operand in that operation can be assigned and the set of resources that the operation uses with time.

For example, an alternative for an ADD operation can be `ADD [src(r_1, r_2), (r_3, r_4), dest(r_5, r_6)], [Issue (time 0); Adder (time 1 2); Writebus (time 3)]`

The first part specifies that r_1 and r_2 are the possible register choices for the first source operand, r_3 and r_4 are the register choices for the second operand and r_5 and r_6 are the register choices for the destination operand. The second part specifies the resource usage with time, that is, the ADD operation uses an Issue slot at time 0, an Adder functional unit at times 1 and 2 and uses the write bus at time 3.

When the scheduler picks an operation to schedule, it picks the first alternative from a list of alternatives for that operation that will allow it to schedule the operation in the earliest possible cycle. Hence it is important to order the list of alternatives in a judicious manner.

Once the scheduler picks an alternative for each operation and all the register files in the processor are defined, the task of the register allocator is clearly defined and a generic register allocator can be used to solve the problem.

But the problem of operation versioning can also be solved by scheduling operations without alternatives and then let the register allocator pick the correct alternative and hence the final schedule based on the register usage and pressure. While this makes the task of the scheduler easy, the register allocator has to be more complex and schedule sensitive.

Iterative schedule Although the use of alternatives solves the problem of operation versioning, it does not help overcome constraints such as the destination of a LOAD operation being the same as the source of an ADD operation when the ADD and LOAD are issued in the same cycle. We use an iterative scheduling technique as shown in Figure 2.

In this method, the code is first scheduled, then dataflow checks are performed. For example, in the above case, we check if the source of the ADD operation is used later on in the program by dataflow analysis. If it is used, then the opcode for the ADD is changed to that of an ADD which does not have the alternative which allows a LOAD to be performed in parallel with it. If there are any such changes, the code is re-scheduled and checked until no more transformations are performed. It is important to note that the new opcode is one that always restricts ILP. This is very important for the termination of the iterative schedule.

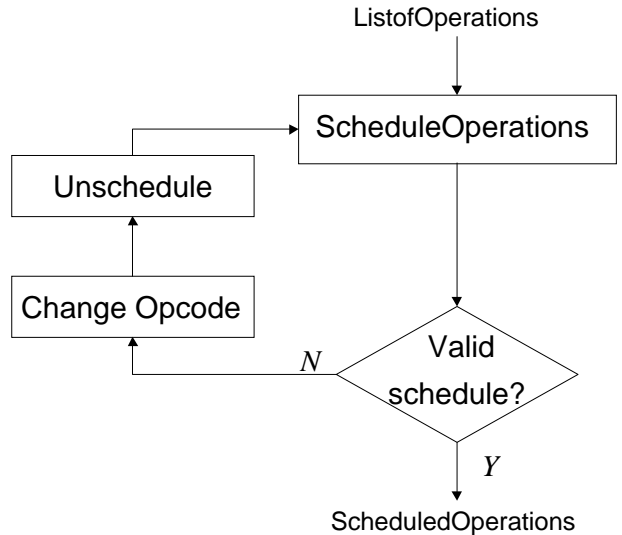


Figure 2: Iterative Scheduling

Bmark	Size VLIW	Size Spill	Size VLIW-Spill	Size DSP
<i>Simple</i>	73	40	33	34
<i>fir2dim</i>	291	79	212	280
<i>n_complex</i>	283	100	183	208
<i>n_real</i>	132	39	93	84
<i>iir_biquad</i>	276	125	151	178

Table 1: Experimental results on five DSP benchmarks. Size is the number of instruction words. Spill is spill code generated by our VLIW compiler.

5 Results

We present the experimental results using five DSP benchmarks [10] in Table 1. In this set of experiments we provide two sets of results for our VLIW compiler:

1. using just the VLIW compiler, without any DSP specific optimizations such as specific register allocation optimizations. The results are given in the 2nd column and
2. potentially employing DSP specific optimizations. The results are provided in in the 4th column. The estimated code size is obtained by subtracting the spill code size of 3rd column from VLIW code size of 2nd column.

The results of the last column are obtained using a Hiperion DSP specific optimizing compiler available to Fujitsu.

Comparing columns 4 and 5, it can be seen that except for the *n_real* benchmark, our VLIW compiler with appropriate register allocation optimizations (i.e. with no spill code) performs better than DSP specific optimizing compiler, which cannot sufficiently exploit ILP.

6 Conclusions and Future Work

In this paper we have described how we modified a re-targetable VLIW compiler framework to implement an optimizing compiler for fixed-point DSPs such as the second generation Hiperion. The irregular architecture of the Hiperion DSP posed numerous hurdles such as operation versioning to this task and we have provided ways to solve those problems. We have also successfully used the Artificial Resource Assignment algorithm to handle irregular ILP. The constraints encountered in this exercise are common in low cost, low power DSPs for which containing code size is very essential.

Although at this time, our VLIW compiler lacks most of the DSP-specific optimizations such as optimal register allocation, we have shown that the assembly code size is smaller than that produced by a DSP compiler without efficient ILP/VLIW optimizations support. Thus, we conclude that it is more difficult to enhance an existing DSP-specific compiler with ILP and SIMD support than to employ DSP-specific optimizations within a VLIW compiler, that has in-built ILP/SIMD support. As part of future work, we plan to introduce DSP-specific optimizations in the VLIW compiler to further reduce the code size.

References

- [1] M. Cintra and G. Araujo. Array reference allocation using ssa-form and live range growth. In *Proceedings of ACM SIGPLAN 2000 Workshop Languages, Compilers and Tools for Embedded Systems*, June 2000.
- [2] J. C. Gyllenhaal, W. mei W. Hwu, and B. R. Rau. Hmdes version 2.0 specification. Technical report, University of Illinois, Urbana, 1996.
- [3] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18:235–253, May 1996.
- [4] P.P.Chang, S.A.Mahlke, W.Y.Chen, N.J.Warter, and W.W.Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275, May 1991.
- [5] S. Rajagopalan, M. Vachharajani, and S. Malik. Handling irregular ilp within conventional vliw schedulers using artificial resource constraints. In *Proceedings of International Conference on Compilers Architecture and Synthesis for Embedded Systems*, November 2000.
- [6] S. Rajan, M. Fujita, A. Sudarsanam, and S. Malik. Development of an optimizing compiler for a fujitsu fixed-point digital signal processor. In *Proceedings of 7th International Workshop on Hardware Software Codesign*. ACM SIGDA, May 1999.
- [7] B. Rau, V. Kathail, and S. Adithya. Machine-description driven compilers for epic processors. Technical Report HPL98-40, Hewlett-Packard Laboratories, September 1998.
- [8] R.E.Hank. Machine independent register allocation for the impact-1c compiler. Master's thesis, University of Illinois, Urbana, 1993.
- [9] A. Sudarsanam. *Code Generation Libraries for Re-targetable Compilation for Embedded Digital Signal Processors*. PhD thesis, Princeton University, November 1998.
- [10] V. Živojnović, J. M. Velarde, and C. Schläger. DSP-stone: A DSP-oriented Benchmarking Methodology. Technical report, Aachen University of Technology, August 1994.