Automatic Generation of Custom SIMD Instructions for Superword Level Parallelism

Taemin Kim
Intel Corporation
taemin.kim@intel.com

Yatin Hoskote
Intel Corporation
yatin.hoskote@intel.com

ABSTRACT

Application specific instruction-set processors (ASIPs) have drawn significant attention from System-on-a-Chip (SoC) community due to the capability of fine grain flexibility and customizability. In order to maximize the benefit of ASIP, automatic instruction set extension (ISE) is required. In the past decade, there have been plethora of researches on automatic ISE for custom scalar instruction. However, due to increasing usage of SIMD instructions to exploit data level parallelism (DLP) that exists both across loop iterations and within a basic block called Superword Level Parallelism (SLP), automatic generation of custom SIMD instructions is the inevitable direction of automatic ISE. In this paper, we propose an algorithm that automatically generates custom SIMD instructions from a set of custom scalar instructions to exploit SLP. We have demonstrated 52.4% and 30.8% performance improvement on average over base instruction set and additional custom scalar instructions, respectively.

1. INTRODUCTION

Application specific instruction-set processors (ASIPs) have drawn significant attention from System-on-a-Chip (SoC) community due to fine grain flexibility and customizability. They can support multiple applications and provides energy efficiency comparable to fixed function hardware by customizing their instructions and architectures accordingly.

However, manual design of custom instructions is a time consuming task that requires significant amount of effort to identify customization opportunity and construct best possible custom instructions in terms of performance, power, area and etc. Thus, in order to maximize the benefit of ASIP and improve design productivity to meet aggressive requirement on time-to-market, automatic instruction set extension (ISE) is inevitable. There have been extensive researches on ISE as shown in [13, 11].They are mainly focused on automatic generation of custom scalar instructions to exploit instruction level parallelism (ILP) by integrating multiple instructions and reclaiming cycle losses by fusing multiple instructions.

Furthermore, ISE by constructing custom single-instruction-multiple-data (SIMD) instructions is a natural direction to further advance automatic ISE technology. SIMD instructions have been used in general purpose processors for many decades [7] to improve performance more by exploiting data level parallelism (DLP) in applications. ASIPs also take advantage of DLP by integrating custom SIMD instructions [1, 2] However, there is almost no tool support to construct them automatically except the work in [6]. It generates custom SIMD instructions to exploit DLP that exists across loop iterations by performing dependency analysis and pattern recognition technique. However, DLP exists not only across loop iterations but also within a loop (i.e. in a straight line code) [9, 4] called superword level parallelism (SLP). We focus on automatic generation of custom SIMD instructions for SLP which has not been studied in ASIP community.

In this paper, we propose an algorithm to automatically generate custom SIMD instructions targeted at SLP. Main idea is that multiple instances of a scalar custom instruction that are independent of one another in a target application are clustered to form a custom SIMD instruction. Since we start from the result of custom scalar ISE, our algorithm can be easily integrated into the existing ISE flow. The algorithm is composed of two parts at large. One is to generate candidates of custom SIMD instructions out of the input custom scalar instructions. However, those candidates could form cyclic dependencies with one another that makes it impossible to schedule them correctly. Thus, in the second part, we prune some of them to remove cyclic dependency with minimal impact on performance. We performed experimentation with Mibench [8] to demonstrate performance improvement of our algorithm over base instruction set and custom scalar instruction combined with the base instruction set. We have demonstrated 52.4% and 30.8% performance improvement on average, respectively.

Our contributions are summarized as follows.

- We have developed an algorithm to perform automatic generation of custom SIMD instructions to exploit SLP.
- Our custom SIMD generation algorithm leverages existing tools of custom scalar instruction generation. Thus, it can be easily integrated into the existing automatic ISE flow.
- We have demonstrated performance improvement by performing experiments on widely used benchmark suite.

Our paper is organized as follows. Section 2 describes previous works on automatic ISE and compilation with SIMD instructions targeted at SLP. Section 3 motivates our work with an example followed by preliminaries and problem formulation in Section 4. Then, we describe our algorithm in detail in Section 5. Section 6 presents experimental setup and results and Section 7 concludes the paper.

2. RELATED WORKS

In this section, we introduce previous efforts in automatic SIMD ISE and software compiler that exploits SLP. In ASIP community, there have been plethora works on automatic generation of custom scalar instructions (e.g. [11, 13]). However, in these days, automatic SIMD ISE is drawing attention due to the fact that many compute intensive applications have rich inter loop DLP which can be exploited by SIMD instructions for performance improvement. Cong et. al [6] proposed automatic SIMD ISE algorithm to exploit inter loop DLP. They adopted dependency analysis technique in vectorizing compiler [3] to determine vectorizable portions in the loop. Then, they apply pattern recognition technique to generate vector instructions for them. Our work is different in the sense that we target different DLP, namely SLP. Combining two approaches would be a future research topic. Chouliaras and et. al [5] introduced electronic system level (ESL) design flow that extracts SIMD extension from vectorized C/C++ description, and then converts them into System-C macros to synthesize. However, it is unclear that they constructed the SIMD extension automatically in their flow.

In software compiler domain, SLP has been exploited to maximize the benefit of SIMD instructions of general purpose processors. Larsen et. al [9] introduced SLP concept first and proposed a compiler algorithm to exploit SLP by using SIMD instructions. Barik et. al [4] also proposed a vectorization algorithm to select SIMD instructions for SLP.

Our work is unique among the works previously mentioned in the sense that we automatically *generate* custom SIMD instructions targeted at *SLP*.

3. MOTIVATING EXAMPLE

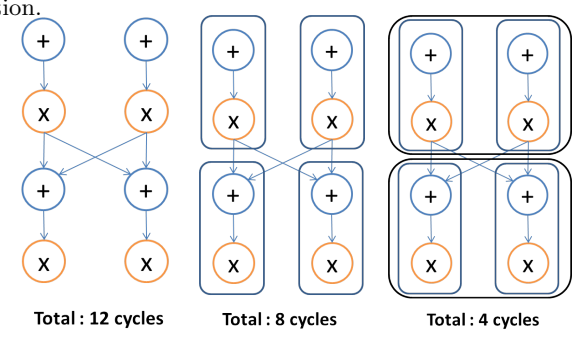
In this section, we present an example that motivates our

978-3-9815370-2-4/DATE14/©2014 EDAA

work. Figure 1 shows customization process of instructions starting from base instructions to custom SIMD instructions. In this example, we assume that a base processor issues one instruction per cycle and includes a 3-read-2-write register file. We also assume that addition and multiplication take physical delay of 0.5 cycle and 1.4 cycles, respectively, whereas their execution latency in processor pipeline is 1 cycle and 2 cycles, respectively.

Figure 1(a) shows an original DFG without any instruction customization. Given the assumption of execution latency of addition and multiplication, it takes 12 cycles to execute the DFG since it performs four additions and four multiplications sequentially. Figure 1(b) shows a result of custom scalar instruction generation. It reduces execution latency to two cycles by combining addition and multiplication together. Note that we cannot add more operations into the new instruction due to the constraint of register file ports. Consequently, it takes eight cycles in total to complete the execution of whole DFG.

We can reduce total execution cycles further by aggregating the custom scalar instructions into custom SIMD instructions as shown in Figure 1(c). Assuming two way SIMD architecture, two custom scalar instructions of same type are combined together to form single custom SIMD instruction. In Figure 1(c), two add-and-multiply instructions are grouped into single instruction. By doing so, we exploit SLP embedded in the DFG, thereby reducing total execution cycles to four cycles[1] which is 3X improvement over base processor configuration.



(a) (b) (c)

Figure 1: (a) Original Data Flow Graph (b) Enumeration of Custom Scalar Instructions (c) Enumeration of Custom SIMD Instructions

4. PRELIMINARIES AND PROBLEM FORMULATION

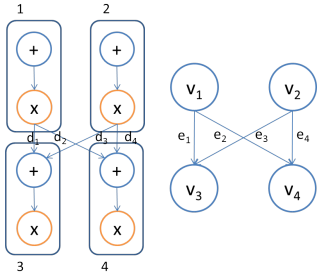
4.1 Preliminaries

Definition 1. **Scalar Instruction Dependency Graph (DG)** is a directed acyclic graph (DAG) denoted as $DG(V, E)$ with vertex set V and edge set E. A vertex in V represents a custom scalar instruction and a directed edge represents dependency relationship between two custom scalar instructions. For example, if there is data flow from $v_i \in V$ to $v_j \in V$, then an edge e_{ij} from v_i to v_j is constructed.

Figure 2 shows an example of *Scalar Instruction Dependency Graph* generated from Figure 1(b). Figure 2(a) is reproduced from Figure 1(b) with indexes on custom scalar instructions and data flows.Each custom scalar instruction numbered as 1,2,3 and 4 is mapped to a vertex and data flows denoted as d_1, d_2, d_3 and d_4 between them are mapped to directed edges e_1, e_2, e_3 and e_4 as shown in the figure.

Definition 2. **SIMD Dependency Graph (SDG)** is a directed graph $SDG(V, E)$ with vertex set V and edge set E. A vertex in V represents a SIMD instruction generated by combining scalar instructions in DG defined in $Definition$ 1. An edge in E represents dependency relationship between

[1]Note that butterfly interconnect between two SIMD operations is assumed to occur within the custom SIMD instruction without incurring any additional execution cycles

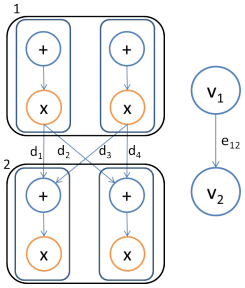


(a) (b)

Figure 2: (a) DFG with custom scalar instructions (b) DG constructed from (a)

two SIMD instructions. If there is a data flow from any operation in a SIMD instruction to any operation in another SIMD instruction, then, we construct an edge between them.

Figure 3 illustrates how SDG is constructed from DG. Figure 3(a) is reproduced from Figure 1(c) with indexes of SIMD instructions. Figure 3(b) is the SDG constructed from Figure 3(a). SIMD instruction 1 and 2 are mapped to vertex v_1 and v_2, respectively. The butterfly data flows are mapped to an edge. Note that multiple data flows denoted as d_1, d_2, d_3 and d_4 are collapsed into single edge e_{12}.



(a) (b)

Figure 3: (a) DFG with custom SIMD instructions (b) SDG generated from (a)

Definition 3. **Custom Instruction Set (CIS)** is a set $C = \{c_1, c_2, \ldots, c_n\}$ where c_i is a custom scalar instruction and n is the number of custom scalar instructions.

Definition 4. **Custom Instruction Instance Set (CIIS)** is a set $CI_i = \{ci_{i1}, ci_{i2}, \ldots, ci_{im}\}$ where ci_{ij} is an instance of custom scalar instruction $c_i \in C$ and m is the number of the instances in a given DFG G.

Definition 5. **SIMD Set (SS)** S_i is a subset of *Custom Instruction Instance Set* CI_i. S_i represents a SIMD instruction composed of instances of custom scalar instruction c_i.

Definition 6. **SIMD Instruction Set (SIS)** $S = \{S_1, S_2, \ldots, S_l\}$ where S_k is a *SIMD set* and l is the number of SIMD instructions. S is a collection of custom SIMD instructions.

An SIS is easily converted to a SDG. An element of the set is mapped to a vertex and dependence relationship between two elements is mapped to an edge.

Definition 7. **Legality of SIMD Instruction Set :** That a SIIS is *legal* means that its corresponding SDG is DAG.

4.2 Problem Formulation

In this subsection we formulate the problem of generating *SIMD Instruction Set* from CIS and $CIISes$. To simplify the problem to solve, we assume that area constraint has already been taken care of in the step of generation of custom scalar instruction that provides an input of our algorithm. We also assume that the number of input and output ports of scalar register file and SIMD register file is same.

Problem 1 Given *Custom Instruction Set* C , *Custom Instruction Instance Set* CIs each of which is associated with an element of C and vector width (VW), determine *SIMD Instruction Set* S that maximizes $\sum (L(S_i) - L_S(S_i))$ where L is a function that computes execution latency of $S_i \in S$ when elements of S_i (i.e. custom scalar instructions) are executed on a base processor and L_S is a function that computes execution latency when elements of S_i are executed in parallel as

a SIMD instruction. In other words, $L(S_i) - L_S(S_i)$ is cycle reduction when S_i becomes a SIMD instruction.

5. GENERATION OF CUSTOM SIMD INSTRUCTIONS

We divide **Problem 1** into two sub-problems. Given a set of *Custom Instruction Instance Set*s, a custom SIMD instruction is generated by combining the instances of a custom scalar instruction by the amount of vector width. We repeat the process for the instances of all of custom scalar instructions. However, custom SIMD instructions generated do not necessarily form a legal *SIMD Instruction Set* due to the possibility of cyclic dependency among custom SIMD instructions. Thus, there should be another process to legalize each *SIMD Instruction Set* in such a way that total cycle reduction is maximized. Therefore, **Problem 1** is naturally divided into two sub-problems as follows.

Problem 2 Given *Custom Instruction Set* C, a set SC of *Custom Instruction Instance Set*s each of which is associated with an element of C and vector width constraint, generate all possible *SIMD Instruction Set*s.

Problem 3 Given *SIMD Instruction Set* S, determine a legal *SIMD Instruction set* S_f that maximizes total cycle reduction.

Clearly **Problem 2** has exponential complexity due to all possible combinations. In addition, the naive solution to **Problem 3** is to remove all possible combinations of elements in a SIS and then check legality of the resulting SIS of each combination and cycle reduction. Thus, it has also exponential complexity. Due to the intractable nature of the problems, we propose heuristic algorithms to solve them in tractable manner. In the following sections, we present details of each algorithm to solve each problem.

5.1 Overview of Our Heuristic

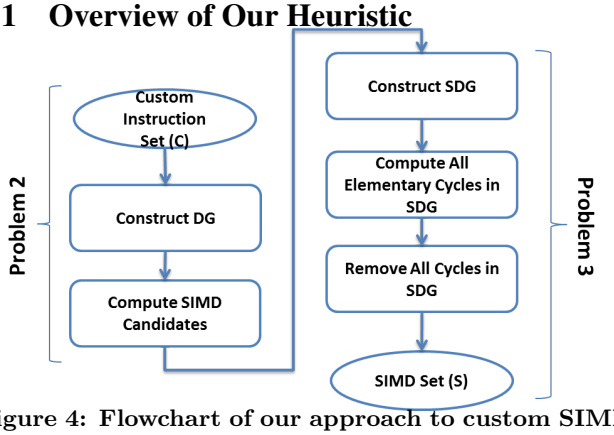


Figure 4: Flowchart of our approach to custom SIMD instruction generation

In this subsection, we overview our approach to generation of custom SIMD instructions. Figure 4 shows flow chart of our approach. As described previously, we solve two separate problems sequentially. For **Problem 2**, we heuristically generate single SIS instead of generating all possible SISes due to the exponential complexity. The elements of the set are possible custom SIMD instructions, namely candidates of custom SIMD instructions [2]. In order to generate the set, we accept a CIS C and a set of CIISes, IC. Each element of IC is associated with an element of C. By analyzing dependency among custom scalar instructions done in *Consruct DG* step, we know which custom scalar instructions can be executed concurrently. Then, we generate SIMD candidates as many as possible so that the probability of maximizing performance is maximized (*Compute SIMD Candidates*). The detailed algorithm is given in Section 5.2.

After generating a *SIMD Instruction Set* (S) comprised of SIMD candidates, we perform legalization of the set to generate final custom SIMD instructions, performed in *Problem 3* part in Figure 4. First we analyze dependencies among SIMD candidates in *Construct SDG* step. Then, in order to legalize S, we have to remove all of cyclic dependencies in SDG. Since we are targeting maximum cycle reduction, we have to minimize performance impact due to the removal of cyclic dependency. In order to remove all of the cyclic dependency, we first enumerate all elementary circuits [12] and then selectively remove vertices in the cycles done in the two steps after SDG construction. Final custom SIMD instructions are generated at the end.

[2]From now on, we call a candidate of custom SIMD instruction simply as a SIMD candidate.

5.2 Generation of Candidates of Custom SIMD Instructions

In this subsection, we propose the algorithm to solve **Problem 2** as described in Algorithm 1. The goal is to generate SIMD candidates as many as possible with minimum cyclic dependency. The intuition behind the maximum number is that the more SIMD instructions the more performance improvement. Note that even if we generate maximal number of custom SIMD instructions, we do not guarantee maximum performance improvement due to the fact that how many elements of the SIS would be eliminated at the legalization step is unknown at this step. How many SIMD candidates will be eliminated is dependent on how much cyclic dependency is in the SIS. Thus, we have to take into consideration minimization of cyclic dependency in the algorithm as well.

We generate the candidates basic block by basic block. (**line 1, 2 and 3**). The reason is to guarantee correct functionality when the SIMD instructions are inserted into the target application. If instances of a custom scalar instruction in different basic blocks are clustered together as a SIMD instruction, its execution might cause incorrect functionality, since the basic blocks are not in the same control flow in general. Thus, we collect instances of a custom scalar instruction in the same basic block (**line 3**).

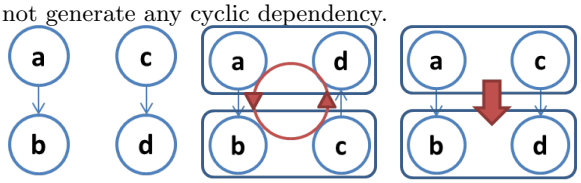
For each custom scalar instruction in each basic block, we construct DG (**line 4**) to capture dependency relationships among instances of the custom scalar instruction. Then, we find instances, mapped to vertices of DG, that are independent of one another to cluster them into a SIMD candidate (**line 6**). We perform the clustering until the number of instances in a SIMD candidate reaches VW. After constructing a SIMD candidate, we remove the vertices and edges associated with the candidate from DG. We repeat the construction until there is not enough independent vertices (**line 5-10**). Finally, the algorithm outputs a SIS whose elements are SIMD candidates.

The objective of this process is two folds. One is to minimize the number of cyclic dependencies among SIMD candidates and the other is to maximize the number of SIMD candidates. In order to achieve the objectives, we use simple priority system to assign each vertex of DG a priority to determine the order of selection in the process of SIMD candidate construction as described in Section 5.2.1.

5.2.1 Priority System

First, we minimize cyclic dependencies by giving higher priority to an earlier vertex in topological order than a later one. The intuition behind the idea is that two clusters do not form cyclic dependency if vertices in one cluster are topologically earlier than those in the other. However, we do not guarantee zero cyclic dependency at this step, since our priority system considers not only topological order of vertices but also degree of a vertex described later to maximize the number of candidates. Figure 5 shows an example of how cyclic dependency is minimized by considering topological order of vertices. Figure 5(a) shows a DG whose vertices and edges represent custom scalar instructions and dependency among them, respectively. Figure 5(b) shows a clustering result when we do not consider topological order. We cluster vertex a and d first and then b and c. As a result, two SIMD candidates form cyclic dependency, which means that one of them should

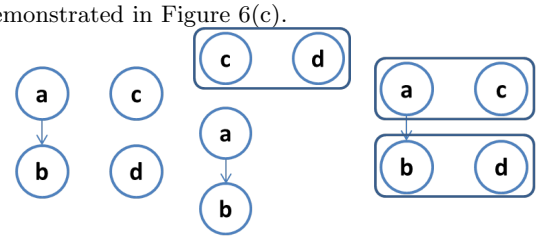
be eliminated at the legalization step. Figure 5(c) demonstrates another clustering result that does not form cyclic dependency by clustering earlier vertices in topological order first. We choose vertex a first because it is one of the earliest vertices in topological order and then we examine vertex c earlier than b and d due to the same reason. By doing so, we do not generate any cyclic dependency.



(a) (b) (c)

Figure 5: (a) Dependency Graph of Custom Scalar Instructions (b) SIMD candidate generation without consideration of topological order of vertices (c) SIMD candidate generation with consideration of topological order of vertices

Second, in order to maximize the number of SIMD candidates, we give higher priority to a vertex with higher degree. Specifically, we pick a vertex of DG that has the highest degree and remove all of its neighbors and associated edges. We repeat the process until the number of vertices in a cluster reaches VW, which completes construction of a candidate of custom SIMD instruction. The reason we give higher priority to a vertex with higher degree than others is that the vertex has less chance of finding a vertex which is independent of itself. In other words, the later we try to find independent vertex of high degree vertex, the smaller the number of vertices that can be clustered together with it. Thus, the possibility of the vertex to become a candidate of SIMD instruction becomes lower than when it is examined earlier. Figure 6 shows an example to demonstrate that considering the degree of vertices affects the number of SIMD candidates. Figure 6(a) is a DG. When we choose vertices c and d first to construct a SIMD candidate as shown in Figure 6(b), we generate one SIMD candidate because vertex a and b are in dependency relationship. However, if we choose vertex a first since it is the highest degree vertex, then we generate two SIMD candidates as demonstrated in Figure 6(c).



(a) (b) (c)

Figure 6: (a) Dependency Graph of Custom Scalar Instructions (b) Generation of custom SIMD candidates without consideration of degree of vertices (c) Generation of custom SIMD candidates with consideration of degree of vertices

5.3 Legalization of SIMD Instruction Set

Once a SIS has been generated, we legalize it so that final SIMD instructions do not form cyclic dependency as described in Algorithm 2. The objective of this step is to maximize cycle reduction of the final SIS, subject to the constraint of zero cyclic dependency. In order to examine performance impact of the legalization, we first compute gain of each SIMD candidate (**line 1**) that represents cycle reduction over custom scalar instruction, described in Section 5.3.1. Then, we examine whether each SIMD candidate participates in formation of cyclic dependency, followed by gain update to assign penalty. After capturing performance impact of each SIMD candidate, we break cyclic dependency by removing SIMD candidates in such a way that performance reduction due to the removal is minimized as described in Section 5.3.2.

5.3.1 Gain Computation

Algorithm 1: Construct a set of custom SIMD candidates

Input: CDFG (G), Custom Instruction Set (C), Custom Instruction Instance Set (CI)
Output: SIMD Instruction Set(SIMDCandi)

```
1  foreach BB in G do
2      foreach cit in C do
3          CIBB = ExtractCI In The SameBB(cit,CI,BB);
4          DG = Construct Dependency Graph(CIBB);
5          while there is a vertex in DG do
6              MIS = Maximum Cardinality SIMD Instruction
               Set(DG);
7              SIMDCandi += MIS;
               // Remove vertices and edges associated
               with MIS
8              Update Dependency Graph(DG, MIS);
9          end
10     end
11 end
```

Basically, gain represents cycle reduction when the instances of a custom scalar instruction in a SIMD candidate are executed in parallel. However, since SIMD instruction typically use a separate register file to accommodate multiple data elements in single register entry, we also have to consider cycle penalty to perform packing and unpacking of data. In addition, we need to consider overhead of shuffling and extracting data since data are not necessarily aligned. Considering cycle saving and overhead, we compute the gain of each SIMD candidate as shown in Equation (1). Note that the gain computation is not final because it is refined later to take into consideration the impact of cyclic dependency on the performance improvement.

$$Gain(S_i) = \Delta L(S_i) - \{P(S_i) + UP(S_i)\} \quad (1)$$

, where

$$P(S_i) = NI_{sc}(S_i)/N_{RFI} \times (1 + C_{shuffle}) \quad (2)$$

$$UP(S_i) = NO_{sc}(S_i)/N_{RFO} \times (1 + C_{extract}) \quad (3)$$

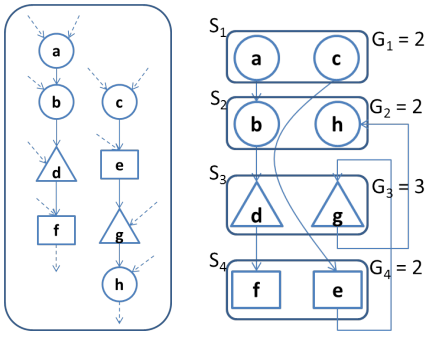
In Equation (1), $\Delta L(S_i)$ is cycle saving of SIMD candidate S_i. $P(S_i)$ and $UP(S_i)$ are cycle penalty due to packing and unpacking operands of S_i, respectively. Their computation is done as shown in Equation (2) and (3). $NI_{sc}(S_i)$ and $NO_{sc}(S_i)$ are the number of input and output operands from and to other scalar instructions, respectively. They incur overhead of operand transfer between scalar and SIMD register files. N_{RFI} and N_{RFO} are the number of input and output ports of both scalar and SIMD register file [3], respectively. $C_{shuffle}$ and $C_{extract}$ refer to cycle penalty to perform shuffling and extracting operands, respectively.

Figure 7 shows an example to assign gains to SIMD candidates. Figure 7(a) is a DG. Each shape in the figure represents a custom instruction. Solid edge shows data flow between them while dotted edge is data flow from/to base scalar instructions. In other words, dotted data are from/to scalar register file. Figure 7(b) shows SIMD candidates generated from Figure 7(a). Each box represents a SIMD candidate. Let us suppose cycle saving of all custom scalar instructions is four, and the numbers of inputs and outputs of scalar register file are two and one, respectively. Considering S_1 in Figure 7(b), its scalar instances accepts four input operands from scalar register file as shown in Figure 7(a). Its packing overhead is $4/2 = 2$ cycles assuming shuffling overhead is zero. However, all of its outputs are connected to SIMD candidates, which means zero unpacking overhead. Thus, its gain G_1 is two cycles (i.e. 4 - (2 + 0) = 2) as shown in the figure. Gains for other candidates are computed in the same way and are shown in the figure as well.

5.3.2 Removal of Cyclic Dependency

Algorithm 2 describes procedures to legalize a SIS. First, we construct SDG to capture all dependency information among

[3]Recall that we assume SIMD register file has the same number of input and output ports as scalar register file



(a) (b)

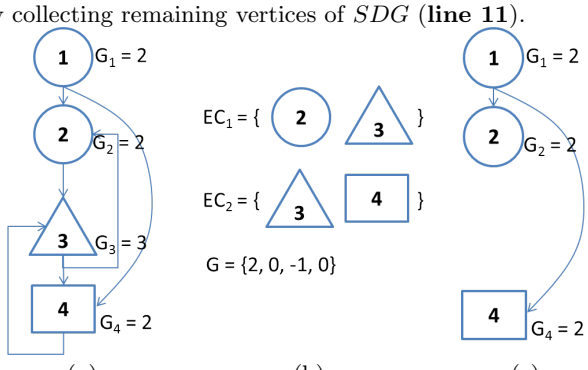
Figure 7: (a) Dependency graph (DG) (b) Candidates of custom SIMD instructions with gains

SIMD candidates (**line 2**). Then, we compute all of the elementary circuits (ECs) in SDG to capture all the cyclic dependency that SIMD candidates form by performing the algorithm in [12] (**line 3**). Each EC corresponds to cyclic dependency. We break an EC by removing a vertex in it. Since removing a vertex affects total cycle reduction of final SIS, we take into consideration the performance impact of vertex removal. We penalize a vertex by the amount proportional to the number of ECs it belongs to (**line 4-6**). In other words, we subtract the amount from the original gain of the vertex as shown in Equation (4).

$$Gain_{new} = Gain_{old} - \alpha \times N_c \quad (4)$$

$Gain_{old}$ and $Gain_{new}$ represent gain of a vertex computed in Section 5.2 and updated gain of the vertex, respectively. N_c is the number of elementary circuits where it belongs in SDG. α is an integer weight that represents the importance of N_c. It is an user input.

Once the gain of each vertex has been updated, we sort vertices of SDG in ascending order of gain (**line 7**) and then, we remove a vertex of the least gain. The removal of the least gain vertex means that its impact on the total execution cycle is possibly minimal or the number of elementary circuits the vertex belongs to is substantially large in comparison to others. We perform the removal until all elementary circuits in SDG are eliminated (**line 8-10**). Finally, the legalization with maximum reduction of total execution cycles is finished by collecting remaining vertices of SDG (**line 11**).



(a) (b) (c)

Figure 8: (a) SIMD dependency graph (b) Elementary circuits identified and updated gains of vertices (c) Final SDG

Figure 8 is an example that illustrates the process of legalizing SIS. Figure 8(a) is a SDG of SIMD candidates in Figure 7(b) with a gain of each vertex. In this example, we set α 2 in Equation 4. After computing all of elementary circuits, the result is shown in Figure 8(b). Two elementary circuits denoted as EC_1 and EC_2 are found. The set G shows updated gains of vertices. As shown in the figure, since vertex 3 belongs to both elementary circuits, its gain is updated to -1 (i.e. $3 - 2 \times 2 = -1$). Then, we remove the vertex because it is the least gain vertex, which removes all the elementary circuits. Finally, we get legalized SIS whose SDG is shown in Figure 8(c).

Algorithm 2: Legalize SIMD Instruction Set

Input: A set of Custom SIMD Instructions(SIMDCandi)
Output: Legal SIMD Instruction Set(CustomSIMD)

```
// Assign gain a SIMD candidate
1  Compute Gain For SIMD(SIMDCandi);
2  SDG = Construct SIMD Dependency Graph(SIMDCandi);
3  SetCycles = Enumerate All Elementary Cycles(SDG);
4  foreach SIMDNode in SDG do
5  |  Count Cycle Attendance And Update Gain(SIMDNode,
   |  SetCycles);
6  end
7  Sort SIMDNode In Ascending Order of Gain(SDG);
8  while There is no cycles in SetCycles do
9  |  Remove The Least Gain SIMDNode(SDG);
10 end
11 CustomSIMD = Vertex(SDG);
```

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

We have developed an automatic instruction set extension tool that generates both custom scalar instructions and custom SIMD instructions. Since our algorithm is orthogonal to the algorithm for custom scalar instruction generation, any algorithms can be used to generate custom scalar instructions. We used Yu and Mitra's algorithm [13]. The tool is implemented in C++ and by using LLVM libraries [10]. The whole flow to generate custom SIMD instructions is depicted in Figure 9. We accept a C/C++ application as input and perform initial compilation with LLVM front-end to generate LLVM intermediate representation (IR) shown as *LLVM IR* in the figure. Based on *LLVM IR* information, we construct control and data flow graph (CDFG) and then feed it to our core procedures, namely, custom scalar and SIMD instruction generation. First, custom scalar instructions are generated. The procedure accepts timing information of all functional blocks from component library, profiling information that provides edge profiling result and constraints such as area, instruction count, vector width, issue width and etc. Based on the information and constraints provided, we generate best possible custom scalar instructions. The custom scalar instructions generated are used as basis for custom SIMD instruction generation as shown in the figure. Our algorithm computes custom SIMD instructions as many as possible at the end of the flow.

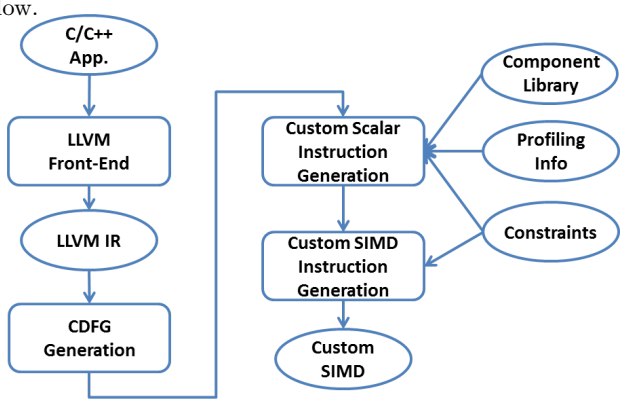


Figure 9: Whole flow to generate custom SIMD instructions from target application description

In order to estimate performance, we used similar performance model presented in [11]. We modeled software and hardware latency of a custom instruction. While software latency is execution cycle that it takes to perform a custom instruction in a base processor, hardware latency is the one when the instruction is executed on custom functional unit. The performance improvement of the instruction is the difference between software latency and hardware latency.

6.2 Results

In this subsection, we present experimental results mainly focused on performance improvement. In all experimenta-

tions, we assume 4-read-2-write scalar and SIMD register files. In addition, we limit the number of custom scalar instructions to 10. Note that this does not mean total number of instances is limited.

When custom SIMD instructions are used with custom scalar instructions, we can achieve even more performance improvement than the case where only custom scalar instructions are used. Figure 10 depicts our claim. In this experiment, we set vector width (VW) as two. The first bar in each benchmark application represents performance improvement over base instruction set, when only custom scalar instructions are used. The second bar represents the case where both custom scalar and custom SIMD instructions are used. The last bar is the performance improvement of the second case (Scalar + SIMD) over the first case (only Scalar). As the graph shows, *Scalar + SIMD* case improves performance up to 85.5% (i.e. 7X) and 52.4% (i.e. about 2X) on average. In addition, it is also 30.8% better than scalar only case on average.

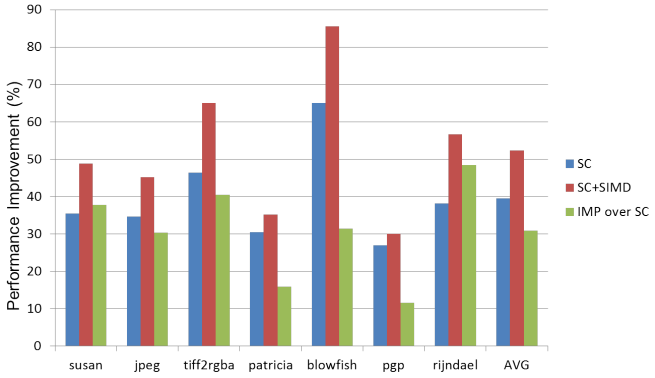


Figure 10: Performance improvement of instruction set extension with custom scalar instructions and hybrid of custom scalar and SIMD instructions

We also performed experimentation with change of vector width to see variation of performance improvement as shown in Figure 11. As the figure shows, performance improvement for larger vector width is quite limited. Performance improvement is smaller even in larger vector width in some cases. One possible reason is that as vector width increases, the number of custom SIMD instructions we can construct decreases due to that we do not allow a custom SIMD instruction whose number of scalar instances is smaller than vector width. Figure 12 verifies the hypothesis. As it shows, the larger the vector width the less the number of custom SIMD instructions generated. However, if we allow SIMD instructions such that the number of instances of custom scalar instructions is less than vector width, we could increase performance more by generating more SIMD instructions even with large vector width. This could be a future research topic.

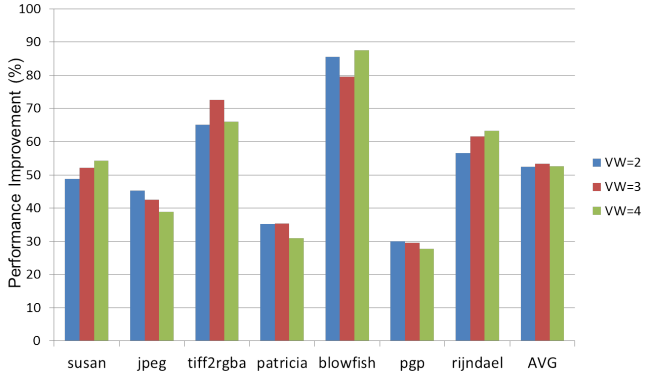


Figure 11: Performance improvement with the variation of vector width

7. CONCLUSION

Instruction set extension through custom SIMD instructions is increasingly drawing attention due to its hybrid characteristics of exploiting parallelism and customization toward target applications. SLP has been recently introduced as new parallelism that can be exploited by SIMD instructions. In this paper, we have presented a technique to perform automatic generation of custom SIMD instructions targeted at SLP. We have demonstrated significant performance improvement by applying our technique to several benchmark applications. Since applications have both SLP and inter loop DLP, combining both SLP and inter loop DLP targeted custom SIMD ISE would be a good research topic to investigate. We are currently investigating what implications there are in that context.

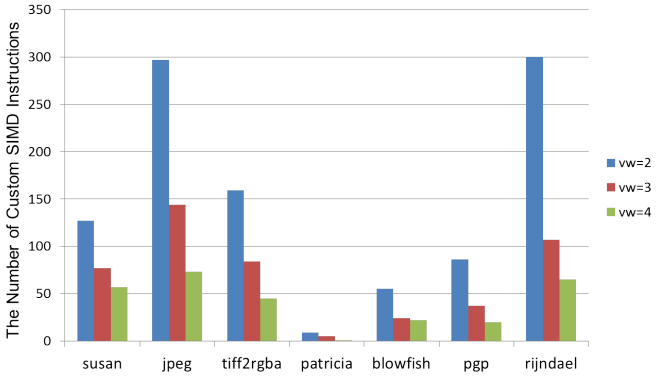


Figure 12: Variation of the number of custom SIMD instructions generated as vector width increases

8. REFERENCES

[1] `http::://www.retarget.com`.

[2] `http::://www.siliconhive.com`.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence- based Approach*. Morgan Kaufmann, 2001.

[4] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, Dec 2010.

[5] M. O. Cheema and O. Hammami. Customized simd unit synthesis for system on programmable chip: a foundation for hw/sw partitioning with vectorization. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2006.

[6] J. Cong, M. A. Ghodrat, M. Gill, H. Huang, B. Liu, R. Prabhakar, G. Reinman, and M. Vitanza. Compilation and architecture support for customized vector instruction extension. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2012.

[7] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14. IEEE, Dec 2001.

[9] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of Programming Language Design and Implementation (PLDI)*, Jun 2000.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Mar 2004.

[11] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 25(7):1209–1229, Jul 2006.

[12] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–756, December 1970.

[13] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the International Comference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 69–78. ACM, September 2004.