A Retargetable VLIW Compiler Framework for DSPs With Instruction-Level Parallelism

Subramanian Rajagopalan, Sreeranga P. Rajan, Sharad Malik, Senior Member, IEEE, Sandro Rigo, Guido Araujo, and Koichiro Takayama, Member, IEEE

Abstract—A standard design methodology for embedded processors today is the system-on-a-chip design with potentially multiple heterogeneous processing elements on a chip, such as a very long instruction word (VLIW) processor, digital signal processor (DSP), and field-programmable gate array. To be able to program these devices, we need compilers that are capable of generating efficient code for the different types of processing elements with efficiency measured in terms of power, area, and execution time. In addition, the compilers should also be highly retargetable to enable the system designer to quickly evaluate different cores for the application on hand and reduce the time to market. In this paper, we show that we can extend a conventional VLIW compilation environment to develop highly retargetable optimizing compilers for DSPs with irregular architectures. We have used the second generation Fujitsu Hiperion fixed-point DSP as our primary example to evaluate the compiler framework. We demonstrate through experimental results that execution time for the assembly code generated using our framework is roughly two times better than that of the code generated by a widely used commercially available DSP compiler. Even without incorporating DSP-specific optimizations in our extended VLIW framework, we demonstrate that the compiled code has a better performance than the code generated by a commercial DSP-specific compiler in all our examples.

Index Terms—DSP compiler, irregular architecture, optimization, system on a chip.

I. 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. It is for this reason that digital signal processors (DSPs) have specialized architectural features such as address generation units and accumulator-based data paths. Unfortunately, obtaining sufficiently dense software by making use of these features has remained a challenge for optimizing compilers for embedded DSPs. Existing DSP compilers generate assembly code whose size is on the average 40% larger than the size of the corresponding handwritten assembly code [11].

The inability of DSP compilers to meet code density and performance constraints at the same time often forces embedded system designers to program in assembly. The DSP architects in turn design application-friendly processors instead of compiler-friendly architectures. However, system designs based on this methodology are unacceptable today because of their long development time and lack of portability to the ever-changing and increasing application demands. In order to bridge this productivity gap, we need compilers capable of not only supporting, but also exploiting DSP specialized architectural features to generate high-quality code in a retargetable manner.

These requirements of a DSP compiler automatically fit well into what a VLIW compilation framework has to offer. A very long instruction word (VLIW) framework has a set of optimizations that can exploit the instruction-level parallelism (ILP) in the DSP, such as multiple specialized functional units and memory banks that allow simultaneous execution of parallel operations. VLIW compilers can be made highly retargetable by using a machine description database (MDD) [7][17], which describes a processor’s microarchitecture and the instruction-set architecture (ISA). A significant advantage of having an MDD is that it allows automation of the traditional backend phases of a compiler such as scheduling, register allocation, and assembly code generation [1].

In this paper, we have developed an optimizing compiler framework for DSPs using the IMPACT VLIW [14] compiler framework. We have used the second generation Fujitsu Hiperion fixed-point DSP as our primary example to evaluate the compiler framework with respect to the quality of the assembly code generated. In an earlier work [16], using the SPAM compiler framework [19], a high-quality optimizing compiler for Fujitsu Elixir, a fixed-point DSP used primarily in cellular telephones, was developed. SPAM is a DSP compiler framework with restricted support for ILP. While retargetability in SPAM was achieved using compiler code libraries, we obtain retargetability by using an MDD. Earlier work [11] on MDD-based DSP compilation view a fixed-point DSP as a processor with an irregular architecture. They provide support for the numerous constraints imposed by the DSP either by allowing special constraint specification in the MDD or by tailoring the different phases of the compiler to suit the various constraints. On the contrary, in our work, we logically transform a DSP with irregular architecture into an equivalent regular VLIW architecture by means of a preprocessing step, thereby reducing the DSP specific code inside the compiler framework to a minimum. We demonstrate through experimental results that execution time for the generated code is roughly two times better than that of the code generated by a widely used commercially available DSP compiler. The main reason why we achieve results superior to a DSP-specific compiler framework...
is that by using our enhanced VLIW framework, we are able to exploit irregular ILP found in DSP architectures to optimize loops. However, we do not sacrifice retargetability.

The organization of the rest of this paper is as follows. The various constraints imposed by DSPs are detailed in Section II using the Hiperion DSP as an example. The IMPACT compiler framework is described in Section III. Section IV describes the DSP compiler framework that we have developed using IMPACT. The results and comparison with other available compilers for Hiperion are provided in Section V. Conclusions are summarized in Section VI.

II. ISSUES IN RETARGETABLE FIXED-POINT DSP COMPIILATION

In this section, we briefly outline some of the common problems faced in retargetable fixed-point DSP compilation.

We highlight the ISA constraints imposed by such data paths in Section II-A using the Fujitsu Hiperion as an example. Fig. 1(a) shows the data path of Hiperion, a typical low-cost fixed-point DSP. It consists of two memory banks, an arithmetic logic unit (ALU) and an address generation unit. The ALU consists of a shift unit, an add unit, and a multiply unit. The register set of such DSPs includes accumulators, general-purpose registers, and address registers (ARs). There are eight ARs and four 32-bit accumulators, each of which is also accessible as two 16-bit general-purpose registers. The two memory banks allow two memory accesses to occur in parallel. In addition to the parallel memory accesses, ILP also exists between the ALU and memory units. However, this ILP requires a lot of conditions to be satisfied which are explained in Section II-A.

A. Code Generation Constraints

The primary application of DSPs such as the Hiperion core is in cellular telephones. Hence, it is extremely important not only to have dense software to reduce area and cost, but also to have a good execution performance to extend the life of the battery as much as possible. It is for this reason that most of the constraints described below arise due to encoding of the instructions. Such constraints are very difficult to capture in both behavior description-based MDDs (MDDs based on ISA descriptions) and structural description-based MDDs (MDDs that extract ISA information from description of a processor’s data path). It is also important to keep in mind that the internals of a processor are seldom transparent to the application developer.

1) Operation ILP Constraints: These constraints describe what a set of operations can and cannot be issued in parallel. Fig. 1(b) shows the set of ILP constraints in Hiperion. MEMop stands for either a LOAD or a STORE operation, ALU stands for operations that use the ALU such as ADD, SUB and Shift operations and MAC stands for all operations associated with the MAC unit such as Multiply, Multiply-Accumulate, etc. Although two LOAD operations can be issued in parallel with a MAC operation, only one LOAD can be issued in parallel with an ALU operation. Hence, these constraints need not be limited by physical resources alone.

2) Operand ILP Constraints: These constraints describe how registers should be assigned to operands of operations issued in parallel. These constraints may or may not affect data flow.

Example—Data-Flow Affecting Constraint: MUL dest1, source1, source2; LOAD dest1, [M1]; LOAD dest2, [M2] is a valid VLIW instruction only if dest1 and dest2 are assigned the same registers as source1 and source2, respectively. Whether this constraint is satisfied or not depends on the data-flow conditions on source1 and source2. In addition, Hiperion ISA also restricts what registers can be assigned to each operand in an instruction.

Example—Constraint Not Affecting Data Flow: ADD dest1, source1, source2; LOAD dest2, [M] is a valid single cycle schedule only if the following conditions are satisfied. If dest1 is assigned CX, then:

1) dest2 can be assigned a register only from the following set of registers \{A0, A1, B0, B1, D0, D1, CX, DX\};
2) source1 can be assigned only one of \{CX, A0\};
3) source2 must be assigned register A1.

While the first constraint is an example of operand constraints across operations, the other two are examples of operand constraints within an operation.

There are also constraints on what addressing mode a memory operation can use, how the latency of an operation varies depending on its operands, resource usage constraints that describe how a processor’s resources are used with time and others. Due to such constraints on the ISA, the same operation has different constraints under different instances. For example, an ADD operation can have different register constraints on its operands depending on what operations are scheduled in parallel with it. We define each instance of an operation as an operation version and the problem of choosing the correct version for an operation as the operation-versioning problem. Hence, it is the compiler’s responsibility to not only ensure that the correct set of constraints are satisfied, but also to ensure that the optimal version for an operation is selected.

III. IMPACT COMPILER FRAMEWORK

In this section, we give a brief overview of the IMPACT [14] compiler framework. IMPACT is a retargetable compiler framework for multiple instruction issue processors. The frontend of IMPACT produces as output an intermediate representation (IR) called Lcode. Prior to code generation, machine-independent
optimizations may be applied on Lcode. Code generation for a specific target architecture is performed in three phases as shown in Fig. 2.

**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, prepass scheduling, register allocation [18], 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) In machine description or Mdes [7], [17], 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 frontend to synthesize the Lcode from C source files. Mspec also includes information that can guide the optimizer. For example, it has the cost of implementing each Lcode operation 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 and when operations are scheduled. The Mdes is described in a high-level text-based representation called Hmdes [7].

Retargetability 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.

### IV. DSP Compiler Support

In this section, we describe the various solutions we adopted to tackle the constraints mentioned in Section II. The DSP code generator flow that we adopted within IMPACT is shown in Fig. 3. We now briefly explain each step in Fig. 3.

#### A. Precode Generation Annotation

The frontend of the IMPACT compiler is made to generate Lcode consisting of only simple operations (like ADD, MUL, etc.) that cannot be broken into smaller operations. This new step is introduced not only to help the optimizer, but also to ensure correctness. For example, it is necessary to convert the three operand Lcode format to two operand format used in most DSPs using MOV operations. In addition, this step also splits the indirect loads and stores generated by the frontend into direct loads and stores if the DSP does not support indirect memory addressing. This provides a better opportunity for the optimizer to condense the code.

#### B. Optimizations

Machine-independent optimizations [1] in IMPACT [14] such as dead code elimination, common subexpression elimination, copy propagation, operation combining, loop optimizations, etc., are performed on Lcode after the Precode generation annotation phase. MAC operations are generated as peephole optimizations in this stage. Several common machine-independent optimizations in general-purpose compilation arena have to be applied with caution for DSPs. For example, constant propagation, which reduces register usage, often leads to poor results for fixed-point DSPs. This is due to the typical one instruction-word penalty and, consequently, a one-cycle penalty to fetch operations with large immediate values that the DSPs suffer.

In order to prevent the optimizer from undoing what the Precode generation annotation phase did, the cost of the recombining of memory operations and address modifying operations is made high in the Mspec. Loops whose iteration counts are known statically and whose loop indexes are used only to increment the loop counts are optimized to make use of the zero overhead looping hardware. In addition to utilizing zero overhead looping capability of DSPs, this step also reduces the pressure on general-purpose registers as a dedicated machine register is now used instead of a general register to keep the loop count.

After this step, the Precode generation annotation phase may have to be performed again for correctness and more optimizations. This loop is shown as the dotted line in Fig. 3.

#### C. Address Register Annotation

After the machine-independent optimization phase, data-flow analysis is performed to identify the types of temporaries. In DSPs, the set of operations used for manipulating contents of ARs is restricted to simple operations, such as addition or subtraction of constants or other ARs. This set of operations is usually orthogonal to the set of operations used to modify contents of general-purpose registers. Often, it is also the case that the data path restricts the destination register of multiply operations to only accumulators. Hence, to classify temporaries as either

![IMPACT code generator flow](image-url)
general, address, or accumulator type, two new register types, namely, an AR type and an accumulator register type are introduced. In order to perform more complex operations than simple addition of constants to ARs, the computation is performed on general-purpose registers using general-purpose operations and the results are copied back to ARs.

Since this step can introduce a lot of copy or MOV operations, it is beneficial to perform machine-independent optimizations such as copy propagation again.

D. Operation Selection or Lcode Annotation

The operation selection phase tags the Lcode operations with the corresponding processor opcode.

E. DSP Optimizations

In this phase, some of the DSP optimizations such as memory bank allocation to exploit the dual memory bank architecture, offset assignment [9] to exploit the auto increment/decrement feature, and array reference allocation are performed. The ISA of DSPs provides special address modifying instructions that update ARs, which are used by memory operations. Since many of the signal processing applications contain kernels that operate on arrays, one of the most important DSP optimizations is to carefully assign ARs to array references so that a majority of the extra address computation operations can be replaced by auto increment/decrement address modifying operations which are free. This can significantly affect the static code size and the dynamic performance of the assembly code as address computation for successive memory operations is performed in parallel with current memory operations. Our implementation uses an enhanced variation of array reference allocation that is described in Section IV-E1.

1) Array Reference Allocation: We call global reference allocation (GRA) the problem of allocating ARs to array references such that the number of simultaneously live ARs is kept below the maximum number of such registers available in the processor and the number of instructions required to update them is minimized. The local version of this problem, called local reference allocation (LRA), has all references restricted to basic block boundaries. LRA has been studied before and there are known efficient graph-based solutions for it such as [2], [6], and [8]. Unfortunately, no general solution exists for GRA. Our compiler infrastructure adopts a solution that is discussed below.

In order to bound allocation to this maximum number of ARs, sometimes it is necessary that two references share the same register. This is done by inserting an instruction between the references or by using autoincrement (decrement) mode. Fig. 4 shows two fragments of C source code that illustrate the effect of applying array reference allocation. If we allocate one AR to each different reference, \( a[i] \) and \( a[i+2] \), in Fig. 4(a), we would need two ARs. By changing the code to use just one pointer to perform all array accesses, as in Fig. 4(b), only one AR is needed for this loop.

Our GRA algorithm is called live range growth (LRG). It is shown in Fig. 5, where \( nars \) is the number of ARs available in the processor, \( R \) is the set of all array references in the program, \( R \) and \( S \) are ranges, and \( R \bowtie S \) represents the new range resulting from the merge operation applied to \( R \) and \( S \).

This technique is based on an operation that successively merges pairs of live ranges. These live ranges are sets of array references of a program and all the references that belong to the same range share the same AR.

The algorithm starts by partitioning all references across a set of initial live ranges, which is done by assigning each individual reference to a separate range. From this point on, ranges
are merged pairwise until the number of ranges is smaller than or equal to the maximum number of ARs available in the processor. The cost of merging two ranges \( R \) and \( S \) is related to the total number of update instructions required by the merge. At each step, the algorithm chooses to merge the pair of ranges which incurs the minimal merge cost.

The merge operation is the crucial point of the LRG algorithm. During the merge process, the algorithm decides whether it needs to insert an update instruction to redirect the AR from reference \( a \) to reference \( b \), based on the indexing distance between these references. If this distance is less than or equal to one, it means that we can use autoincrement (or decrement) mode to redirect the register. Otherwise, an explicit update instruction will be inserted on the path from \( a \) to \( b \).

Let \( a \) and \( b \) be array references and \( s \) be the increment of the loop containing these references. Let \( \text{index}(a) \) be a function that returns the subscript expression of reference \( a \). The indexing distance between \( a \) and \( b \) is the positive integer

\[
\text{d}(a, b) = \begin{cases} 
\text{index}(b) - \text{index}(a), & \text{if } a < b \\
\text{index}(b) - \text{index}(a) + s, & \text{if } a > b
\end{cases}
\]

where \( a < b \) or \( a > b \) if \( a(b) \) precedes \( b(a) \) in the schedule order.

In Fig. 6, we have an example of merge operation. In 6(a), references are divided into live ranges \( R \) and \( S \). In our notation for live range, small squares represent array references and squares with the same color are in the same range. Moreover, an edge between two references of a live range indicates that the references are glued together through autoincrement mode or an update instruction. In Fig. 6, symbol "++" ("--") following a reference assigns a postincrement (decrement) mode to that reference. Notice that a live range can include references across distinct basic blocks. In Fig. 6(b), all references belong to the same range \( R \ni S \), which means that they will share the same AR. The cost of this merge operation is given by the three new instructions that had to be inserted to update the register.

In order to enable references to share the same AR, the algorithm needs to enforce, at compile time, that each reference \( b \) should be reached by only one reference \( a \), since this will allow the computation of the indexing distance \( d(a, b) \) at compile time. This requirement can be satisfied if the references in the control flow graph (CFG) are in single reference form (SRF) [3], a variation of static single assignment (SSA) form [4]. Similarly, as in the case of SSA form, \( \phi \) functions are inserted at certain join basic blocks. These blocks are given by the iterated dominance frontier [4] of the set that contains all references. Further details on the SSA form may be found in [4] or modern compiler optimization books such as [13]. Fig. 7 shows the CFG of our example in Fig. 4 with the references in the SRF. The result of each \( \phi \) function (in blocks B1, B3, and B6) is assigned to a new (virtual) array reference \( \text{wt} \), which becomes the resulting reference at the joining block. These are called virtual references because they only tell the compiler that the AR must be pointing to these resulting elements in blocks that have \( \phi \) functions such that the register will be pointing to the correct array element the next time it is used in the paths. By solving all \( \phi \) functions, we can see that three update instructions will be necessary to make all references share the same AR in this case.

GRA is a typical DSP optimization, but, as the code generator flow shows in Fig. 2, it is implemented in machine-independent optimization step. Actually, it is performed before most of the classical optimizations because techniques such as common subexpression elimination (CSE) may destroy some opportunities of applying GRA. For example, by allowing two array references to share the same index expressions, one cannot encapsulate array references, precluding GRA.

IMPACT has a module called Lopti, which is responsible for doing all machine-independent optimizations in Lcode. We have
implemented GRA and attached it to the Lopti module, which calls GRA routines just after loop detection and jump optimization.

IMPACT has implemented almost everything we needed to add GRA to the _Lcode_ module. There were just two things missing: routines to compute dominance frontier [13] and type information to make array references identifiable on _Lcode_. We have added functions to compute dominance frontier and iterated dominance frontier to the _Lcode_ module. These features are necessary to put array references in the SRF [3], which is required to apply GRA.

Another modification we made to IMPACT is related to the annotation of array references. GRA requires array references to be kept atomic. Unfortunately, IMPACT dismantles all array references into smaller _Lcode_ instructions. One way to enable the GRA algorithm to recover the original reference is to annotate the _Lcode_ instructions associated to the dismantled reference such that it is distinguishable from other _Lcode_ instructions. To address this, we added type information to some _Lcode_ load/store instructions, telling the Lopti module when they are associated to array elements. This attribute is set in the last phase of IMPACT’s frontend, while expressions are being converted to _Lcode_ operations.

Having load/store operations that are array accesses identified and performing GRA before any code motion/elimination allows us to recover index expressions by matching patterns of add, sub, mul, and load/store instructions in the _Lcode_. After extracting this information from the _Lcode_, we apply the LRG algorithm to perform the array reference allocation optimization.

**F. Scheduling and Register Allocation**

Here, we describe how the operation and operand constraints described in Section II are overcome in a retargetable manner.

1) _Operation ILP Problem Revisited_: As stated in Section II, the irregular ILP in DSPs cannot be directly described in the Mdes of the IMPACT compiler. For example, from the operation ILP constraints shown in Fig. 1(b), we can see that there are some problems associated with defining the issue width of the processor. We know that the issue width of the processor must be at least three as there are a maximum of three operations in parallel. However, if the issue width is defined as three, we need to ensure that the other operation ILP constraints shown in Fig. 1(b) are not violated.

In order to solve such problems, we use a preprocessing step which uses the artificial resource assignment (ARA) algorithm [15], shown in Fig. 8 and briefly described here. This step converts irregular ILP into a set of resource-based ILP constraints by assigning artificial resources to the operations. The new artificial resources may not have a physical equivalent in the data path.

This algorithm takes as input the set of all possible combinations of operations that can be issued in parallel. The output of ARA algorithm is an augmented resource usage section of the Mdes with artificial resource usages for each operation such that the following constraints are satisfied.

**Constraint 1**: Every pair of operations that cannot be issued in parallel must share an artificial resource.

**Constraint 2**: Every pair of operations that can be issued in parallel must not share a resource.

**Constraint 3**: The total number of artificial resources generated must be minimum.

Since the size of the reservation table is dependent on the total number of resources and the resource usage of each operation, we introduce the third constraint, which minimizes the total number of artificial resources generated. In addition to these constraints, the algorithm also needs to be retargetable. Hence, we adopted a graph-based algorithm that uses graph coloring techniques as described below. The main steps in the algorithm shown in Fig. 8 are explained with an example from the _Hiperion_ ISA as shown in Fig. 9.

1) The first step of ARA algorithm is construction of a compatibility graph \( G = (V, E) \). A vertex \( v \in V \) in the graph corresponds to an operation in the ISA and an edge \( e = (v_i, v_j) \in E \) exists between two vertices \( v_i, v_j \in V, i \neq j \) if the two corresponding operations can be performed in parallel. For example, in Fig. 9, there are five vertices in the compatibility graph, one each for _ADD_, _Shift_, _Multiply_, and two _Load_. The parallel combinations in Fig. 1(b) are captured by the straight line edges in the compatibility graph.

2) The complement \( G' \) of the compatibility graph shown in dotted lines in Fig. 9 is then constructed. An edge between two vertices in \( G' \) implies that the two operations cannot...
be performed in parallel. An immediate solution that follows is to assign an artificial resource to each edge in \( G' \). Since this solution can cause the size of reservation table to explode, a better solution is obtained in the next step.

3) From [15], the minimum number of artificial resources needed to satisfy the operation ILP constraints is obtained by performing the minimum edge clique cover [5] on \( G' \). Note that a clique in \( G \) represents the fact that no two operations represented as vertices in \( G \) can be issued in parallel and, hence, all operations in a clique in \( G \) can share a resource. In order to use graph coloring techniques, the problem of finding the minimum edge clique cover on \( G' \) is first converted to an equivalent problem of finding the minimum vertex clique cover [5] on graph \( G_1 \). \( G_1 \) is obtained from \( G' \) based on a transformation given in [10]. This transformation involves adding a vertex in \( G_1 \) for every edge in \( G' \) and an edge exists between two vertices in \( G_1 \) iff the vertices connected by the two corresponding edges in \( G' \) form a clique. The problem then simplifies to performing graph coloring on the complement of \( G_1 \) [5]. We leverage the research done in graph coloring to both obtain the exact solution [12] and for heuristics. In the example shown in Fig. 9, three cliques \( C1, C2, \) and \( C3 \) are required to obtain the minimum edge clique cover.

4) The final step in the ARA algorithm translates the result of the minimum edge clique cover algorithm into the resource usage section of Mdes in IMPACT.

The advantages of using this algorithm are that it is a highly retargetable solution to the operation ILP problem and it helps avoid processor specific schedulers when such irregular constraints exist, thereby allowing us to use the IMPACT’s reservation table-based scheduler.

2) Operation Versioning Problem Revisited: In order to tackle the problem of operation versioning mentioned in Section II, we use the ability of the IMPACT scheduler to allow multiple alternatives for an operation [7].

An alternative for an operation defines both the set of registers that each operand in that operation can be assigned and its resource usage [7], [14].

For example, an alternative for an ADD operation can be \( \text{ADD} \ [\text{src} (r_{1}, r_{2}), (r_{3}, r_{4}), \text{dest}(r_{5}, r_{6})] \). [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, i.e., 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.

We illustrate the use of alternatives in solving operation versioning problems by means of an example from the Hiperion. The complete set of constraints for \( \text{MUL} \), \( \text{dest1} \), \( \text{source1} \), \( \text{source2} \); \( \text{LOAD} \), \( \text{dest2} \), [M] to be a valid single cycle schedule are as follows:

1) \( \text{source2} = A1 \);
2) \( \text{source1} = \) one of the registers \( A0, A1 \);
3) if \( \text{dest1} = CX \), \( \text{dest2} = \) one of the registers \( A0, A1, B0, B1, D0, D1, CX, DX \);
4) if \( \text{dest1} = DX \), \( \text{dest2} = \) one of the registers \( A0, A1, B0, B1, C0, C1, CX, DX \).

While constraints 1 and 2 are examples of operand constraints within an operation, constraints 3 and 4 are examples of operand constraints across operations. The processor-independent register allocator in IMPACT cannot take into account register allocation constraints such as constraints 3 and 4, where the register assignment options for an operand in an operation is dependent on assignment of registers to other operands in the same operation. For this reason, we generate two alternatives for the \( \text{MUL} \) operation, namely, \( MUL_{CX} \) and \( MUL_{DX} \) corresponding to constraints 3 and 4. Similarly proceeding for the case where a \( \text{MUL} \) and two \( \text{LOADs} \) can be in parallel, we end up with two more alternatives for \( \text{MUL} \). The \( \text{MUL} \) operation has a total of five alternatives, including the case where there are no other operations in parallel with it. Since each alternative has the same resource usage, we now apply the ARA algorithm described in Section IV-F1 on alternatives instead of operations as shown in Fig. 3.

Although we solved the problem of operation versioning using alternatives, we introduced a new problem of choosing a good alternative for each operation. Since each operation can have several alternatives as illustrated above, the problem of picking the optimal alternative can be combinatorially explosive. The following heuristic was used to solve this problem. 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. As it is now important to order the list of alternatives in a judicious manner, we list the alternatives in the decreasing order of the number of operations that can be issued in parallel with that alternative.

3) Iterative Schedule: The techniques described to overcome the operation and operand ILP constraints described earlier have some limitations.

1) The ARA algorithm described in Section IV-B used to overcome operation ILP constraints has some assumptions [15] on the ISA. For example, the ARA algorithm requires that if a certain set of operations can be issued in parallel, then any subset of those operations should also be capable of being issued in parallel. Although we expect most DSP architectures to satisfy these constraints, this requirement may not be met by some DSPs due to limitations in operation encoding.

2) The alternatives technique used to overcome operand ILP constraints cannot tackle constraints that affect data flow mentioned in Section II.

In order to overcome these two limitations, we use an iterative scheduling technique as shown in Fig. 10.

In this method, the code is first scheduled using the Mdes. Then, data-flow checks are performed if required. This is a DSP specific step that needs to be in the framework. For example, in the example for operand ILP constraint with data flow mentioned in Section II-B, we check if the source operands of the
MUL operation are used later on in the program by data-flow analysis [1]. If it is used, then the alternative for the MUL is changed to the next one in the list of alternatives for MUL. In a similar fashion, alternatives violating the assumptions of ARA algorithm are also changed. If there are any such changes, the code is rescheduled and checked until no more transformations are performed. It is important to note that the new alternative is one that restricts ILP. This is very important for the termination of the iterative schedule.

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 IMPACT’s general processor-independent register allocator [18] can be used to solve the problem of register allocation.

V. RESULTS

We have implemented a compiler for the Fujitsu Hiperion DSP using the Enhanced IMPACT framework. We present the experimental results using DSP Stone benchmarks [20] in Tables I–III. Table I gives a comparison of the number of static DSP instructions generated by two compilers, namely, our Enhanced IMPACT framework and a commercial DSP specific compiler. Except for two examples with respect to static code size—N.real.updates and N.complex.updates, as shown in Table I—enhanced IMPACT compiler performs better than the commercial compiler for all benchmarks with respect to both dynamic code size/execution cycles and static code sizes.

Upon examining the code generated using IMPACT, it was found that the reason for increase in code size of N.real.updates and N.complex.updates is a lot of spill code and this in turn is due to the prepass scheduling phase. This phase is aware of only the operation constraints and is totally unaware of all the operand constraints. This leads to increased usage of only certain registers, thus, forcing the register allocator to spill. Also, in general-purpose compilation, the long latency operations like memory operations get higher priority over integer arithmetic operations during scheduling. We found that moving memory operations further away from the instructions using them and scheduling them earlier almost always leads to increased spilling. In addition, due to the small heterogeneous register set, it is better to load values from memory whenever needed than to keep them in registers. So, it was always better to turn off memory copy propagation. As can be expected from typical general-purpose compilers, the enhanced IMPACT framework performs very well in the dynamic instruction count and execution cycles. This is mainly due to the fact that most of the loops are well optimized by IMPACT’s loop optimizations, thus, reducing the dynamic instruction count.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have shown how to extend a conventional VLIW compilation environment to develop highly retargetable optimizing compilers for DSPs with irregular architectures. We have demonstrated through experimental results that execution time for the assembly code generated using our framework is roughly two times better than that of the code generated by a widely used commercially available DSP compiler. Note that we have achieved the results even without incorporating DSP-specific optimizations. The main reason why we could achieve such a performance advantage is that we could effectively optimize loops using our VLIW framework extended using artificial resource assignment algorithm to support irregular ILP architec-
tures. We obtain the performance advantage without sacrificing retargetability.

As part of ongoing and future work, we are focusing on the following extensions:

1) improving the register allocation heuristic to perform efficient register allocation for architectures with heterogeneous register sets;
2) code size and irregular constraint-sensitive optimizations;
3) support for DSP-specific optimizations;
4) support for generation of subword parallel/media instructions.

Thus, we believe that our framework will help in developing a synergistic architecture-compiler design methodology for future embedded processors.

REFERENCES

Subramanian Rajagopalan received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1994 and the M.A. degree in electrical engineering from Princeton University, Princeton, NJ, in 1998. He is currently working toward the Ph.D. degree in electrical engineering at the same university.

His current research interests include compilers, networking, and security.

Sreeranga P. Rajan received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1986, the M.S. degree in electrical engineering from the University of Southern California, Los Angeles, in 1987, and Ph.D. degree in computer science from the University of British Columbia, Vancouver, BC, Canada, in 1995.

He is currently a Member of Research Staff with the Fujitsu Laboratories of America, Sunnyvale, CA. Previously, he was with Stanford Research Institute (SRI) International and Philips Research Laboratories. His research interests include compilers, high-level design tools for hardware and software, verification, and storage area networks.

Sandro Rigo was born in Jundiai, Brazil, in 1975. He received the B.Sc. degree in applied mathematics and the M.Sc. degree in computer science from the State University of Campinas (UNICAMP), Campinas, Brazil, in 1997 and 2000, respectively. He is currently working toward the Ph.D. degree in computer science at the same university.

His current research interests include code optimization for embedded systems, simulation, and description languages for computer architectures.
Guido Araujo received the Ph.D. degree from Princeton University, Princeton, NJ, in 1997.

He is currently an Associate Professor with the Institute of Computing, State University of Campinas, Campinas, Brazil. Previously, he was a Compiler Consultant with Conexant Systems Inc. and Mindspeed Inc. He is the Brazilian coordinator of the ProTeM-CC CNPq/NSF Collaborative Research Project on Design Environments for Application-Specific Programmable Processors, a collaboration between Princeton University, MIT, and UNICAMP, and is a CNPq Research Associate. He is also one of the designers of the SPAM compiler. He has authored or coauthored a number of papers in journals and conference proceedings, including ACM Transactions on Design Automation of Electronic Systems and the ACM/IEEE Design Automation Conference. He has refereed for a number of journals and conferences and served on the program committee of the ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems. His current research interests are in the area of compiler optimizations for embedded processors, including code generation and code compression algorithms, dynamic compilation, architecture evaluation, and all aspects of computing related to the design of embedded systems.

Prof. Araujo received the Best Paper Award at the 1996 ACM/IEEE Design Automation Conference.

Koichiro Takayama (M’98) received the B.E. and M.E. degrees in electronic engineering from Osaka University, Osaka, Japan, in 1985 and 1987, respectively.

He is currently a Senior Researcher with the Fujitsu Laboratories of America, Sunnyvale, CA. From 1987 to 1998, he was a Research Scientist with the Fujitsu Laboratories, Kawasaki, Japan. His current research interests include computer-aided design of digital systems.