MASTER

Fast and efficient vector DSP simulation through automatic vectorization

Mundichipparakkal, J.

Award date:
2016

Link to publication
Fast and Efficient Vector DSP Simulation through Automatic Vectorization

Author:
Jumana Mundichipparakkal
0926084, MSc.Embedded Systems

Supervisors:
Prof. Dr. Henk Corporaal
Electronic Systems Group

Dr. Roel Jordans
Electronic Systems Group

Dr. Mohamed Bamakhrama
Intel Corporation

External Committee Member:
Prof. Dr. Kees Van Berkel
Department of Mathematics and Computer Science

August 31, 2016
Abstract—Vector DSPs are quite common in embedded SoCs used in compute-intensive domains such as imaging and wireless communication. To achieve short time to market, it is crucial to provide system architects and SW developers with fast and accurate instruction set simulators of such DSPs. To this end, a methodology for accelerating the simulation of vector instructions in vector DSPs is proposed. The acceleration is achieved by translating the vector instructions in a given vector DSP binary into host SIMD instructions. The key advantage of the proposed methodology is its independence from the host architecture. This independence is achieved through: (1) describing the semantics of each target instruction as an inlined auto-vectorizable C/C++ function, and (2) exploiting the state-of-the-art auto-vectorizing compilers to transform these semantics automatically into host SIMD instructions. A prototype tool that realizes the proposed methodology has been implemented and tested using a set of real imaging kernels targeting a set of commercial vector DSPs. Preliminary results show that on average, the proposed methodology provides a 4x reduction in simulation time of a vector operation\(^1\) compared to the unaccelerated version.

I. INTRODUCTION

Programmable vector DSPs have emerged in the past two decades as a key enabler of achieving the right balance between programmability, high performance, and low power in several compute-intensive domains such as imaging [1] and wireless communication [2]. These vector DSPs enable SoC designers to differentiate their products by implementing customizable features in SW and delivering post-silicon updates to the customers as needed. In order to reduce the time to market, it is of utmost importance to start the SW development for such DSPs as early as possible. Ideally, the SW development can start as soon as a functionally-correct executable model of the HW is available to both HW and SW teams. Typically, these executable models embed processor simulators that simulate the different processors integrated in the system. For SW developers, the speed of these processor simulators is quite important. In this work, we tackle the problem of accelerating vector DSP simulators. Vector DSPs and their workloads have distinct properties, that need to be considered while designing their simulators, compared to general-purpose processors. Firstly, these vector DSPs are often based on VLIW architectures with varying instruction sets [3] per generation. As a result, the simulators need to be retargetable. Secondly, the programs running on these DSPs in domains such as imaging and wireless communication are quite often static (i.e., not self modifying).

Traditionally, three approaches exist for simulating processors [4]: (1) interpreted simulation, (2) compiled simulation, and (3) dynamic translation. Compiled simulation is known to be the fastest but the least flexible, while interpreted simulation is the most flexible but the slowest one. Regardless of the chosen simulation approach, the traditional approach of simulating the vector instructions is through for-loops that execute the corresponding host scalar instructions sequentially. This approach is quite inefficient as each M-cycle target instruction takes \(M' \times N\) host cycles to be simulated, where \(M'\) is the cost (in cycles) of the scalar host instruction and \(N\) is the number of elements in the vector. In this work, we capitalize on two recent advances. The first is the proliferation of SIMD extensions in general purpose processors. For example, the Advanced Vector Extensions (AVX2, [5]) from Intel provide 256-bit registers which can operate on vectors accessed as 8, 16, 32 or 64 bit elements. The second advancement is the huge improvements in auto-vectorizing compilers. For example, major compilers like LLVM [6], GCC and Intel C++ Compiler (ICC) have loop vectorizers and basic block vectorizers that are capable of performing many smart loop and basic block vectorization optimizations. These vectorizing optimization passes are activated by default for any optimization level higher than -O1 in these compilers. Recently, several attempts have been made towards exploiting the host SIMD extensions for accelerating the simulation of vector processors in dynamic binary translators [7], [8]. However, these attempts share the following caveats: (1) they operate at the tool specific Intermediate Representation (IR) level and require custom vector extensions to them, (2) they depend directly on the knowledge of the host architecture in the sense that they require extensions to the code generation back-end to accommodate new host SIMD instructions, and (3) they have to manage the challenges associated with SIMD width management per new architecture generations.

A. Thesis Contributions

The major contributions of this thesis are:

1) Proposes a methodology for accelerating the SIMD instruction set simulation of vector DSPs. The key advantage of this methodology is its independence from the host architecture. This independence from the host architecture is achieved by describing the semantics of each target instruction as an inlined C/C++ function that is amenable for auto-vectorization. As a result, any host with a decent auto-vectorizing C/C++ compiler can be

\(^1\)In this thesis the author worked with a VLIW processor. The terminology used is: instruction refers to the VLIW instruction and operation refers to one target instruction a function unit of an issue slot executes.
used to simulate the target binaries while benefiting from the host SIMD instructions.

2) I Propose a set of guidelines for leveraging the capabilities of state-of-the-art vectorizing compilers towards achieving (1). I show how this methodology can be applied in a processor simulation framework and in binary translators.

3) A prototype simulator tool, called Vectorizing Compiled Simulator (VCSIM), targeting commercial DSPs from the imaging domain. VCSIM signifies the novelty and feasibility of the idea in (1). Furthermore, I demonstrate the speedup obtained by applying the proposed methodology in VCSIM, that achieves a near hardware speedup leading to just 4× slow down compared to the real native execution.

B. Context

This project was conducted at Intel Corporation within the Imaging and Camera Technology Group (ICG) located in Eindhoven, Netherlands. The author was an intern in the Processor and Compiler Tools team of ICG for 9 months starting from November 2015 to July 2016. ICG produces an Image Processing Unit (IPU) for Intel SoCs. An IPU is a multi-core heterogeneous system consisting of multiple programmable imaging DSPs (IDSP), scalar control processors, and hardware accelerators. Each IDSP is based on a VLIW architecture with multiple vector issue slots targeting data parallel multimedia applications. The author worked on the instruction set of the IDSP in the 4th and 5th generations of the Intel IPU. Both these cores are 512 bit vector wide. The processor will be referred to as IDSP further in this document.

To make the development of these processor architectures easy, fast, verifiable and retargetable, ICG has an automated processor development flow for designing a wide range of VLIW DSPs. The tool flow consists of a Software Development Kit (SDK) which includes a compiler, simulator, debugger and hardware generation tools. The author worked in the team that is responsible for the simulator part of this IPU SDK.

A typical vector VLIW DSP architecture includes multiple issue slots that can either be vector or scalar as shown in Figure 1. A vector issue slot has SIMD functional units associated with the vector issue slots.

II. BACKGROUND

In this section, different simulation approaches are discussed to enable the reader to understand the following chapters.

A. Instruction Set Simulators

Instruction set simulators [4] are defined as those that accept, as input, a target binary and simulate it on the host architecture. According to [4], they can be classified into three classes: (i) interpreted, (ii) statically compiled, and (iii) dynamically compiled. In class (i), the simulator mimics the execution of the processor by performing all the operations that a processor does, i.e., fetch, decode, and execute, during the simulation runtime. This is illustrated in Figure 2. In class (ii), the simulator offloads the fetch and decode steps by doing them before the simulation run-time in the following manner. Firstly, the target binary is decoded and translated into a program that can be an intermediate representation like C/C++/compiler intermediate code. An example of the translated program is shown in Figure 3. Secondly, this translated program is compiled targeting the host architecture. Lastly, the resulting executable is ran on the host architecture to generate the simulation traces. Class (iii) represents a hybrid of (i) and (ii) in which the code generation done in class (ii) before simulator run-time is also done during the simulation run-time.

In Figures 2 and 3, we observe that the common step during simulation runtime is the execute phase (i.e., the execution of vadd). vadd is said to represent the instruction semantics. Clearly, the implementation of such semantics (e.g., vadd) is crucial to the overall simulator performance. An example implementation of vadd is shown in Figure 4.

```c
PC = start_address;
while (is_running) {
    instr = memory[PC/4]; // Fetch instruction */
    PC += 4; /* advance PC by 4 bytes */
    opcode = decode(instr); /* extract opcode */
    switch (opcode) {
        ...
        case VADD: /* vector add instruction */
            /* extract operand register indexes*/
            Ra = (instr >> 21) & 0x1F;
            Rb = (instr >> 16) & 0x1F;
            /* vector add instruction */
            RF[Rc] = vadd(RF[Ra], RF[Rb]);
            break;
        ...
        ...
    }
}
```

```c
Figure 2. Example C code showing the execution phase of an interpreted instruction set simulator. In this example, the instruction is a vector addition (vadd).
```

```c
PC = 0;
while (execute) {
    switch (PC) {
        ...
        case 8:
            /* execute add operation*/
            RF[2] = vadd(RF[3], RF[5]);
            break;
        ...
        ...
    }
}
```

```c
Figure 3. Example C code showing the execution phase of a compiled instruction set simulator. In this example, the instruction is a vector addition (vadd). Observe that the register indexes are already decoded during translation phase prior to execution.
```
Figure 1. IDSP Architecture template which is a VLIW core that can have multiple vector and scalar issue slots. A scalar issue slot consists of multiple Scalar Function Units(SFU) and a vector issue slot consists of multiple Vector Function Units(VFU). Data path of the core includes both vector and scalar memories and register files for data storage. PC is the Program Counter and SR is the Status Register.

```cpp
// W is the vector width in bytes
// S is the vector element size in bytes
// W is assumed to be always a multiple of S
template <size_t W, size_t S>
Vector<W, S> vadd(Vector<W, S> a, Vector<W, S> b) {
    Vector result;
    for (size_t i = 0; i < (W/S); i++) {
        // The [] operator is overloaded such that a[i]
        // returns the i-th element in the vector
        result[i] = a[i] + b[i];
    }
    return result;
}
```

Figure 4. An example implementation of a vector instruction semantic in C++. In this example, the instruction is a vector addition (vadd).

III. RELATED WORK

Michel et al. [7] proposed an approach for translating target SIMD instructions onto host SIMD instructions for speeding up processor simulation. The approach introduced development of IR extensions that can serve as a layer between target and host SIMD instruction sets. The authors demonstrated the feasibility of the idea by showing the IR extensions for three sets of instructions from ARM NEON to run on x86 architecture and reported a speedup of 4× with a synthetic benchmark. Compared to my approach, [7] requires: (1) extending the IR manually to cover new target SIMD instructions, and (2) updating the back-end to support new host capabilities.

Fu et al. [8] proposed vector extensions to the IR of the Tiny Code Generator of the QEMU [9] binary translator to leverage host SIMD capabilities for emulating a vector architecture. They demonstrated significant speedup for target binaries containing ARM NEON as well as IA32 instructions when emulated on x86 hosts with SSE extensions. To this end, the authors extended the IR for vector support as well as implemented a backend for x86 SSE instructions. The drawbacks of this approach compared to mine are: (1) the manual effort needed to extend the IR with vector support, and (2) the effort needed to extend the code generation phase to emit new host SIMD instructions.

In recent years, differences in the SIMD instruction and associated complexity in programming them at intrinsic level are stated as prevailing challenges in the multi core industry [10]. Introduction of a common intermediate language [11] has been proposed to tackle this problem. In order to deal with the programming complexity associated with SIMD cores, auto vectorization has been actively researched in [12]–[15]. The three most popular compilers for C/C++; GCC [16], [17], ICC, and LLVM [18], [19] have reportedly made some very good progress there. In this thesis, the author tries to take advantage of these advancements and available compiler
The DSP core description, (B) the auto-vectorizable semantics, and (C) the target binary to be simulated. These inputs are provided in the various phases of simulation depending on the simulation approach. There are two major phases for the simulator generation and execution typically, which are elaborated below.

1) Phase 1- DSP generation: This phase comprises the tool generation steps for running the instruction set simulation. For an interpreted simulator, an interpreter tool is generated whereas for a compiled simulator, a translator tool is generated as explained in Section II. DSP description file is the common input to this phase for both simulator types, which aids in the respective tool generation. The DSP description specifies the architecture of the DSP and its internal features (e.g., number of registers, number of issue slots, etc.). Typically, the DSP description is vendor-specific and thus, is outside the scope of this work. The generated tools (i.e., interpreter or translator) are compiled on a host to generate the respective tool executable, once per core. Tool source generation is the first step and compilation of tool source into tool executable is the second step in both approaches.

2) Phase 2- Simulator generation: This phase performs the simulation per target binary executable of an application which is the input to this phase. The input binary is translated into an intermediate representation in the compiled simulator, whereas the input binary is interpreted in an interpreted simulator, which forms the third step in both simulators respectively. When the interpreted simulation provides simulation results after third step, compiled simulation has an additional fourth step for compiling the translated code and running it for getting the simulation results. However, both these simulators need semantic implementation for the execution phase in simulation as illustrated in Figures 2 and 3.

The proposed methodology by the author lies in the third input (input C) to the simulators, the auto-vectorizable C/C++ semantics library linked into the final executable created during the simulation runtime. This semantic library linking takes place in step 4 of the compiled simulator and step 2 of the interpreted simulator. These are the respective steps in which the executable for the simulation runtime is generated per simulator. The semantics specify, for each instruction, the data processing actions executed by that instruction without taking into account any HW-related effects such as timing or pipelining. The key idea is to write the semantics function per vector operation in C/C++ in a way that is amenable for auto-vectorization. To this end, the author proposes leveraging the auto-vectorizing compilers for the simulator executable generation. Executing both simulations with the help of an auto-vectorizing compiler can guarantee the use of host SIMD capabilities for target vector intrinsics simulation ensuring significant performance improvement.

Automatic vectorizers have compiler passes that perform extensive code analysis, that could convert a loop (Loop Vectorizer) or straight line codes (SLP) into potential vector instructions, that are profitable in performance compared to their scalar executions. There are three major phases to the infrastructure to enable SIMD instruction simulation using host SIMD capabilities. As the compilers get smarter, it can be argued that the heavy optimizations done by going to the IR level code translation could more or less be performed by the compiler itself. Using C/C++ language as an intermediate representation for translation comes with additional advantages of readability, portability and adaptability to next generation processors.

IV. PROPOSED METHODOLOGY

In this section, the author presents a detailed description of the proposed methodology to accelerate the execution phase of vector semantics like the one illustrated in Figure 4. Furthermore, applying the proposed methodology in the context of the instruction set simulators discussed in Section II is demonstrated.

A. Overview

Figure 5 illustrates the proposed methodology and how it can be applied in the design of compiled and interpreted instruction set simulators. The methodology accepts three main inputs: (A)

Figure 5. The realization of the proposed methodology in compiled and interpreted instruction set simulators. There are two phases of simulation: DSP Generation Time and Simulation Time. Compiled simulator flow is in the left and interpreted simulator flow is in the right in each phase.
vectorizer optimization. First is the legality analysis phase that analyzes the legality of conversion of the scalar code into single vector operations. Second is the cost model analysis phase that calculates the profitability of such conversions. The last one is the code generation phase in which the best possible host vector instructions are chosen and issued by the compiler on passing the first two steps positively. The major constraints for the vectorizer to perform these steps are the challenges in dealing with the data alignment, masking for control flow instructions, non-unit stride access to memory and fixed-length nature of SIMD vectors. Therefore, it is required to aid the compiler by restructuring the naive code written by a programmer to ensure auto-vectorization.

Nevertheless, there are two problems to address while leveraging the capabilities of an auto-vectorizing compiler. (1) To get all the semantic loop nests vectorized, and (2) to find the best possible match per operation from the host side. (1) can be achieved by aiding the compiler by removing the vectorization barriers, where as achieving (2) requires additional guidance for the compiler. Especially, enabling emission of the best possible instruction can not only help in reducing the number of instructions but also the overhead of data movement between these instructions. Multimedia applications involving fixed point calculations are known to be the most difficult for compilers to vectorize and optimize, as intermediate results are most often not matching to the data width of the operands and the results. In addition, the semantics written as a loop nest will require multiple control flow statements to round the results or saturate the results to the data width of the result, which become costly to get vectorized.

In order to write such auto-vectorizable semantics, the first major step is the modeling of vector registers of the target architecture for representing operands of vector semantics for simulation as explained below in Section IV-B.

B. Modeling Vector Registers

Recall from Figure 1, a typical vector VLIW DSP architecture includes multiple issue slots that can be either vector or scalar. A vector issue slot has multiple SIMD functional units processing data parallel operations on vector elements. Vector elements are contained in vector registers and vector memory associated with the vector issue slots. A register file is typically modeled as an array of integers whose width is the vector width of the target and size is the capacity of the register file. For instance, a vector register file with vector width 512 and capacity 32 would be modeled as,

```c
struct VectorRegister {
  uint32_t data_32[vec_width/32];
  uint16_t* data_16 = reinterpret_cast<uint16_t*>(data_32);
  uint8_t* data_8  = reinterpret_cast<uint8_t*>(data_32);
  static const size_t L_8  = vec_width/8;
  static const size_t L_16 = vec_width/16;
  static const size_t L_32 = vec_width/32;
};
```

Figure 6. Vector Register Class implemented to model vector registers of target architecture, that supports any vector width and 8,16 and 32 SIMD ways.

Assignment operator are added in the C++ class which are not shown in the code snippet.

The Intel IPU IDSP architecture used for the experimental evaluation in Section VI supports a 512 bit wide vector ISA and therefore, a vector register for the architecture is defined as a datatype below.

templated ISAX<512>::VectorRegister vr_t;

To guarantee vectorization, the vector class model proved to be sufficient as it passed vectorization of some simple basic arithmetic operations like addition. However, the class model itself is not enough to guarantee vectorization of other complex semantics. To guarantee vectorization, a detailed analysis has been performed on some test operations and vectorization is achieved by following certain additional steps further. The author approached the vectorization problem by taking two iterative approaches as shown in Figure 7. First one is towards ensuring the vectorization of each semantic, by aiding the compiler in identifying and removing the barriers for the vectorization possibility of a loop-nest, and the second one is towards achieving the most suitable instruction from the host side per vectorized operation. These two steps are detailed in the Sections IV-C and IV-D respectively.

C. Refactoring code to remove vectorization barriers

A naive semantic loop nest written for an operation is initially passed through any vectorizing compiler. All the vectorizing compilers today provide a nice feature; a detailed report on the vectorization step performed by the optimization pass which includes a status remark stating if the loop is vectorized or not. If the status is unsuccessful, programmer needs to keep refactoring the code until the loop nest gets vectorized as illustrated in iteration 1 in Figure 7. The report also provides the programmer with reasonable hints towards the cause of vectorization failures. Some of the major refactoring steps that the author noted while working through this iteration process are the following:\footnote{The examples shown are explained as per the report of the LLVM 3.8 compiler.}

```c
template<size_t vec_width> struct ISA {
  public:
    struct VectorRegister {
      public:
        uint32_t data_32[vec_width/32];
        uint16_t* data_16 = reinterpret_cast<uint16_t*>(data_32);
        uint8_t* data_8  = reinterpret_cast<uint8_t*>(data_32);
      static const size_t L_8  = vec_width/8;
      static const size_t L_16 = vec_width/16;
      static const size_t L_32 = vec_width/32;
    };
};
```
The problem lies in the legality phase of the vectorization. Therefore, it is advised to write the if-else cases in simple blocks with ternary operators that can be potentially turned into a select operator on the host. This situation arises a lot in many of the semantics of the target architecture. An example of such a naive code not being vectorized is the vec_cond_add() operation as shown in Figure 8. This operation produces an output vector \( R \) whose elements are the sum of the two input vectors \( A \) and \( B \), if a condition element in another input vector \( C \) in the respective index is true. If the condition element is false, result vector element is the element of the input vector \( A \). This is implemented as an iteration in which, \( A \) element is chosen when \( C \) element is zero and sum of the \( A \) and \( B \) elements is chosen otherwise. Note that the data type of a vector is the \( vr_t \) data type created from the vector class model as declared in Section IV-B.

The vectorization report for the code snippet compilation stated that the compiler could not convert the if cases into a select operation. This means that the compiler tried to convert the if case into a select operation but failed due to some reason. The problem lies in the legality phase of the vectorization.

1) Control flow: An auto-vectorizer chooses a select operation (C language ternary operator) for resolving if cases. Therefore, it is advised to write the if-else cases in simple blocks with ternary operators that can be potentially turned into a select operator on the host. This situation arises a lot in many of the semantics of the target architecture. An example of such a naive code not being vectorized is the vec_cond_add() operation as shown in Figure 8. This operation produces an output vector \( R \) whose elements are the sum of the two input vectors \( A \) and \( B \), if a condition element in another input vector \( C \) in the respective index is true. If the condition element is false, result vector element is the element of the input vector \( A \). This is implemented as an iteration in which, \( A \) element is chosen when \( C \) element is zero and sum of the \( A \) and \( B \) elements is chosen otherwise. Note that the data type of a vector is the \( vr_t \) data type created from the vector class model as declared in Section IV-B.

The vectorization report for the code snippet compilation stated that the compiler could not convert the if cases into a select operation. This means that the compiler tried to convert the if case into a select operation but failed due to some reason. The problem lies in the legality phase of the vectorization.
this extent, vectorized code is taken through a quality check iteration process shown as iteration step (2) in Figure 7. In this phase, the assembly emitted for the vectorized code is thoroughly examined, along with possible mapping instructions from the host intrinsics. Most of the time, the code generated is bound to be a non-optimal solution or contains extra overhead in restructuring the data, which could be avoided by providing additional aid to the compiler. Some of the major considerations taken for minimizing such overhead and emitting direct mapped instruction from the host are stated below.

1) **Data width mismatches**: Passing the right operand data width helps the compiler in choosing the matching operation from the intrinsic subset for a particular data width. For instance, passing an add operation with 8 bit element chooses the equivalent 8 bit addition operation from the host intrinsics, 16 bit operands chooses the 16 bit equivalent and so on. If a function call is made by mismatching data types, compiler will unnecessarily do up-conversion of the vector and then down-conversion while updating the results. In the case of not finding the right match, as in Intel KNL\(^3\) architecture that supports only 32 or 64 bit operands, compiler does the up-conversion to find the most suitable operation for the previous cases. This case arises mainly in the case of operations involving immediate or constant operand.

For instance, the naive implementation of the vec\_add\_c() operation, in which a constant is added to all the elements of a vector, is shown in Figure 12. The assembly emitted for this operation by an ICC compiler includes a vpbroa operation that broadcasts the constant into a single vector, but the compiler generates 32 bit vector of C and hence up converts the equivalent 8 bit addition operation from the host intrinsics, 16 bit operands chooses the 16 bit equivalent and so on. If a function call is made by mismatching data types, compiler will unnecessarily do up-conversion of the vector and then down-conversion while updating the results. In the case of not finding the right match, as in Intel KNL\(^3\) architecture that supports only 32 or 64 bit operands, compiler does the up-conversion to find the most suitable operation for the previous cases. This case arises mainly in the case of operations involving immediate or constant operand.

2) **Compiler Idiom Identification for Direct Mapping**: If a direct match for an intrinsic exists in the host architecture, that is the best possible solution to be achieved per operation. Compilers have a scalar pattern recognition code attached with a vector instruction, to be recognized as potential vector instruction, if this pattern is applied in loops. This is called a compiler idiom. ICC compiler has a set of such idioms for multimedia extensions [20]. On identifying such an idiom of the compiler, the best mapping of the operation can be easily achieved. One such idiom is that of vec\_abssat

\(^3\)Knights Landing (KNL): 2\(^{nd}\) generation Intel Xeon Phi Processor.
operation that produces the absolute value of the data operand by saturating the minimum value to the maximum value that can be represented by the vector width. Finding such a solution can be extremely beneficial, as the resulting speedup comes not only by vectorization but also on identifying a better solution by leveraging the already identified patterns. The naive implementation of the `vec_abssat` operation is done as shown in Figure 14 which emitted multiple vector operations instead of the best mapping host intrinsic. One example to illustrate this is a `vec_sub_abssat()` operation.

This operation performs a subtraction operation and returns the saturated absolute value of the result. One approach to write the semantics for the same is to realize two loop nests, one for subtraction and the other for calculation of the saturated absolute value. Another approach is to combine these loops into one, by leveraging the already identified patterns. The first approach emits the required operations per loop but stores the data back and forth in memory for the two loops. Whereas, with the second approach, compiler easily identifies the dependency between the operations, which then emits two best possible instructions avoiding additional moves per loop. In this case, the instructions emitted are `vpsubw` and `vpmaddw`. The semantics code for the same is written as shown in Figure 16.

4) **Variable Substitution:** In the case of ternary operations, using intermediate variables for accessing memory makes it easier for the compiler to emit direct vector instructions and avoid a lot of overhead, especially for ICC and GCC. One such example is the `vec_cond_add()` operation discussed before in Section IV. The vectorized version as shown in Figure 9 emits a lot of overhead code with minimal performance enhancement to the scalar version for both ICC and GCC. On introducing intermediate variables for memory accesses per vector element, as shown in Figure 17, this issue is resolved. The problem lies in the compiler not being able to determine the legality of accessing the un-taken branch element for calculation, for which extra code is generated to verify the process.

**V. Vectorizing Compiled Simulator (VCSIM)**

Vectorizing Compiled Simulator (VCSIM), is the new instruction set simulator created for the ICC cores that supports idioms recognition in the case of AVX-512 SKX\(^4\) instructions.

Because the last approach is the best, the compiler should try to use this approach. However, the compiler is not able to do this all the time. In the case when no direct mapping operations can be found on the host side, it is possible to create a combination of identified patterns to achieve the best solution. One example to illustrate this is a `vec_sub_abssat()` operation. This operation performs a subtraction operation and returns the saturated absolute value of the result. One approach to write the semantics for the same is to realize two loop nests, one for subtraction and the other for calculation of the saturated absolute value. Another approach is to combine these loops into one, by leveraging the already identified patterns. The first approach emits the required operations per loop but stores the data back and forth in memory for the two loops. Whereas, with the second approach, compiler easily identifies the dependency between the operations, which then emits two best possible instructions avoiding additional moves per loop. In this case, the instructions emitted are `vpsubw` and `vpmaddw`. The semantics code for the same is written as shown in Figure 16.

\(^4\)Intel microarchitecture Skylake
which lacks coverage on the binary verification which converts the intermediate code into a C during the translation period to save the simulation time. Most needed during the simulation. These steps could be performed simultaneously by the simulator perform part of the architecture specific decoding of the semantics of the execution of instructions in the compiled model from it. The simulation compiler here is the csimgen. This intermediate file is taken as an input by the assembler which outputs the final binary executable for the target architecture. This intermediate file is mapped and scheduled intermediate representation (IR) for the VLIW compiler comprises of a set of tools that produce optimized ICG as well as the existing simulation flow. The in-house simulator here is the csimgen which converts the intermediate code into a C code. In contrast, the interpreted simulator takes the binary generated by the assembler as the simulation input and performs the traditional interpreted simulation as explained in Section II.

A. Problem

The compiled simulator of the ICG is compiler-IR dependent which lacks coverage on the binary verification. Additionally, the semantics of the execution of instructions in the compiled simulator perform part of the architecture specific decoding needed during the simulation. These steps could be performed during the translation period to save the simulation time. Most of the semantic function calls have multiple control flow statements that check the behavior of the operation execution according to the functional units. These semantics are external library calls and not in-lined. This costs a lot in terms of performance due to the repeated stack access for every semantic invocation.

On the other hand, the interpreted simulator uses the target binary as input solving the issue of binary verification, however, it is the slowest approach for simulation as seen in Section III. The performance issues associated with semantic function calls as explained above also hold in this case as both simulators use the same set of semantic functions. Considering the limitations of both approaches, a new simulator was proposed that could have the benefits of both and additionally, could be used to accommodate auto-vectorization of vector semantics for performance acceleration. The requirements for the new tool are stated below in Section V-C.

C. Requirements

The following requirements were identified by the simulator team towards a new processor simulator, that ultimately achieve the major goal of performance but also try to address a few other existing issues in the current solutions explained above in Section V-B.

1) Performance: Simulator should be designed towards achieving better performance compared to the existing solutions.
2) Instruction Set Simulation: Simulation should take binary as input to verify the firmware binary of the target. That means, correct simulation shall match execution on real hardware.
3) C/C++ semantics: Simulator execution should take C/C++ semantics for execution of operations. This will enable integration of the auto-vectorization strategy explained in Section IV
4) Decoupling from the compiler flow: Simulator should not depend on the compiler framework.
5) Cycle Accuracy: Simulator should give correct number of execution cycles on the processor.
6) Bit accurate: Simulator should be bit accurate.
7) Retargetable: Simulator should be able to support any core designed from ICG processor template.

D. Literature Review

A great deal of research work has been focusing on improving the speed of the interpreted simulation and have

<table>
<thead>
<tr>
<th>Macros used for compiler idioms</th>
<th>Compiler Idiom Macro Expansion</th>
<th>Intel IPU4 DSP Target SIMD intrinsic</th>
<th>Intel AVX-512 Host SIMD intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS (x)</td>
<td>(x) &gt; 0 ? (x) : (-x)</td>
<td>vec_abs saturation</td>
<td>vpabsw</td>
</tr>
<tr>
<td>MIN (x)</td>
<td>(x) &lt; (y) ? (x) : (y)</td>
<td>vec_min</td>
<td>vpmintsw</td>
</tr>
<tr>
<td>MAX (x)</td>
<td>(x) &gt; (y) ? (x) : (y)</td>
<td>vec_max</td>
<td>vpmaxsw</td>
</tr>
<tr>
<td>ADDSAT (x)</td>
<td>PACK_SATURATE_SIGNED (x + y)</td>
<td>vec_addsat</td>
<td>vpadw</td>
</tr>
<tr>
<td>SUBSAT (x)</td>
<td>PACK_SATURATE_SIGNED (x - y)</td>
<td>vec_subsat</td>
<td>vpsubw</td>
</tr>
<tr>
<td>PACK_SATURATE_UNSIGNED (x)</td>
<td>MIN_MAX ((x) &lt; -32768) : 32767)</td>
<td>intermediate used</td>
<td></td>
</tr>
</tbody>
</table>

Figure 17. Vectorized vec_cond_add() operation optimized for better performance by introducing variables for all the array accesses per vector element.

<table>
<thead>
<tr>
<th>void vec mux(vr_t &amp;R, vr_t &amp;A, vr_t &amp;B, vr_t &amp;C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
</tr>
<tr>
<td>uint16_t a, b, c;</td>
</tr>
<tr>
<td>for (int i=0; i&lt;R.vector_elements_16; i++) {</td>
</tr>
<tr>
<td>a = A.data_16[i];</td>
</tr>
<tr>
<td>b = B.data_16[i];</td>
</tr>
<tr>
<td>c = C.data_16[i];</td>
</tr>
<tr>
<td>R.data_16[i] = (c == 0)? a + b : a;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

The methodology proposed in Section IV. In the remainder of this chapter, an overview of the existing simulator tool sets are discussed by identifying the problems associated with them that led to the proposal for a new simulator.

A. Existing Simulators in ICG

There are two existing simulators for the ICG cores. One is a compiled simulator and the other is an interpreted simulator. Figure 18 shows the compiler flow of the processors of ICG as well as the existing simulation flow. The in-house compiler comprises of a set of tools that produce optimized and scheduled intermediate representation file (.hive.s) from the application source file. This .s file is target program mapped and scheduled to the target architecture. The existing compiled simulator takes this scheduled code and generates an executable simulation model from it. The simulation compiler here is the csimgen that led to the proposal for a new simulator.

### Table I

<table>
<thead>
<tr>
<th>ICC Compiler Idioms tried for the target intrinsics and corresponding host intrinsics emitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros used for compiler idioms</td>
</tr>
<tr>
<td>---------------------------------</td>
</tr>
<tr>
<td>ABS (x)</td>
</tr>
<tr>
<td>MIN (x)</td>
</tr>
<tr>
<td>MAX (x)</td>
</tr>
<tr>
<td>ADDSAT (x)</td>
</tr>
<tr>
<td>SUBSAT (x)</td>
</tr>
<tr>
<td>PACK_SATURATE_UNSIGNED (x)</td>
</tr>
</tbody>
</table>
resulted in a large varieties of simulators [21]. In order to get rid of the costly fetch/decode process in software, all new approaches focus on preforming the decode step in a translation phase prior to execution. Compiled simulators and dynamic translators fall into that category.

Traditional compiled simulation [22] takes the target binary as input although other approaches have been tried. [23] demonstrates the idea of using object files as input in order to reduce the compile time overhead caused by the translation. Yet another approach is to take compiler intermediate output as input. At the same time, there have been quite some effort taken to accelerate the compiled simulation as well. In the first version [24], translations were performed to C using macros for instruction execution. Similarly, FSCS [25] translates to C which also performs control flow related optimizations during translation to improve simulation performance. Reportedly, the challenge in the traditional compiler was dealing with indirect branches [22] which has been solved by heuristics based approach in SyntSim [26]. SyntSim also gives a detailed approach in optimizing compiled simulator by using function in-lining. On the other hand, instead of translating to C code, [27] demonstrated the idea of translating to intermediate low level code for further acceleration. Such low level IR code helps in performing heavy optimizations and gives more predictability about the behavior of program than using C. However, in a follow up study, [28] reviewed the approach of compiled simulation extensively and reported the major limitations posed by this approach. Some of them are flexibility loss in handling real time operating system, dependency on source code availability and inability in handling self modifiable code.

Considering the restrictiveness of compiled simulators, many authors tried combining the interpreted and compiled simulation to keep the goodness of both; performance of compiled simulation and flexibility of interpreted simulation. Just in time cache compiled simulation [28] was presented to be highly flexible and re-targetable. However, performance hugely depends on the complexity involved in decoding. The idea is to cache the decoding done in order to reuse the translated code. Moreover, cache size increases with pipeline stages in a complex architecture like VLIW. [29] introduced instruction-set compiled technique (IS-CS) which performs decoding at compile time as in compiled simulator but does re-decoding at runtime if an instruction is modified at runtime. Meanwhile, with the advent of MPSoCs, binary translation caught it’s attention as a simulation technique.

In the recent years, dynamic binary translators have been identified as the fastest instruction set simulation technique [21]. Just-in-Time binary translated simulator initially performs interpretatively and simultaneously profiles the simulation identifying the basic blocks that are repeatedly executed [31]. Such basic blocks are called hot-spots and are identified for binary translation. Thus the performance is achieved not loosing flexibility. This has been a very active area of research today [32] [33]. Just-in-Time compilers are also used in this scenario [34].

Binary translation and processor simulation have a clear relationship. Essentially, both of them try to run the code of one platform on another platform. The most effective approach for binary translation is direct mapping from one ISA to another which unfortunately makes it target dependent. A clear distinction between existing binary translators has been discussed in [35]. Static binary translators and static compiled simulators are so closely knitted in principle [36], as both of them try to translate the binary during compile time to later perform the execution on the host. In the recent literature, there have been discussions on both static [37] and dynamic binary translators (DBT) for VLIW architectures [38]. The authors of [38] reported the advantage of the DBT method of VLIW over interpreted simulation but did not speak of how it compared with static translation or compiled simulation approach.

1) Conclusion: For the target IDSP, lack of flexibility of compiled simulation approach does not raise any potential issue as there is no case of self modifying code. Therefore, the compiled simulation approach is valid to adopt for the use case. At the same time, the author proposes using C/C++ code as the IR as in Syntsim [26] in which all C/C++ semantics functions are inlined. It can be argued that with the advent of
compiler optimizations today, going to low level C/C++ as intermediate representation would not hamper performance to a large extent. The trade-off involved in making this decision are mainly between performance, maintainability, development time and readability.

E. Design Strategy

Considering the requirements in Section V-C, the author lists below the design decisions taken towards satisfying the requirements.

- Compiled simulation approach for simulation is identified as the best approach for performance as specified in Section III. This is the initial step towards achieving the requirement (1).
- Simulator is designed as an instruction set simulator (for binary verification and decoupling from compiler flow). This fulfills the requirements (2) and (4).
- Simulator calls C++ semantic functions for execution. This meets the requirement (3).
- Data widths, pipeline stages and delay slots are modeled accurately as the data path and control flow path of the target architecture. This enables meeting the requirements (5) and (6).
- The translator tool for the compiled simulator is generated automatically from an IDSP core specification XML file, which models the core resources and pipeline models per target. This satisfies the retargetability requirement (7).

The proposed tool is called Vectorizing Compiled Simulator (VCSIM), as shown in Figure 19. VCSIM is a retargetable compiled simulator that takes firmware binary as simulation input and takes C/C++ semantics for the execution of semantics which is amenable or auto-vectorization by vectorizing compilers.

![Diagram](image)

Figure 19. Proposed simulator Vectorising Compiled Simulator that performs instruction set simulation taking compiled simulation approach. This simulator takes C/C++ semantics for the execution of operations.

F. Tool Overview

The deployment view of the VCSIM tool flow is shown in Figure 20. The entire tool flow is divided into two phases.

1) Phase 1: The first phase of the tool shows the generation of the compiled simulator utilities which is the same as the DSP generation phase for simulation shown in Figure 5. The tool designed for this is the core_parser tool, written in Python. The input to the tool is the processor specification file that gives the description of the entire core architecture including the ISA. This tool generates the translator tool for the compiled simulator as well as the simulation models and cell models required for the simulation of a core.

The translator tool for VCSIM takes target binary as input and translates the same into C/C++ code. The translation phase of compiled simulation performs the fetching and decoding steps of every instruction, for which a disassembler of the target binary is essential. For this, a disassembler generation library has been used that has routines for disassembling the binary. The disassembler library and the translator executable are linked together to generate the final translator tool executable per core, vcsim-gen. The functionality of the translator tool is elaborated in Section V-G.

Apart from the translator, parser tool also generates storage models and simulation models required for simulation. Storage models include the data structures representing registers and memories where as simulation models include the utilities for processor execution phase: handling pipelines, jumps, delay slots, termination and operation execution modules. These models are detailed in the Sections V-H1 and V-H2 respectively.

2) Phase 2: The second phase in the tool flow shows the compiled simulation process executed per application, which is the same as the simulation phase explained in Figure 5. The target binary to be simulated is passed as an input to the vcsim-gen of the core that generates the simulator code in C++. Simulator tool inlines the function calls per execution of operations, which are included as a header file. This header file contains the vector semantics, specified as loop nests targeted for auto-vectorization. This semantics header file is one of the major contributions of the thesis as stated in Section I.

G. Translation phase

The translator tool reads the target binary as an input and disassembles the ELF file with the help of the disassembler tool. This disassembled ELF is stored in a program buffer array allocated dynamically. After this process, the code generation phase begins where each instruction is fetched and decoded one by one and converted to the corresponding C/C++ semantic function calls in the generated simulator source code. The translation process is depicted in a UML activity diagram shown in Figure 21.

Instruction translation has multiple architecture related concerns to be addressed for correct simulation. As mentioned before in Section I-B and illustrated in Figure 1, the architecture is a VLIW core that supports both vector and scalar operations. The operations in a VLIW are statically scheduled during the compile time by a scheduler. The major components of the instruction translation step are discussed in detail below.

1) Fetch and decode: This step performs the fetching and decoding per issue slot operation of the VLIW. In a VLIW architecture, each instruction is composed of multiple operations belonging to each issue slot. This is illustrated for a VLIW architecture with four issue slots in Figure 22. The disassembler [39] tool has functions that help in decoding the operations per issue slot one by one. The required data in the encoded instruction is extracted by various function
calls targeting each required element encoded in the opcode. The elements of an operation encoding include opcode of the operation, register indices for input and output data and bus multiplexer values. In the IDSP architecture, instructions consume the same data width per issue slot that makes stepping through operations a lot easier. The simulation of a single VLIW instruction becomes several function calls to semantics of each operation executed sequentially. Some of the major steps to be taken care of during the fetch and decode process are elaborated below.

a) Handling VLIW instructions: In a VLIW architecture, in every cycle, a new operation can be executed on every issue slot. In simulation, each of these operations are fetched and decoded sequentially. As register files and memory can be shared between issue slots, Read After Write (RAW) dependency is possible in this sequential simulation approach. To solve such dependency errors, all calculations per issue slot are performed by reading registers and then, the execution results are stored in some temporary variables that store the output per issue slot output port. One issue slot output port can carry only one output value per cycle, facilitating atomic access of these variables used. Once done, the copying from temporary variables to registers is performed per issue slot after checking the bus valid bits.

b) Handling static pipelining: Many operations in the ICG cores are deeply pipelined. The architecture features time stationary encoding, where the bits representing operands of the pipelined operation are distributed over multiple cycles. This means, operand register indexes, immediate values and bus selection bits are issued in the instruction in which they are actually used. The total stages of the pipelined operation depends on the latency of the functional unit on which it is executed. For instance, suppose an arithmetic unit has a latency of 1 for its two inputs and latency of 2 for its output. This means, the opcode of an operation issued in cycle 1, gets its two operand data in cycle 2 and it emits its output in cycle 3. Therefore, an operation that takes multiple cycles will have multiple sub-operations corresponding to each stage. This information about the total stages of the opcode issued is not part of the instruction. Therefore, these sub-operations have to be created and issued by keeping track of the instruction while decoding. For instance, a pipelined add operation performed by the example arithmetic unit is created as shown in Figure 23.

To this end, pipeline buffers are introduced per issue slot that can keep track of the stages of the pipelined operation once issued. The pipeline buffer is modeled as a data structure
that has variables per input port and output port of the issue slot to store the input data and output data corresponding to an issued operation. The size of the pipeline buffer of an issue slot becomes the latency of the longest pipeline operation. When a pipelined operation is fetched, a pipeline initialization step is performed that allocates a buffer per pipeline operation statically. Pipeline operation execution function is called as soon as all the input arguments are received and the result of the execution stage is stored in the buffer variable for output. This process of pipeline handling is detailed with the same add example in Appendix A.1.

2) Register Write: As stated in Section V-G, results of the operation executions are passed onto temporary variables to eliminate the RAW dependency issue. In this step of translation, the register indices to write the results per issue slot have to be determined. For this, we need to look into the datapath of the processor as shown in Figure 24. Issue slot output ports are connected to buses which can be shared between issue slots. However, a bus can only be accessed by an issue slot port in every cycle which is encoded as a bus selection bit in the instruction in every cycle. Such a multiplexed bus is the bus 1 in the Figure 24, that is shared between output 1 of both issue slots 1 and 2. At the same time, a register file can read from multiple buses, one bus in every cycle which is determined by another multiplexer as well. In the figure, register file 1 can take an input from buses 1 or 2 and register file 2 can take an input from buses 3 and 4. Moreover, a bus can write into multiple register files in a cycle, as the case of bus 1 in the figure. These cases are considered for updating the results into the register files in every cycle. This process is performed by checking the set of valid bits that choose, buses and registers that are encoded in the instruction. This step basically identifies the register to write to per issue slot and performs the copying from the issue slot output port temporary variable to the right register. Appendix A.2 demonstrates this register write process with the help of an example.

H. Simulation Phase

This phase involves the real execution of operations that are fetched, decoded and translated into C++ semantic function calls per operation in every cycle. Various models that are generated by the core_parser tool in Figure 20 for the simulation phase are explained below.

1) Storage models: In order to model the state of processor during simulation, registers and memories of the IDSP, as shown in Figure 1, have to be modeled accurately and efficiently. In VCSIM, they are modeled as described below.
a) Registers: A scalar register file is modeled as a static array, `uint(width)_t RF[(capacity)]`, in which data type of array becomes the width of a register and the size of the array becomes the capacity of the register file. The register file widths are assumed to be multiples of 8 that easily matches the C standards. It is important to make sure that the registers map directly to the host registers available for ensuring performance. These array accesses are expected to be identified for repeated accesses by the compiler which will reuse the data by using registers and lowest levels of data cache. The modeling of vector registers has been already discussed in Section IV-B.

b) Memory: IDSP has separate data memories and program memory. As there are no instructions that operate on the program memory directly, there is no need to explicitly simulate the program memory. There are two data memories for an IDSP, (1) data memory for the scalar issue slots which has data width of scalar register files and (2) vector memory for the vector issue slots which accesses a vector at a time. The data memories can be simulated by means of a single array with data indices used as memory access addresses. The access of various data widths can be performed by manipulating the indices as required. Vector memory access is facilitated by using memcpy/memset functions available in the C standard.

2) Simulation models: During the translation phase performed by the vcsim-gen tool, all the instructions of the input application binary are fetched and decoded into C/C++ function calls as described in Section V-G. Therefore, the simulator tool executable generated per application performs the execution of operations one by one until the processor terminates. The state machine of the simulation is shown in Figure 25. Initial state is the execution state controlled by the state variable execute set to 1. When the program terminates, the execution state is transited to idle state and the program returns to main.

a) Control flow: Most semantic functions simply perform a computation and modify the contents of registers, which is easily dealt with by passing the correct parameters. However, the semantic functions that modify the control flow of the program require more careful consideration. Especially to the fact that the ISP utilizes delay slots, i.e. the first few instructions right after the branching, more instructions are executed even if the branch is taken before the real jump happens. This delay is the latency of the program counter register. For this reason, we have to develop some sort of mechanism that delays the execution of control flow changes. This has been done with the help of jump buffers that get set when jump operation is issued, has a counter that keeps counting until the delay slot count is set, and then update the program counter at the end of the delay slot. The size of the jump buffer is equal to the delay latency of the program counter plus 1. The same goes for the execution of the operation that terminates the program which is delayed by the latency of the status register. Appendix A.3 demonstrates this jump handling process with the help of an example and activity diagram.

Limitation: The downside of taking the target binary as simulation input is the lack of knowledge about the potential entry points in the program execution. Computed branches have target destination of control flow determined by values updated in the operand register during runtime. This case arises as a result of switch statements and function pointers in a program. As any program counter can be a potential entry point, this implies having a label for every program
counter. This can be implemented as a switch case or goto labels for all instructions in the program which can impose major limitations for compiler optimizations. Compiler will encounter a new basic block per operation in every cycle, which can limit the amount of optimizations. Due to lack of time during the thesis, further work has not been done for solving the computed branch performance issue. Appendix A.4 illustrates with an example, the cases of direct branch and indirect branches that can occur in a program.

Figure 25. State machine of the processor simulator

b) **Execution state::** The execution state of simulation goes through an iteration of activities as shown in Figure 26. In every cycle, operations are executed as in-lined function calls to the operation semantics. Once finished, the register file writing per issue slot gets performed as copy instructions. The program counter is then incremented by 1. Before jumping to the incremented program counter, jump or termination operation issue is checked. If any of them met the delay slot latency, the program counter is updated to the respective address and jump happens to the respective entry point.

I. **Verification**

In order to evaluate the VCSIM tool, there are two major requirements. First one is to verify the functional correctness of the simulation and the second one is to verify the reliability of the simulator tool by running test cases that cover maximum semantics.

1) **Functional correctness:** Verification of simulation correctness was performed by comparing the final memory dump of the simulator with that of the existing simulators. Simulator writes the memory footprint of the data memory into a binary file after simulation, which is also generated by the existing simulators. Simulation correctness is verified by comparing both these files. Moreover, cycle accuracy of the simulator is also validated by comparing with the cycle counts emitted by the existing simulators.

In addition, a simulation trace is printed into a trace file that gives the register values and memory access values read and written during the execution of every instruction including addresses of every memory operation. Such a trace file is generated by existing simulators as well. Cross checking with the trace printing also helps in validating and debugging the simulation in each instruction execution step. This has been the major debugging technique used for validating the VCSIM tool.

2) **Semantics correctness:** In order to validate the simulation functionality, it is essential to run the simulator for various applications that can potentially validate all the operation semantics. The major functionalities include memory access operations, control flow instructions, arithmetic and logical computations that have to be validated. For this, a set of applications from the academic benchmark suite Polybench [40] was used. An automated infrastructure is built to test all these applications and generate a test report file.

VI. **Evaluation**

In this section, a detailed evaluation of the proposed methodology is presented. The author demonstrates using real DSPs and real workloads, the advantages of leveraging the methodology as claimed in Section IV.

A. **Experimental Setup**

The evaluation experiments are performed in two stages. First stage is the micro-benchmarking experiments performed per target vector operation based on small testing kernels to evaluate the applicability of the proposed methodology. The major benchmark used for evaluation in stage 1 is:
1) A set of 40 operations from the target intrinsics. These operations are the most used vector operations in real noise reduction filters running on Intel IPU4 IDSP. Each operation in this category is executed 1 billion times. To get a fair speedup on the tests, all the inputs for calculations were totally provided during runtime in order to avoid the compiler optimizing the work away during compile time.

Second stage involves macro-benchmarking experiments performed with real firmware of IDSPs to evaluate the overall performance of the proposed methodology in simulation. Moreover, the quality and performance analysis of the simulator tool, VCSIM, is presented.

The major benchmarks used for evaluation in stage 2 are:

1) A set of DSPs and VLIW cores from Intel ICG as shown in Table II. In total, 8 cores from ICG are tested that include three commercial cores: two IDSPs (IPU4 and IPU5) and one scalar control processor.

2) A matrix multiplication kernel which performs multiplication of square matrices of size $32 \times 32$.

3) A $5 \times 5$ window sized sobel filter operating on images of size $512 \times 512$.

The matrix multiplication and sobel filter applications are manually written to use the SIMD intrinsics of the IDSP directly.

For running the benchmarks in both stages of evaluation, various host architectures are used as shown in the Table III.

B. Micro-benchmarking

This section presents the first stage of evaluation that consolidates the experiments and results associated with the claims on the proposed methodology.

### Table II

<table>
<thead>
<tr>
<th>ICG Core</th>
<th>Number of Issueslots</th>
<th>Instruction Word Size</th>
<th>Number of Register Files</th>
<th>Vector Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tad</td>
<td>1</td>
<td>32</td>
<td>3</td>
<td>...³</td>
</tr>
<tr>
<td>Tadray</td>
<td>2</td>
<td>64</td>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>Pearl</td>
<td>2</td>
<td>61</td>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>Pearl Ray</td>
<td>3</td>
<td>111</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>Avispa Demo 1</td>
<td>26</td>
<td>900</td>
<td>35</td>
<td>...</td>
</tr>
<tr>
<td>SP3</td>
<td>*</td>
<td>*</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>IDSP3</td>
<td>*</td>
<td>+</td>
<td>+</td>
<td>512</td>
</tr>
<tr>
<td>IDSP5</td>
<td>*</td>
<td>+</td>
<td>+</td>
<td>512</td>
</tr>
</tbody>
</table>

### Table III

<table>
<thead>
<tr>
<th>SIMD Intrinsics</th>
<th>Vector width</th>
<th>Vendor</th>
<th>Compilers used</th>
<th>Total Operations Tested</th>
<th>Vectorized</th>
<th>Not Vectorized</th>
<th>Not Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEON</td>
<td>64</td>
<td>ARM</td>
<td>LLVM 3.9</td>
<td>40</td>
<td>35</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>SSE4.2</td>
<td>128</td>
<td>ARM</td>
<td>LLVM 3.8.0</td>
<td>60</td>
<td>52</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>AVX2</td>
<td>256</td>
<td>Intel</td>
<td>GCC 6.1.1</td>
<td>70</td>
<td>60</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>AVX512-SKX</td>
<td>512</td>
<td>ICC</td>
<td>ICC 16.0.3</td>
<td>80</td>
<td>70</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

### Table IV

## 1) Vectorization Results: The vectorization results of the top operations tested from the noise reduction filters are shown in Table IV. The table shows the total number of operations tested for each filter. There are only two operations in each filter that fail auto-vectorization. The non-vectorized operations are the same in both filters, which are two variants of vec_shuffle() operation. This case is illustrated in an example code in Figure 27. vec_shuffle() operation fills the result vector $B$ by elements of $A$ at indices pointed by the elements of $B$. These indices can be randomly ordered which are determined by the data filled in vector $B$ during runtime. This poses a major challenge to the vectorizer due to the random data access from noncontiguous points in memory. While this case fails with both GCC and LLVM, ICC can vectorize the same by enforcing vectorization with the use of compiler pragma, `#pragma simd`, before the loop nest. However, the vectorization benefit is almost negligible, giving a speedup of only 1.06 over the non-vectorized version. On this forced vectorization, ICC chooses the shuffle intrinsic available for 32 bit vector elements in Intel AVX2 as Intel intrinsics do not support shuffle operation for 8 or 16 bit data so far.

### 2) Performance evaluation of operation classes: Operations chosen for the vectorization tests are classified into sub classes of operations as shown in Table V. Moreover, the table shows the operations that achieved the minimum and maximum speedup on vectorization. Performance improvement per operation subclasses in the Table V is shown in Figure 28. The figure shows the average, minimum and maximum speedup achieved per operation subclass.

On average, vectorization of semantic functions provides a speedup of $4 \times$ per target operation on emitting AVX2 instructions. Intuitively, one would expect a speedup of around $16 \times$ in this case as 16 scalar operations in a for loop run as one host vector intrinsic after vectorization. Practically, there are

```
void vec_shuffle(vr_t &R, const vr_t &A, const vr_t &B)
{
    uint16_t r, a, b;
    for (int i=0; i<R.L_16; i++) {
        R.data_16[i] = A.data_16[B.data_16[i]];
    }
}
```

Figure 27. vec_shuffle() operation semantics that failed to get auto-vectorized by compiler

### Table V

## 1) Vectorization Results: The vectorization results of the top operations tested from the noise reduction filters are shown in Table V. The table shows the total number of operations tested for each filter. There are only two operations in each filter that fail auto-vectorization. The non-vectorized operations are the same in both filters, which are two variants of vec_shuffle() operation. This case is illustrated in an example code in Figure 27. vec_shuffle() operation fills the result vector $B$ by elements of $A$ at indices pointed by the elements of $B$. These indices can be randomly ordered which are determined by the data filled in vector $B$ during runtime. This poses a major challenge to the vectorizer due to the random data access from noncontiguous points in memory. While this case fails with both GCC and LLVM, ICC can vectorize the same by enforcing vectorization with the use of compiler pragma, `#pragma simd`, before the loop nest. However, the vectorization benefit is almost negligible, giving a speedup of only 1.06 over the non-vectorized version. On this forced vectorization, ICC chooses the shuffle intrinsic available for 32 bit vector elements in Intel AVX2 as Intel intrinsics do not support shuffle operation for 8 or 16 bit data so far.

### 2) Performance evaluation of operation classes: Operations chosen for the vectorization tests are classified into sub classes of operations as shown in Table V. Moreover, the table shows the operations that achieved the minimum and maximum speedup on vectorization. Performance improvement per operation subclasses in the Table V is shown in Figure 28. The figure shows the average, minimum and maximum speedup achieved per operation subclass.

On average, vectorization of semantic functions provides a speedup of $4 \times$ per target operation on emitting AVX2 instructions. Intuitively, one would expect a speedup of around $16 \times$ in this case as 16 scalar operations in a for loop run as one host vector intrinsic after vectorization. Practically, there are

```
void vec_shuffle(vr_t &R, const vr_t &A, const vr_t &B)
{
    uint16_t r, a, b;
    for (int i=0; i<R.L_16; i++) {
        R.data_16[i] = A.data_16[B.data_16[i]];
    }
}
```

Figure 27. vec_shuffle() operation semantics that failed to get auto-vectorized by compiler

### Table V

## 1) Vectorization Results: The vectorization results of the top operations tested from the noise reduction filters are shown in Table V. The table shows the total number of operations tested for each filter. There are only two operations in each filter that fail auto-vectorization. The non-vectorized operations are the same in both filters, which are two variants of vec_shuffle() operation. This case is illustrated in an example code in Figure 27. vec_shuffle() operation fills the result vector $B$ by elements of $A$ at indices pointed by the elements of $B$. These indices can be randomly ordered which are determined by the data filled in vector $B$ during runtime. This poses a major challenge to the vectorizer due to the random data access from noncontiguous points in memory. While this case fails with both GCC and LLVM, ICC can vectorize the same by enforcing vectorization with the use of compiler pragma, `#pragma simd`, before the loop nest. However, the vectorization benefit is almost negligible, giving a speedup of only 1.06 over the non-vectorized version. On this forced vectorization, ICC chooses the shuffle intrinsic available for 32 bit vector elements in Intel AVX2 as Intel intrinsics do not support shuffle operation for 8 or 16 bit data so far.

### 2) Performance evaluation of operation classes: Operations chosen for the vectorization tests are classified into sub classes of operations as shown in Table V. Moreover, the table shows the operations that achieved the minimum and maximum speedup on vectorization. Performance improvement per operation subclasses in the Table V is shown in Figure 28. The figure shows the average, minimum and maximum speedup achieved per operation subclass.

On average, vectorization of semantic functions provides a speedup of $4 \times$ per target operation on emitting AVX2 instructions. Intuitively, one would expect a speedup of around $16 \times$ in this case as 16 scalar operations in a for loop run as one host vector intrinsic after vectorization. Practically, there are
Table V

INTEL IPU4 IDSP OPERATIONS TESTED CLASSIFIED INTO OPERATION SUBCLASSES. FOR EVERY SUBCLASS, OPERATIONS THAT ACHIEVED MAXIMUM AND MINIMUM SPEEDUP ARE SHOWN.

<table>
<thead>
<tr>
<th>Maximum speedup</th>
<th>All operations</th>
<th>Basic Arithmetic</th>
<th>Complex Arithmetic</th>
<th>Basic Logic</th>
<th>Complex Logic</th>
<th>Shift</th>
<th>Comparison</th>
<th>Masked</th>
<th>Miscellaneous</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>addsat</td>
<td>mul</td>
<td>addsat</td>
<td>or</td>
<td>and</td>
<td>asrsat_wsplit_ic</td>
<td>lsr</td>
<td>eq_ic</td>
<td>rmux_ic</td>
<td>flag_to_vec_u</td>
<td>imax</td>
</tr>
<tr>
<td>Minimum speedup</td>
<td>isum</td>
<td>mul_ic</td>
<td>min max</td>
<td>and_ic</td>
<td>llsat_ic</td>
<td>lsl_ic</td>
<td>le</td>
<td>cond_add</td>
<td>even</td>
<td>isum</td>
</tr>
<tr>
<td>Rest</td>
<td>add</td>
<td>max_ic</td>
<td>min</td>
<td>subabsat</td>
<td>asrrnd_ic</td>
<td>lsl</td>
<td>eq</td>
<td>ge</td>
<td>cond_add_ic</td>
<td>odd</td>
</tr>
<tr>
<td></td>
<td>add_c</td>
<td>subabsat</td>
<td>suba</td>
<td>sat</td>
<td>lsr_ic</td>
<td></td>
<td></td>
<td>ge_ic</td>
<td></td>
<td>mux</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>absbat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rmux</td>
</tr>
</tbody>
</table>

Figure 28. Speedup on various operation classes on switching the vectorization flag on and off on top of -O3 compiler optimization level. The speedup noted here is the best speedup achieved from the three compilers that were used to benchmark. The architecture is Intel Haswell that emitted A VX2 instructions. In every subclass, the maximum speedup achieved per operation, geometric mean of all the speedups per operation and minimum speedup achieved are shown.

There are many reasons for the lower speedup. One of them is the loop transformation optimizations that a compiler performs on a loop nest in -O3 optimization level even when vectorization is off. This brings the cycle cost of a semantic loop nest executed with scalar operations of the target much lower. One of the other reasons is the cost of memory moves that need to be performed per operation. In many cases, some instructions are also required to reorder the vectors as the host architecture and target architecture are of different widths. Most of the time, computation part of the operation takes only about 30% of the time that it takes to finish the entire semantic function execution.

The best speedup achieved from all the operations tested comes from the vec_addsat operation in the complex arithmetic class. This is one of the operations whose best mapping on the host is emitted by the vectorizer on feeding the compiler with the right idioms as explained in Section IV-D. Speedup achieved in this case not only comes from the vectorization of all the scalar operations, but also from the reduction of multiple vector operations into one. This points to one of the major advantages of the auto-vectorization strategy. Utilizing the compiler idioms of an operation, most of the complex operations that were part of semantics before could be easily mapped onto single host intrinsic eliminating the complex control flows and additional instructions executed for realizing their semantics. This can magnificently improve performance in such cases as seen in the case of vec_addsat operation.

On the other hand, the minimum speedup comes from the reduction subclass for all compilers. Reduction loops are noted to behave very differently per compiler. For instance, LLVM compiler does not reduce the loop if the loop count is provided statically as implemented in the VectorRegister Class. Instead, it unrolls the entire loop generating a lot of scalar code. This unrolling could be prevented by passing the loop count as a
Average speedup in basic arithmetic is huge mainly due to 6 element each. Vectorization for this gives a nice speedup of integer into a vector by converting integer bits into a vector category is the vec_flag_to_vec_u(). Another interesting operation in this operation shown in Figure 9, which gives a speedup vec_mux(). This performance issue is even worse in SSE4.2 for moving the data to shift between two result vectors each of 256 bit width. This performance advantage it brings in vectorizing masked operations. Masked operations are operations that have an integer as operand for vector operation which ultimately needs the integer to be converted to a vector before executing the operation. The vectorized version of vec_flag_to_vec_u is shown in Figure 30. Comparison operations are special as the results of comparison between two vector operations is an integer, whose bits represent the results per element operation of the operand vectors. This is possible because the result of comparison operations is a boolean. To this end, vectorization could be difficult as the operands are vectors and the result is an integer. On identifying the compiler idiom for this, these operations also got vectorized. One example of such an operation is shown in Figure 31. This sub-class also shows an average speedup of 4x.

3) Host independence: The same loop nests for semantics are passed through various architectures in Table III with different SIMD widths and ISA to test the host independence feature of the proposed methodology. The result of this experiment for the add operation of the target is shown in Table VI. Passing the code for 512 bit vector wide target intrinsic emitted 8 vector add instructions for ARM Neon, 4 for Intel SSE4.2 and 2 for Intel AVX2. Vector register widths are 64 bits, 128 bits and 256 bits for ARM Neon, SSE4.2 and AVX2 respectively. Note that, Intel has AVX-512, 512 bit wide vector for Intel Xeon KNL architecture, but it does not support 8 bit or 16 bit operations. At the same time, Intel has released the ISA for Intel AVX-512 SKX instructions which is a new architecture that directly maps to the vector width of IDSP and supports 8 bit and 16 bit operations. On generating assembly for this architecture, a single add operation was emitted for the target add operation as desired. This clearly demonstrates the host independence nature of my solution in which the varying SIMD widths and new host machines are easily handled by

---

```c
void vec_mul(vr_t &Rlsb, vr_t &Rmsb, const vr_t &A, const vr_t &B) {
    int32_t r;
    for (int i=0; i<Rmsb.L_16; i++){
        Rmsb.data_16[i] = static_cast<uint16_t>((A.data_16[i] * B.data_16[i]) >> 16));
    }
    for (int i=0; i<Rmsb.vector_elements_16; i++){
        Rlsb.data_16[i] = static_cast<uint16_t>((A.data_16[i] * B.data_16[i]) & 0x0000ffff);
    }
}
```

Figure 29. vec_mul() operation semantics written as two separate loops to generate two result vectors as required: one for MSB of the results and the other for the LSBs.

```c
void vec_flag_to_vec_u(vr_t &R, const vr_t &A) {
    for (int i=0; i<R.L_16; i++){
        R.data_16[i] = (A&(0x1 << i)) !=0;
    }
}
```

```c
void vec_ge(vr_t &R, const vr_t &A, const vr_t &B) {
    for (int i=0; i<A.L_16; i++) {
        a = (A.data_16[i] > B.data_16[i]) ? a | (0x1 << i) : a;
        R = a;
    }
}
```

---

8https://llvm.org/bugs/show_bug.cgi?id=28090
except for AVX-512, all the other architectures are available to test and the author performed the vectorization experiment of operations as shown for add operation in Table VI with the LLVM compiler. Running the same code on all these architectures gives an average speedup of 3× as shown in Figure 32. This shows that the solution guarantees a reasonable performance enhancement on all tested vector architectures. Speedup per vector operation obtained on all the listed architectures is added in Appendix B.1.2.

Table VI

<table>
<thead>
<tr>
<th>Intel IPU4 IDSP (512 bit wide vector)</th>
<th>ARM Neon (64 bit wide vector)</th>
<th>Intel SSE4.2 (128 bit wide vector)</th>
<th>Intel AVX2 (256 bit wide vector)</th>
<th>Intel AVX512 SKX (512 bit wide vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vadd.i16</td>
<td>paddw</td>
<td>vpaddw</td>
<td>vpaddw</td>
<td></td>
</tr>
<tr>
<td>vadd.i16</td>
<td>paddw</td>
<td>vpaddw</td>
<td>vpaddw</td>
<td></td>
</tr>
<tr>
<td>vadd.i16</td>
<td>paddw</td>
<td>vpaddw</td>
<td>vpaddw</td>
<td></td>
</tr>
<tr>
<td>vadd.i16</td>
<td>paddw</td>
<td>vpaddw</td>
<td>vpaddw</td>
<td></td>
</tr>
<tr>
<td>vadd.i16</td>
<td>paddw</td>
<td>vpaddw</td>
<td>vpaddw</td>
<td></td>
</tr>
<tr>
<td>vadd.i16</td>
<td>paddw</td>
<td>vpaddw</td>
<td>vpaddw</td>
<td></td>
</tr>
<tr>
<td>(vld1.16,vst1.16)</td>
<td>(movdqu)</td>
<td>(vmovdqu)</td>
<td>(vmovdqu16)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 32. Geometric mean of speedup achieved on vectorization of semantics per three different vector ISAs: (1) ARM NEON that supports 64 bit vector register, (2) Intel SSE4.2 that supports 128 bit vector register and (3) Intel AVX2 that supports 256 bit vector register. The speedup is computed as the ratio of time taken for execution of all operations with the compiler flags for vectorization-off and vectorization-on. ARM Neon was ran with LLVM 3.9 and rest were ran with LLVM 3.8 compiler with -O3 optimization level.

The compiler.

Except for AVX-512, all the other architectures are available to test and the author performed the vectorization experiment of operations as shown for add operation in Table VI with the LLVM compiler. Running the same code on all these architectures gives an average speedup of 3× as shown in Figure 32. This shows that the solution guarantees a reasonable performance enhancement on all tested vector architectures. Speedup per vector operation obtained on all the listed architectures is added in Appendix B.1.2.

4) Compiler independence: For evaluating the portability of the methodology on various vectorizing compilers, the author worked with three popular vectorizing compilers: LLVM, ICC and GCC. For analyzing the applicability and performance, all the tests per operation were performed on all of them. This experiment is intended towards showing the portability of my solution on various compilers and not as a benchmark of their vectorization capabilities. As seen in Figure 33, all of them are equally good for applying this methodology with only slight performance variations. The first column shows the geometric mean of the time taken to run all the operations on each of them. When GCC and LLVM are almost close in performance, ICC is identified to perform relatively well. GCC fails for the class of comparison operations, which can either be because GCC does not recognize the vector to integer idiom used as shown in Figure 31 or because it has a different compiler idiom for this case. Most of the time, the instruction selection of the compiler is identified to vary considerably which also results in performance variations.

ICC functions better on explicit vectorization strategy more than being an auto-vectorizer. When LLVM and GCC vectorize a loop by analyzing memory aliasing, alignment issues etc. by themselves, ICC is aided by the compiler pragma, _pragma simd, before the loop nest to get them vectorized. That is an additional step taken for aiding ICC during these experiments, which can easily be controlled by a macro for keeping the solution compiler independent. This was unavoidable as ICC is truly an explicit vectorizer and the aim is to include ICC in testing the possibility of vectorization using multiple compilers. It has to be stated that adding this pragma would not be very safe in all cases and has to be used with caution. Even though the approach proposes low level C code to be written that is not compiler specific, it is clear that the performance varies between compilers. All these compilers have different flags and pragmas that can be added to do aggressive vectorization by explicitly helping compiler for vectorization. Execution time per vector operation on compiling with LLVM, GCC and ICC is shown in Appendix B.1.2.

5) Reduced effort in development: By using _C/C++ code for semantics description for compiled simulation and utilizing the existing compiler infrastructure for vectorization, effort required in leveraging the host SIMD capabilities has been reduced considerably. In the proposed methodology, the effort required in extending the IR for accommodating vector instructions of various types and widths, as well as lowering step for every back end instruction is totally left to the existing compiler infrastructure. After applying the major vectorization steps, as discussed in Section IV-C, very less effort is required in vectorizing the loop nests for vector semantics. The author tried to estimate the time invested for this process, from writing
the loop nest for each semantic by hand, vectorizing and optimizing it, and testing the correctness of semantics per class of operations. The assumption taken here is that there is a testing framework already available. The estimated time per operation subclasses is as shown in Figure 34. On average, it should take around 40 minutes per operation, considering the geometric mean of each subclass of operations. At the same time, taking a weighted average of each subclass would give a better estimate of the time taken per operation as the number of operations in each subclass can vary hugely.

C. Macro-benchmarking

This section evaluates the simulation performance enhancement using the new compiled simulator, VCSIM. From Section VI-B, ICC is identified as the best compiler for vectorization. Therefore, ICC is the chosen compiler for the experiments in this section. Intel AVX2 instructions are chosen for vectorization as it is the closest architecture to the IDSP in terms of vector width.

1) Impact of proposed methodology on simulation: For evaluating the effect of auto vectorization on the overall simulation performance, two simple firmware applications are considered: matrix multiplication and sobel filter. The speedup noted for each of them is shown in Figure 35.

In the matrix multiplication kernel, the overall speedup in simulation achieved is $1.5 \times$ compared to the non-vectorized version with -O3 optimization. At the same time, vector execution part of the code saw a speedup of $10 \times$. Two major vector operations executed are vec_add and vec_mul operations. The distribution of the time taken for the entire application showed that in the case of non-vectorized version, 65% of the time taken for simulation was spent in the scalar part of the code as well the simulation overhead. Moreover, after vectorizing, this overhead increases a bit further due to the extra costly instructions required for data reordering for vectorization. Analysis of the time spent in simulation shows vector move being the bottleneck in this case. In the vectorized version, scalar code and simulation code together take 94% of the time. The distribution of the execution time in both vectorized and non-vectorized versions is shown in Figure 17 in Appendix B.2.1.
The second firmware application considered is the $5 \times 5$ sobel filter. This application binary has 10 vector operations executed out of which 9 of them got vectorized. The non-vectorized operation is the shuffle operation as explained in Section VI-B1. The distribution of time taken for the entire application showed that in the case of non-vectorized version, 55% of the time taken for simulation was spent in the scalar part of the code as well the simulation overhead. The overall simulation speedup observed is $2 \times$ and the vector execution part showed a speedup of $9 \times$ in this case. Similar to the matrix multiplication, vector execution has reduced the simulation time considerably after vectorization while the simulation section including the non-vectorized shuffle operation has increased. The distribution of the execution time in both vectorized and non-vectorized versions is shown in Figure 18 in Appendix B.2.1.

It is important to note that real heavy applications from firmware could not be tested as they require support for wide range of semantics in C++ which could not be done during the period of the thesis. But the overall speedup per operation in Figure 28 shows that on an average, vector part of simulation can be reduced by $4 \times$ which will become relevant in compute intensive and densely scheduled vector VLIW firmware. On average, for the considered applications, overall simulation speedup achieved is $1.75 \times$, which includes a $9.5 \times$ speedup for the vector execution part. The speedup that can be achieved on overall simulation is data dependent as well as schedule dependent.

2) Retargetability of the simulator: For checking the reliability of the tool, the simulation compiler was generated and tested for 8 cores shown in Table II from the ICG regression. Out of these, 6 cores are scalar VLIW, ranging from 1 to 27 issue slots and two of them are the vector IDSPs, which have both scalar and vector issue slots. The semantics operations to all the scalar architectures were implemented manually in C/C++ by the author for the complete instruction set per each core. However, both IDSPs could not be tested rigorously due to the requirement of implementation of C/C++ semantics for entire instruction set manually, which is indeed a tedious and time consuming process.

3) Simulator Optimization and Evaluation: As discussed in Section V-H2, loss of control flow and the high level structure of code are the major limitations in taking binary as input for simulation. Handling these cases requires extra control flows in the simulator code to check for jump cases that impose limitations in the performance of the simulator. Running a performance analysis showed that the hot spots in the simulator function are the jump handling routines.

The other major concern in simulation performance is the possibility of indirect branches that enforce labeling all the cycles in the program. Such labels for all cycles in a program limits the compiler optimization scope as execution of every cycle becomes a basic block that has to be executed in order. Indirect branches are generally introduced by switch cases and function pointers in the application source code. To this end, a routine is introduced in the translator that identifies the case of indirect branches in the application binary. In programs with no such computed branches, labels were provided only to the entry points captured during the translation time. This is controlled by a macro in the simulator code which is set or unset by the translator while generating the simulator source code. This should eliminate the optimization limitation of compiler due to small basic blocks per cycle and widen the scope of optimization if there are no computed branches in the program.

As expected, goto version performed slightly better than the switch versions. The execution time distribution between a switch case version and goto version for matrix multiplication application is shown in Figure 36. Simulator and jump handling part of the code is reduced by the optimizations performed. However, the hot spot of execution in vectorized versions is shifted to the multiple vector move operations performed by the compiler for vector operations. The VectorRegister class in Figure 6 shows the memory allocation done per vector register. Apparently, compiler performs all these vector operations as memory operations, loading and storing data every now and then. This is a serious limitation compared to the scalar
operations where compiler identifies the dependencies and reuse data by storing them in registers. Around 15% of the simulation time is spent performing the copy between register and temporary variable introduced for VLIW RAW, which is the register write step during the execution.

4) Simulator Quality Analysis: For analyzing the performance quality of the new simulator, VCSIM, the simulator performance is compared with both existing simulators as well as the native execution on the real IDSP hardware. Comparing with the existing approach helps to judge the benefit of the new simulator where as comparing with the real hardware shows the room for improvement in the new design.

For this, the matrix multiplication application is simulated with the existing compiled simulator of ICG and compared with the VCSIM for both switch version as well as the goto version introduced in Section VI-C4. Taking the total cycle count from the simulation and considering the hardware clock frequency to be 400 MHz, native execution time is calculated as in Equation 1,

\[
time_{native} = \frac{(total\_cycles)/(400 \times 10^6)}{seconds} = \frac{68}{68\times 10^6}
\]  

All these execution times are normalized to the time taken for the execution on ICG compiled simulator to show the relative speedup as shown in Figure 37.

Even though the VCSIM approach shows more than 100× speedup to the existing solution, this has to be taken with a grain of salt as the existing solution also simulates the host and includes SystemC threads for checking for device calls in every cycle. The system simulated with the existing approach is the default system including a host, IDSP and busses only. In such a simple system the slow down due to the SystemC threads is reportedly 4×. This can slow down the VCSIM by 4× which then makes it around 40× faster considering the VCSIM. It is still worthwhile to note that the new simulator improves the performance to the existing compiled simulator as all function call executions are inlined and the semantic for each has been simplified rigorously. Th pipeline buffer for each pipeline is allocated statically during the translation which also adds to the performance enhancement.

Comparing to the native execution, goto version is 4× slower where as the switch version is 5× slower. Running a performance analyzer shows that the major hot spots in the simulation are the vector moves done per vectorized operation for both getting the operands and copying the results. The compiler does not identify the reuse of the data and lifetime of the data in vector register as it would normally do in scalar registers. This is mainly due to the fact that vector registers are modeled and accessed as memory elements. Most of the time, compiler might be find difficulty in analyzing the dependency between vector elements that lead to repeated loads and stores from the memory. Copying between the temporary variables and registers for the RAW dependency is a major simulation overhead.

![Figure 37. Speedup of execution of the VCSIM switch and goto version compared to the ICG compiled simulator. Slowdown to the native execution on the real hardware is also presented.](image)

VII. Conclusion

With the capabilities of auto-vectorizing compilers today, it can be argued that majority of the vector intrinsic semantic functions can be auto-vectorized by following certain code refactoring principles. Even though it is an open problem that auto-vectorizers are not capable of generating highly performing vector code as compared to manual optimizations [10], I demonstrated the applicability of auto-vectorization in a vector processor simulator application. I also presented the key guidelines for making maximum use of vectorizing compilers for emitting vector code, which can be applicable for any application of this nature. To this end, I introduced the idea of leveraging compiler idioms for optimizing vectorized code to emit direct mapped host intrinsic for faster simulation.

The approach of depending on the compiler for emitting host SIMD achieves host independence for running target code on any potential host vector core that supports an auto-vectorizing compiler. I show that SIMD width management is totally managed by the compiler, making this approach reliable and retargetable for future generations at both target and host level. By making use of the existing compiler infrastructure and writing low level C code, development effort for a new simulator framework to support vectorization is significantly reduced and is made much more maintainable.

I demonstrated that an average speedup of 4× can be achieved per vector intrinsic operation of the target considered. Moreover, an average speedup of 9.5× for vector execution is demonstrated on an application simulator, with an average speedup of of 1.75× for the entire simulation. At the same time, covering a variety of benchmarks is not possible at this stage as the semantic of operations have to be specified manually. The advantage of using the proposed methodology can vary hugely depending on the nature of the algorithms and density of the
VLIIW schedule. This means that the impact of the approach will be surfaced in the simulation of vector computation bound application.

I believe that on a perfectly matching architecture, the speedup that could be achieved with this methodology can get closer to a manual mapping of the intrinsics. Tests performed on emitting Intel AVX-512 SKX instructions for the IDSP gave a single host intrinsic operation with required moves for getting operands and writing results back. This indeed shows the potential benefit of running on a best mapping architecture, compared to other architectures which emitted additional vector reordering instructions.

I think that this strategy can be applied in wide varieties of binary translators including QEMU [41] for code migration and faster emulation of SIMD cores. To this end, it is much easier to stick to the compiler idioms of a standard compiler. Moreover, it will be highly beneficial if all the standard compilers agree to the same pattern as compiler idioms for a particular class of instruction selection. Looking at the capabilities and work going on in these compilers, there is a trend of significant development and improvement of the vectorization feature each of them has to offer. It can be claimed that my solution will keep getting better along with the evolution of these compilers.

ACKNOWLEDGMENT

I gratefully acknowledge the support and guidance of my supervisors, Prof. Henk Corporaal (TU Eindhoven), Dr. Mohamed Bamakhrama (Intel), and Dr. Roel Jordans (TU Eindhoven), throughout this work. I sincerely appreciate their effort and time spent in conducting multiple discussion sessions that were quite useful for completing the work.

I am thankful to all my colleagues in Intel, who have provided me great help and support during the course of this work, making it a joyful learning experience. Special thanks to Joao Peralta Moreira, Felipe Chies Augustos, Rosilde Corvino and Merten Popp for offering me plenty of help when I really needed it. I would also like to thank Mr. Saito Hideki (Intel ICC compiler engineer) who provided me with valuable insights on working with auto-vectorization.

I extend my deepest sense of gratitude to my lovely friends (Jose, Pompong) in India who had supported me a lot in the very beginning of this graduation journey and to my wonderful friends in Eindhoven (Avi, Tamara, Mathews, Vinay and Malloven folks) for their unending support. I will always be indebted to Vishal who encouraged me to pursue this academic dream and who keeps guiding me like the north star.

Last but not least, I am extremely grateful to my family for their unconditional love and support.

REFERENCES


Appendix

A Vectorizing Compiled Simulator

In this appendix, the author tries to explain in detail some of the various aspects of the simulator generation of the Vectorizing Compiled Simulator (VCSIM) tool introduced in Section V of this thesis report.

A.1 Handling static pipelining

Static pipeline handling is one of the main features of the IDSP as explained in Section V-G of the thesis report. In this section, the author presents a detailed explanation of the strategy used in handling the static pipeline case, illustrating with an example.

Figure 1 depicts an example VLIW issue slot that has an Arithmetic unit. As the figure shows, the issue slot has two input ports and two output ports. The input ports of the issue slot are connected to the register ports 1 and 2 respectively. Similarly, the outputs of the function units are connected to the issue slot output ports.

For every operation issued, operands come from the connected register ports and output goes into the issue slot output ports, which then gets carried by a bus to the respective register to write the result into. This means, a pipelined operation issued by this issue slot at the maximum will have to store these input values and output values for an operation. A pipeline buffer is used as the storage structure for handling the same. An example pipeline buffer model for the issue slot in Figure 1 is shown in Figure 2. In addition to the inputs and outputs, the opcode is stored to determine the operation that the pipe has to execute. Moreover, every pipeline buffer has a set bit to establish if it is occupied.

The idea behind pipelining of operations in a processor is to maximize the utilization of the resources performing independent sub tasks per operation in the same cycle. This means, when a pipeline operation is executing, another operation can be issued in the same cycle. For handling such multiple operations, issued by the issue slot that are running various stages of their pipe in the same cycle, an array of such pipeline buffers need to be created that can be allocated per pipelined operation. The size of such an array is determined by the maximum stages a pipe operation in the issue slot can have plus 1. This size is determined by finding the maximum pipeline operations that can be active in parallel per issue slot. Let us say function unit 2 of this issue slot gives output in 3 cycles. Therefore, the maximum number of stages this issue slot can execute is 3. Then the pipeline buffer declaration for this issue slot is an array of size $3 + 1$,

```
pipeline_buf_slot1 pipe_buf_slot1[4];
```

Once a pipeline operation is fetched, a pipeline_initialize() operation is called which allocates a buffer array to this operation. The opcode of the operation is passed as an input for the initialization. A variable pipe_buf_pointer_slot1 points to the index of the pipeline buffer array that can be allocated to the pipeline operation for this issue slot. The pipeline buffer is allocated in a circular buffer fashion by taking a modulo of the incremented value by the size of the array. Global variable used for handling the pipeline execution of all issue slots is the pipe_instance variable that increments when a pipeline operation is issued and decrements when a pipeline operation is finished. The pipe_instance value is polled in very cycle by the simulator to see if there is an active pipeline operation that needs to execute.

Assume that an add operation is issued in the current cycle which is executed by the arithmetic unit in the Figure 1. Considering that the arithmetic unit has the time shape of latency 1 for the inputs and...
Figure 1: An example issue slot of a VLIW that has two function units (FU). One of the FUs is an arithmetic unit that has an input latency 1 and output latency 2. The port connection for the input and output data path are shown along with a bus that is multiplexed between the issue slot output port.

```c
typedef struct pipeine_buffer_issueslot1 {
    int rf1_op1;       // input
    int rf1_op0;       // input
    int slot1_op0;     // output
    int slot1_op2;     // output
    boolean pipe_set; // occupancy bit
    int pipe_opcode;   // opcode
} pipeline_buf_slot1;
```

Figure 2: Pipeline buffer structure for the Issue slot 1 in Figure 1

latency 2 for the output, the required sub-operations for the add operation are detailed below one by one. The first step is shown in Figure 3.

Figure 3: Operation issue cycle for pipelined add operation

Suppose that the `pipe_buf_pointer` is pointing to 3 at this time. Then the pipe buffer array allocated is 3 for this opcode. For this index of the buffer, `pipe_set` variable is set to 1 and `pipe_opcode` is set to the opcode value of add. `pipe_instance` is incremented by 1 after allocating the pipe buffer.
is also incremented by 1 and its remainder with 4 is updated as the new value (for circular buffering). These steps are shown below.

pipe_buf_slot1[3].pipe_set = 1;
pipe_buf_slot1[3].pipe_opcode = opcode_add;
pipe_instance = pipe_instance + 1;
pipe_buf_pointer_slot1 = (pipe_buf_pointer_slot1 + 1) % 4;

Afterwards, the sub operations per each pipeline stage are issued one by one in the upcoming cycles as shown in Figures 4 and 5 respectively. Note that these sub operations are created automatically per pipeline operation by the parser tool which is shown in Figure 23 of Section V-G1 in the report. The Figures 4 and 5 show the execution of those sub operations.

```
Cycle n + 1:
// Both operands are obtained in this cycle as inputs have a latency of 1.
// RF1 array is the RF1 register array whose indices ip0 and ip1 are decoded
// during the translation phase.
pipe_buf_slot1[3].rf1_op1 = RF1[ip0];
pipe_buf_slot1[3].rf1_op2 = RF1[ip1];
// As both inputs are obtained in this cycle, operation execution is
// performed in the same cycle and the result is stored in the pipeline
// buffer output variable. std_add is the semantic function in C for the
// addition operation which is inlined.
pipe_buf_slot1[3].slot1_op1 =
    std_add (pipe_buf_slot1[3].rf1_op1, pipe_buf_slot1[3].rf1_op2);
```

Figure 4: Stage 1 for the pipelined add operation.

```
Cycle n + 2:
// Output stage copies the computed value stored in the pipeline buffer into
// the temporary variable of the issue slot port to which the result is
// written to.
temp_slot1_op1 = pipe_buf_slot1[3].slot1_op1;
```

Figure 5: Stage 2 for the pipelined add operation

As this is the end of the pipeline operation, pipe_instance is decremented by 1 and the pipeline buffer is freed by resetting it.

pipe_instance = pipe_instance - 1;
pipe_buf_slot1[3].pipe_set = 0;

A.2 Handling register write

This section explains the procedure of register write step in the translation phase explained in Section V-G2 of the thesis report.
In every cycle, all the buses are polled to see if any of them are active. For each bus, data from the respective issues slot ports is to be carried. Needless to say, only one issue slot port can write to a bus in one cycle which is selected by a multiplexer. For the output issued in the Figure 5, output is stored into the variable representing output of the issue slot 1 output port 1. In this cycle, bus 1 in the Figure 1 will be active which has the multiplexer selection bit set for the issue slot port output 1 to carry its data.

Register write step in the translation phase will have the corresponding copy instruction set in this cycle for copying the value from the issue slot port output to the register to write into. That is determined by the bus selected for the issue slot port. For instance, every register input will have a valid bus set to it to read data from the respective bus every cycle and the corresponding register write index. And for every bus, there are bits corresponding to the issues lot output port it takes data from. Both these cases are resolved during translation to copy the data from the issue slot output port value to the right register index. In the simulator, this copying instruction is issued as in Figure 6.

```c
// Mapping between issueslot output port and bus as well as the
//mapping between the bus value and register port is resolved internally by
//a few routines that work on the multiplexer select bits.

if(temp_slot1_op1_set == 1) {
    RF1[8] = temp_slot1_op1; // 8 is decoded for output of the operation
}
```

Figure 6: Register write step during the simulation of the IDSP that updates the execution result of an operation to the write register.

### A.3 Handling control flow

This section details the control flow handling step in the simulation phase explained in Section V-H2 in the thesis report.

Once a jump operation is issued during the program, the address to jump to is stored in a buffer. This is required because, similar to the pipeline stages, a jump operation has multiple stages of delay slots until the jump really happens, which is determined by the latency of the program counter. This is stored as a global variable `pc_latency`. The jump buffer model used for handling the jump case is shown in Figure 7. A counter variable `jump_count` is used to count until the delay slot finishes per jump. Jump buffer also has a set bit that establishes the occupancy of the buffer.

#### A.3.1 Jump buffer

During the delay slot cycles, it is possible to have other jumps issued. Therefore, a total number of jump buffers required is equal to the delay latency plus 1, which is the maximum number of active jumps possible. Suppose, the delay slot latency is 5 for the VLIW machine considered. Then the jump buffer declaration for this machine is,

```
jump_pipe_buf jump_buffer[6];
```

A variable `jump_buf_pointer` is used to point to the index of the jump buffer array that can be allocated to the new jump operation when issued. The jump buffer is allocated in a circular buffer fashion by taking a modulo of the incremented value by the size of the array allocated, which is 6 in my example. Global variable used for handling the jump execution is the `jump_instance` variable that increments when a jump
typedef struct jump_pipeline_buffer{
    boolean jump_set;
    int jump_count;
    int jump_address;
} jump_pipe_buf;

Figure 7: Jump buffer structure for handling the control flow execution during simulation.

operation is issued and decrements when a jump operation is finished. The `jump_instance` value is polled in
very cycle by the simulator routine to see if there is an active jump operation that needs to execute.

The steps that the jump handle process goes through, when a jump operation is issued, is depicted in an
activity diagram in Figure 8.

Figure 8: Activity diagram of jump execution handling done during the translation phase.

The first step, when a jump operation is issued, is the jump initialization process as shown in Figure 9. The parameter passed for the initialization of the jump is the jump address.

Figure 9: Operation issue cycle for jump operation

**Jump Initialization:** Suppose the `jump_buf_pointer` is pointing to 3 at this time. Then the jump buffer allocated is 3 for this opcode. For this index of the buffer, `jump_set` variable is set to 1 and `jump_address` is set to the jump address to jump to. `jump_instance` is incremented by 1 after allocating
Cycle n + delay_latency:
PC = jump\_buffer[3].jump_address;

Figure 10: Program counter update cycle for the jump operation

The jump buffer. jump\_buf\_pointer is incremented by 1 and modulo with 6 (for circular buffering). These steps taken by the jump handler are performed as below.

\[
\begin{align*}
\text{jump\_buffer[3].jump\_set} &= 1; \\
\text{jump\_buffer[3].jump\_address} &= \text{jump\_address}; \\
\text{jump\_buffer[3].jump\_count} &= 0; \\
\text{jump\_instance} &= \text{jump\_instance} + 1; \\
\text{jump\_buf\_pointer} &= (\text{jump\_buf\_pointer + 1}) \% 6;
\end{align*}
\]

Afterwards, in the upcoming cycles, the jump\_count is incremented and checked with the delay latency value. If the incremented count equals the delay latency value, jump address is updated to the PC as shown in Figure 10. As this is the end of the jump operation, jump\_instance is decremented by 1 and the jump buffer is freed by resetting it.

\[
\begin{align*}
\text{jump\_instance} &= \text{jump\_instance} - 1; \\
\text{jump\_buffer[3].jump\_set} &= 0;
\end{align*}
\]

A.4 Direct and Indirect Branches

In the section V-H of the thesis report, limitations encountered in the simulator optimization due to indirect/computed branches are discussed. This section shows two example execution programs to illustrate the cases of direct and indirect branches.

A.4.1 Direct branch

Suppose an application has only two jumps issued in a program size of 400 as shown in Figure 11.

PC 0 : std\_imm; 
PC 1 : std\_add; 
\ldots 
PC 230: std\_jump(320); //direct, value 320 available during translation 
PC 231: std\_sub; 
\ldots 
\ldots 
PC 400: std\_jump(6); //direct, value 6 available during translation

Figure 11: Disassembly of a program binary showing the direct jumps in a program.

As the figure shows, the jumps are directly a part of the instruction encoding as an immediate value and therefore, only programs 6 and 320 need to be labeled as they are the only entry points of the program. This is adapted to be so in the goto version where during the translation, jumps are inspected to see if there are only direct jumps; only then the potential jumps are labeled.

But that is not the case always. There are indirect branches possible in a program as shown in another example below.
A.4.2 Indirect branch

Suppose an application has two jumps issued in a program size of 400 as shown in Figure 12.

```
PC 0 : std_imm;
PC 1 : std_add;
...
PC 230: std_jump(R[3]); // indirect, point to jump to is available as
       // a register value filled during simulation runtime
PC 231: std_sub;
...
PC 400: std_jump(R[5]); // indirect, point to jump to is available
       // as a register value filled during simulation runtime
```

Figure 12: Disassembly of a program binary showing the indirect jumps in the program.

In this case, both jump values are updated during simulation runtime. This makes it nontrivial to identify the jump target during the translation time. The register value that is updated can potentially be any program counter and therefore all steps require label, which makes every cycle of execution a new basic block for the compiler.
B Evaluation

In this appendix, graphs related to some of the conclusive results, in the Evaluation Section VI of the thesis report are added for further reference.

B.1 Micro-benchmarking

This section presents the result graphs related to the Section VI-B in the thesis report.

B.1.1 Host independence

This section presents three graphs associated with the Figure 32 in Section VI-B3 of the report, which showed the speedup achieved on vectorization of the semantic of the IDSP vector operations by three different architectures. The graphs in Figures 13, 14 and 15 show 40 operations benchmarked and the speedup achieved per operation, for respective SIMD instruction set architectures ARM Neon (64 bits), Intel SSE4.2(128 bits) and Intel AVX2(256 bits). Compiler used for ARM Neon is LLVM 3.9 and for the rest is LLVM 3.8 with -O3 optimization level.

Figure 13: Speedup on vectorization of target intrinsic execution on ARM NEON that has 64 bit vector register.
Figure 14: Speedup on vectorization of target intrinsic execution on Intel SSE4.2 that has 128 bit vector register.

Figure 15: Speedup on vectorization of target intrinsic execution on Intel AVX2 that has 256 bit vector register.
B.1.2 Compiler independence

This section presents the graph associated with the Figure 33 in Section VI-B4 of the thesis report, which showed the compiler performance per operation subclass. The graph in Figure 16 shows the time taken for the execution of the vectorized versions of all the 40 operations, on compiling with the three different compilers: LLVM 3.8, GCC 6.1.1 and ICC 16.0.3. The variation in performance per compiler is clearly visible in this graph.

![Figure 16: Execution time per vector intrinsic semantic execution on compilation with three compilers: LLVM 3.8, GCC 6.1.1 and ICC 16.0.3.](image)
B.2 Macro-benchmarking

This section presents the result graphs related to the Section VI-C of the thesis report.

B.2.1 Simulation Execution Time

This section presents two graphs associated with Figure 35 in Section VI-C1 of the thesis report, which showed the speedup on overall simulation and vector execution part of two firmware applications. The graphs in this section show the execution time distribution of the firmware application benchmarked for both vectorized and non-vectorized version: Figure 17 shows the time distribution for Matrix Multiplication application and Figure 18 shows it for the Sobel filter. Compiler used for running the benchmarks is ICC 16.0.3 and the instruction set architecture is Intel AVX2.

![Figure 17: Execution time distribution for vectorized and non-vectorized execution of matrix multiplication application](image1)

![Figure 18: Execution time distribution for vectorized and non-vectorized execution of sobel filter application](image2)