Experience with Partial Simdization in Open64 Compiler using Dynamic Programming

Dibyendu Das
AMD
dibyendu.das@amd.com

Soham Sundar Chakraborty
AMD
soham.chakraborty@amd.com

Michael Lai
AMD
michael.lai@amd.com

Abstract

In this work we implement a dynamic programming based approach for partial vectorization (simdization) in the Open64 compiler. Our work is an extension of a recent work published by Barik et al. (to be referred as BZS from now)[9] and utilizes a dynamic programming methodology to extract simdizable operations from straight-line code. The dynamic programming based algorithm has been shown to be superior to the traditional Superword-Level Parallelism (SLP) techniques pioneered by Larsen and Amarsinghe. Our contribution is in extending and adapting the BZS algorithm for the Open64 compiler. As part of this, we have simplified certain aspects of the dynamic programming approach including restricting the number of cases that are considered as candidates for finding the best solution. Our algorithm has been applied in the middle-end of the compiler unlike its implementation at the low-level optimizer in the BZS algorithm. Specifically, we have modified the LNO (Loop Nest Optimizer) which utilizes high-level WHIRL, adding both the analysis and transform phases in the simdization module of the LNO. This helps in reasoning with the algorithm at a higher level than originally proposed and results in a cleaner implementation. We have added several key ideas to the work including the notion of picking the right packing factor (also called tile-size or k-tiling in BZS) using an iterative process. Our algorithm can also identify reduction candidates (implemented as a separate pass) and partially simdize such operations. We have run our adapted algorithm on the SPEC CPU2006 benchmark, 435.gromacs, and have achieved a speedup of over 20% for both 1-copy (speed) as well as multi-copy (rate) runs. We are continuing to adapt and tune the algorithm for other benchmarks and applications.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors-Code Generation

General Terms DAG, CFG

Keywords SIMD, Simdization, Vectorization, Dynamic Programming, Optimizing Compiler, Code Generation.

1. Introduction

Partial simdization refers to simdization of straight line code. The end result of the transformation is a mix of simdized instructions and non-simdized (scalar) instructions. Partially simdized code is supposed to execute faster than un-transformed full scalar code. With the advent of processors that support wide SIMD vector registers up to 256 bits and SIMD instructions that can operate on such registers, it has become important for compilers to extract simdizable operations from a program automatically. The compiler also needs to use packing/unpacking operations judiciously to switch between the SIMD and non-SIMD code. Traditional vectorization is loop-based and depends on loop-dependence analyses to simdize. Loop-based techniques suffer from the constraints that if certain parts of a loop are simdizable and certain parts are not compilers may refrain from simdization. Some compilers may apply loop-fission or loop distribution to simdize such loops. The first well-known work to partially simdize straight-line code was by Larsen and Amarsinghe [15] which they termed Superword Level Parallelism (SLP). They showed that sufficient benefit can be derived on processors supporting SIMD instruction set by partially simdizing sections of code which are computationally hot. They also posited that by unrolling a loop and applying their method, loop-based simdization opportunities can also be effectively extracted.

Recent work by Barik et al. [9] uses a dynamic programming based approach to tackle the partial simdization issue. In their work they model the straight line code as a dataflow graph (DAG). They also employ a packing factor, called k-tiling, where k nodes (operations) are assumed to reside together in a vector register. Usually k is chosen based on the size of the SIMD operation (float,double,int etc) and the size of the SIMD register file (128,256, ...). Once k is known, k operations that can be executed in parallel are packed together. The operations that can be packed are chosen from an available set of independent operations. The cost of such a k-tiling is derived from the cost of its constituent children nodes on which these nodes depend. This is the crucial part of the algorithm and employs dynamic programming rules to pick between various alternatives. A bottom-up approach on the DAG is employed to calculate such vector costs till all the nodes are covered. A top-down cost-based policy is then employed to pick which operations should remain packed and which should be operated as scalars. The main advantage of this algorithm is a global view of the operations. The dataflow model with the dynamic programming approach does away with a local, greedy heuristic applied in SLP. The quality of the final solution, in terms of which operations should be simdized and which should remain scalar, is better than SLP, resulting in code that is more efficient.

Our work is an adaptation of the BZS algorithm applied in the context of the Open64 compiler. However, several modifications have been made to the original algorithm for improved speedup and compile-time requirements. First of all we do away with a fixed k-tiling approach. Instead, our algorithm uses a tiling factor of k which varies from $2, 3, 4, \ldots$. We pick a tiling factor which yields the best possible solution in terms of the lowest cost for the final

transformed code. This is based on our observation that a fixed k dependent on the SIMD vector register size may not be the best approach as the amount of parallelism that may exist in a portion of code may vary. For example the amount of parallelism that exists in a key hot loop of the 435.gromacs yields best to a 3-tiling rather than 4-tiling. The value of k is 4 if we use a 128-bit SIMD register file size and a basic data type of float. Using multiple tiling sizes and picking the best tile size allows us to be more flexible but with a small compilation time overhead. Secondly, our approach simplifies the original dynamic programming formulation by using only a few of the dynamic programming cost computation rules that are most likely to yield good reults. This is done to keep the compilation time in check as a large number of possibilities are examined. Thirdly, our algorithm is implemented at the middle-end of the Open64 compiler, at the Loop Nest Optimizer (LNO) phase that utilizes High-Level WHIRL [3]. Our transformation phase introduces vector types in the High-Level WHIRL [5] to represent operations that have been simdized. Some special packing, unpacking and shuffle operations are encoded at this level by calling intrinsic functions. These intrinsics are later lowered to their specific machine level instructions at the code generation phase. Applying the dynamic programming algorithm at the middle-end results in a cleaner solution as the loop-based vectorization module of the Open64 compiler has traditionally resided in LNO. In addition, any mix-and-match approach of loop-based simidzation and partial simdization can be transparently applied in the future.

As the first phase of our work, we have applied our algorithm for partial simdization to a key loop in 435.gromacs from the SPEC CPU 2006 benchmark suite [4]. The existing loop-based simdization module in Open64 fails to simdize this loop. Note that none of the existing optimizing compilers (commercial or research-prototypes) is able to simdize this loop currently in an efficient manner. On the contrary, we have shown that our algorithm is able to simdize about 90% of this loop and gain as much as 20% – 22% for speed and rate runs.

Our paper is organized as follows. Section 2 deals with background and related work. We detail our approach in Section 3. Section 4 outlines the experiments and results on AMD's platforms. We conclude in Section 5 and provide some possible future directions. Throughout the paper we will use the term simd(simdization,simdizable) loosely to represent vectorization of straight-line code that targets SIMD instructions.

2. Background and Related Work

Traditional loop-based vectorization has been a well researched area [7, 12]. Loop-level vectorization has been successfully adopted in several compilers like Open64 [3] and others such as Intel compiler (icc) [2], IBM xlC [6] and GNU compilers [20]. Some of their traditional vectorization capabilities have also been compared in [19].

The advent of bulk data processing applications in multimedia and graphics domains have influenced new processor architectures to provide special SIMD units to achieve better performance. Also, specialized instruction sets like SSE, AVX etc. have been introduced to facilitate SIMD operations. [13, 25] have discussed various loop transformations targeting the multimedia processors for simdization. However, traditional vectorization cannot exploit the SIMD architecture fully due to various programming complexities as well as compiler drawbacks [20]. For example, SIMD architectures facilitate packing/unpacking of non-contiguous data elements as well as specific operations for particular data types that are not exploited well by traditional loop-based vectorization techniques. Hence optimizing compilers are required to seek new techniques and approaches beyond traditional loop-based vectorization to exploit the advanced SIMD features of modern architectures.

Superword Level Parallelism (SLP) was introduced by Larsen and Amarsinghe [15] to capture a new set of simdization opportunities among the operations within a basic block. Unrolling of the innermost loop body in nested loop structures was also employed to create bigger basic blocks to expose additional parallelism and to identify more SLP opportunities. Following this work, there have been efforts to apply SLP across the basic block boundaries [26]. Shin et. al. [23, 24] have also exploited SLP in the presence of control flow using *if-conversion*. Nuzman and Zaks [21] have extended SLP based vectorization for the outer loops.

While [15–17] identified opportunities of vectorization on congruent and aligned data, there are advanced memory access SIMD instructions which can give performance improvements for various applications [8]. Hence there have been approaches which explore vectorization in presence of complex memory accesses. [22] has proposed vectorization of interleaved data accesses on SIMD instructions. [10, 11, 27] have vectorized programs with unaligned and irregular data accesses. There have also been several efforts [9, 14, 15, 18] to determine the optimal grouping of operations. [14] generates permutations for operations grouping. They have also used integer linear programming to generate the permutations of operations and perform a bi-directional traversal on data flow graph to pack the operations as SIMD instructions. Their approach is mainly based on the contiguous memory operations and cannot handle scalar operations. In [18] Leupers uses grammar rule based tree pattern matching on data flow tree along with integer linear programming formulation for code selection. However, our work is closest to Barik et al.'s approach [9]. In the following subsection we briefly introduce their approach.

2.1 Overview of Barik-Zhao-Sarkar Algorithm

In [9], the costs of operations are considered as the determining factor while deciding if an instruction should be executed in scalar or SIMD form. Therefore, a dynamic programming based approach is applied along with a specified cost model to arrive at the best possible operation groupings. It mainly comprises of the following steps:

- data dependence graph is built for each basic block.
- The cost model for performing scalar and SIMD computations is specified. This comprises of a set of rules based on how a node n in the data dependence graph can be computed depending on whether its predecessors are positioned as scalars or SIMD operations. There are two sets of rules - one for computing the scalar costs and the other to compute SIMD costs of k-tiles.
- *Pass 1.* The dependence graph is traversed from source to sink to compute the minimum scalar and vector costs for each possible scalar and k-tile.
- *Pass 2.* For each tile in the dependence graph the best costs are used to ascertain whether the tile should be executed as a SIMD operation. Non-tiled nodes are executed later as scalars.
- *Pass 3.* The dependence graph is traversed from sink to source and the tiles are used to generate SIMD instructions.

3. Our Approach

In this section we elaborate the various parts of our algorithm. We will also highlight the modifications made to the BZS algorithm for the Open64 compiler. The working of our algorithm is illustrated using an abbreviated example from a key loop in the SPEC 2006 benchmark [4], 435.gromacs, belonging to the subroutine inl1130. The key loop is shown in Figure 1. Our algorithm is sub-divided into the phases described next.

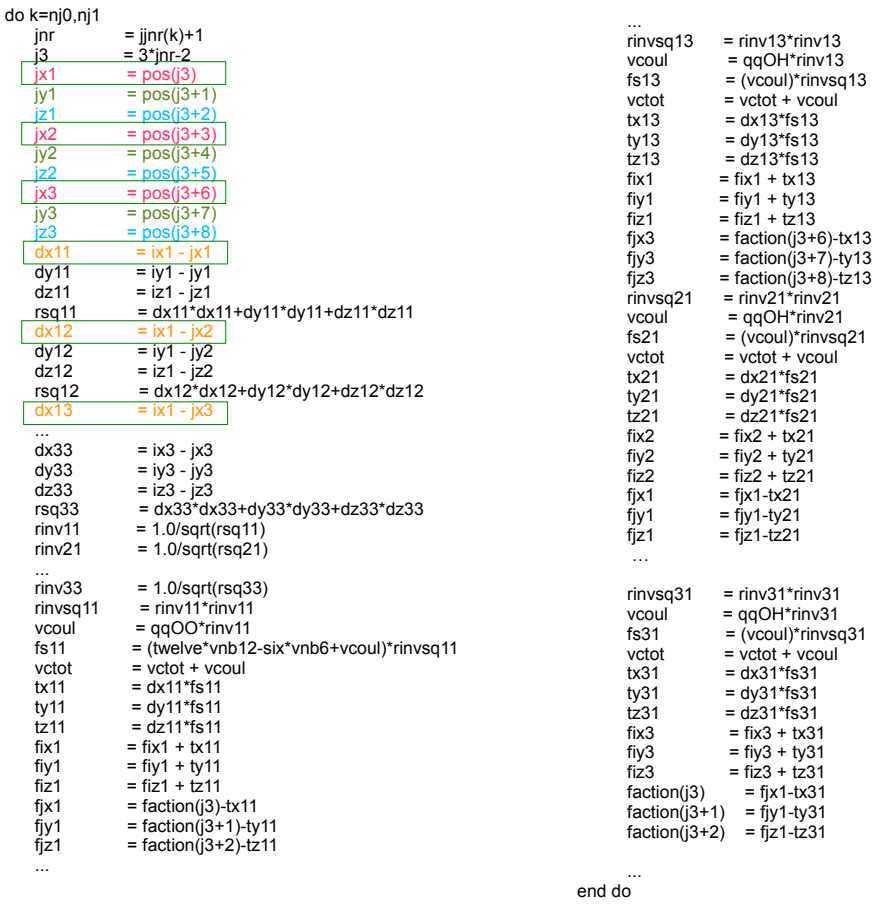


Figure 1. The key loop from inl1130 belonging to 435.gromacs

3.1 Phase I – Initialization

This is the preparatory phase of the partial simdization analysis. In this phase, we identify the inner loops of functions that contain only straight-line code i.e. loops without embedded control structures. The High-Level WHIRL representation of such a target loop is taken and a dataflow DAG is built where the nodes represent the operations in the WHIRL tree and the edges represent the data dependences. Nodes having exposed reads and writes that need to be available outside the loop are tracked specially for correct code generation. An important point to note here is that the DAG faithfully recreates the WHIRL tree. Store to a datum that is reloaded later is not short-circuited. This is because our analysis is being applied at a high level and we need not focus on load-store optimizations at this stage. During the DAG creation phase, we also assign scalar costs to operations (nodes). This is required for the dynamic programming phase later. The operation costs selected are very generic so that the algorithm can work well across a wide range of processors.

Once the DAG is ready, a topological sorting of the DAG is carried out and the nodes are placed at levels. The numbering of levels start at 0, for the nodes which have no children (predecessors) and end at the highest level, $maxLevel$, for nodes which do not have any successor. Placing nodes at levels is done to simplify the tiling process in Phase II. We will see later that the grouping of operations for simdization is done only among nodes which are at the same level. This is a major modification with respect to the BZS algorithm which works with a **ready list** for tiling operations that are simdizable. The ready list is a collection of nodes whose predecessors have been processed and all the data dependences have been satisfied. The presence of a ready list increases the compilation time several folds as tiling is a compute-intensive operation. Hence we segregate simdizable nodes into levels which control the number of nodes that can be tiled. This may result in a few lost simdization opportunities but the advantage in terms of compile time is quite high when the DAG is large.

The loop in Figure 1 is the inner loop of subroutine inl1130 and contains only straight-line code. Hence this loop is a good candidate for partial simdization. The DAG for this code is fairly large and contains around 700 nodes. A glimpse of how the DAG looks like with its topological levels is shown in Figure 2. The figure contains the nodes which are required till the construction of the $rsq11, \ldots, rsq33$ variables. The rest of the DAG is not illustrated due to its size. It is interesting to note that the variables $ix1, iy1, iz1, \ldots, ix3, iy3, iz3$ are $live - in$ as they are allocated in the enclosing loop (not shown in the example, the interested reader can look at the full source of inl1130). Also, there are several variables which are both $live - in$ and $live - out$. These include $vctot$ which actually turns out to be a **reduction** variable.

In Figure 2 the **ILOAD** nodes for loading the array elements $pos(j3), \ldots, pos(j3 + 8)$ are located at the zeroth topologically-sorted level (the first two statements that set the

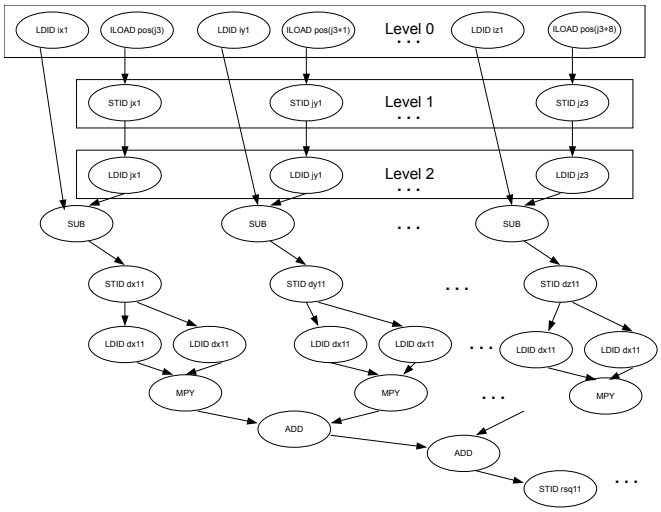


Figure 2. A part of the DAG from inl1130

index $j3$ are ignored. We will not discuss this any further in this paper). The **ILOAD**ed values are stored in the **STID** nodes $jx1, jy1, jz1, \ldots, jx3, jy3, jz3$ at level 1 followed by these values being loaded again at level 2. These are then subtracted from the live-in values $ix1, iy1, iz1$ at level 3. The subsequent results are stored in $dx11, dy11, dz11, \ldots, dx33, dy33, dz33$. These values are loaded again, multiplied and added to arrive at the $rsq11, rsq12, rsq13, \ldots, rsq31, rsq32, rsq33$ at topological level 9. The full DAG of inl1130's inner loop consists of more than 700 nodes and has more than 40 topological levels.

3.2 Phase II – Tiling

Tiling refers to packing isomorphic operations that are independent and which can be executed as a possible SIMD operation on a processor. For example, four integers of size 4-byte each can be packed in a tile of size 4, referred to as a 4-tile, if the SIMD width is 128-bits. If the SIMD width is 256-bits 8 such integers can be packed resulting in an 8-tile. Thus, there are two factors that actually control the simdization - a) the amount of SIMD parallelism that exists in the code and b) the SIMD width. If the amount of parallelism available exceeds the SIMD width, we can only pack up to the SIMD width. Similarly, if the SIMD width exceeds the parallelism available, we can pack only up to the available parallelism. While the first case is limited by the available SIMD width, in the second case we may employ additional loop unrolling to increase the available parallelism. However, in this work we will not look into loop unrolling techniques to drive additional parallelism.

In k-tiling we try to pack k isomporphic independent operations (of type o) as a tile represented by $t = \langle o^1, o^2, \ldots, o^k \rangle$. To do this, we traverse the DAG using the topological levels starting at level 0 and move to the maximum level. At each level we pick the operations which are isomorphic and then create all possible combinations of such operations, each such combination being a tile. For example in Figure 2, at level 0, there are 9 **ILOAD**s each corresponding to an access of the array pos. All these **ILOAD**s are isomorphic. If we want to create 2-tiles from these, we need to create $\binom{9}{2}$ tiles which represent all the possible combinations of these nodes. The dynamic programming phase will pick up these tiles and their associated costs to reach at the best possible solution later. Similarly, for a 3-tiling, we need to create all possible $\binom{9}{3}$ tiles. It is easy to see that if we have a large DAG this tiling procedure may lead to high compilation times. In order to control this we take the following steps - a) we do not create tiles beyond the size of $k = 4$. This restriction works well for most applications involving 32 and 64 bit data types. b) Secondly, we do not create tiles for isomporphic operations which share ancestors up to a level i.e. if some of the k isomporphic operations share a parent or a grandparent we do not pack these operations in a k-tile as they may not lead to good solutions when dynamic programming is applied.

As stated earlier, since the available SIMD parallelism may be lesser or greater than the SIMD width and may vary from one part of the code to the other, we employ the method of k-tiling where $k = 2, 3, 4, \ldots$. We choose k such that $k \leq S_w/S_t$ where S_w is the SIMD width and S_t is the size of the largest type used in the code. In inl1130, all operations are of type float and $S_w = 128$. This means $2 \leq k \leq 4$. Without delving into the details, inl1130 lends itself to the best possible simdization (without unrolling) when $k = 3$ because the amount of parallelism available in the loop shown in Figure 1 is 3. This means that when we pack 3 isomorphic 32-bit floating point operations in a SIMD register of size 128-bits we get the best results. A good k-tiling for a piece of code is achieved via an iterative process. However, in the current implementation we have arrived at the best solution offline and have added capabilities to handle 3-tiling only. Future work will involve handling the full capability of this iterative process.

After the tiling phase is executed, we have a collection of k-tiles for each level for each set of isomorphic operations. If the level is l and the isomorphic operation is o, we will refer to the list of such k-tiles at l for o, as T_l^o. Here $T_l^o = \bigcup t$ where each t is a k-tile. At each level we will refer to the set of all tiles as $T_l = \bigcup_o T_l^o$. The full set of tiles $T = \bigcup_l T_k$.

The BZS algorithm is based on the notion of k-tiling, where k is a static number. This may not always yield the best solution. Hence, in our work we relax the concept of tiling to extract the maximal SIMD parallelism from a piece of code using an iterative process. If applied to inl1130, the BZS algorithm would have tried to find 4-tilings of the loop body, which turns out to be sub-optimal. In the BZS algorithm, the k-tiles are not created before the dynamic programming phase is applied. Instead, they are created as the dynamic programming proceeds using a ready list of nodes whose predecessors have been computed. Our experience with this approach is that as the graph grows large, the list of nodes may explode leading to enormous compile time growth. Hence, we adopt a more conservative approach of using the independent operations at a particular topological level to drive the T_l^o sets. These sets are created prior to the dynamic programming phase.

3.3 Phase III – Dynamic Programming Application

The dynamic programming phase is level-based. It starts at level 1 of the DAG, examines all possible k-tiles at that level for each operation o, and calculates the best vector cost and scalar cost. The vector cost ($vcost$) corresponds to a tile t while the scalar cost ($scost$) is associated with a single node.

The final outcome of the dynamic programming algorithm is to find which operation needs to be scalar and which operations can be packed as k-tiles. In the best case all the nodes are part of some tile t while in the worst case all the nodes remain scalar. Packing and unpacking instructions are introduced at the zones where scalar operations need to be transformed to SIMD operations and vice-versa. The outline of our algorithm is given in **Find_KTileSimpleBestCost** in Figure 3.

When the DAG is created all the nodes of the DAG have an associated $scost$ that is initially set to the operator cost. Similarly when the k-tiles are created, these are associated with $vcost$s which are the costs of performing the k operations using a SIMD instruction (this initialization is not shown in Figure 3 for simplicity). The

Input: A DAG with topological levels and tiles
Output: The $scost$s and $vcost$s of the nodes and tiles
Procedure Find_KTileSimpleBestCost(DAG) {
1: // Iterate over all the levels starting at $l = 1$
2: // Assume 2 children of every operation
3: **foreach** $1 \leq l \leq maxLevel$ **do**
4: **foreach** (node $n \in DAG$) **do**
5: //ComputeScalarCost
6: **if** (children of n are both scalars)
7: nCost = scost(n) + scost(lchild(n)) + scost(rchild(n))
8: **else if** (both children of n are parts of k-tiles)
9: // assume t_1, t_2 to be the tiles left/right child is part of
10: // unpcost() is the cost of unpacking a k-tile
11: nCost = scost(n) + vcost(t_1) + vcost(t_2) + unpcost(t_1) + unpcost(t_2)
12: **endif**
13: **else if**
14: // some other cases of node matching handled here
15: ...
16: **endif**
17: **if** (nCost $<$ scost(n))
18: scost(n) = ncost
19: Remember the configuration of the good cost
20: **endif**
21: **endfor**
22: //ComputeVectorCost for each k-tile t
23: **foreach** (operator o) **do**
24: **foreach** (k-tile $t \in T_l^o$) **do**
25: // Let $t_1 = \langle o_1^1, o_1^2 \ldots o_1^k \rangle$
26: // Let $t_2 = \langle o_2^1, o_2^2 \ldots o_2^k \rangle$
27: t_1 = k-tile corresponding to the left predecessor tile of Fig 4
28: t_2 = k-tile corresponding to the right predecessor tile of Fig 4
29: **if** (TilesMatchFully(t, t_1, t_2))
30: pcost = vcost(n) + vcost(t_1) + vcost(t_2)
31: **else if**
32: // some other cases of tile matching handled here
33: ...
34: **endif**
35: **if** (pCost $<$ vcost(t))
36: vcost(t) = pcost
37: Remember the configuration of the good cost
38: **endif**
39: **endfor**
40: **endfor**
41: **endfor**
}

Figure 3. Level-based Dynamic Programming Algorithm.

$vcost$s used in our implementation are small integers that reflect the costs of the SIMD instructions on the processors. However, we have tried to keep them as generic as possible so that the dynamic programming algorithm is robust.

Given these initial sets of costs, our algorithm starts at the topological level of 1, skipping level 0 which consists of nodes that have no children. At level 1, we first compute the $scost$s of all the nodes. The $scost$ of a node is controlled by various dynamic programming matching rules as given in [9]. To explain a few of them (refer to the top part of Figure 4), assume that a node n has two children which are not part of any k-tile. In such a case the cost is computed as $scost(n) + scost(lchild(n)) + scost(rchild(n))$. In case both the children are part of some tiles t_1 and t_2, the cost can be computed as $scost(n)+vcost(t_1)+vcost(t_2)+unpcost(t_1)+unpcost(t_2)$, where $unpcost()$ is the cost of unpacking a tile so that the scalar components can be used. In case both the children are part of the same tile, the cost becomes $scost(n) + vcost(t) + unpcost(t) + unpcost(t)$ where t is the tile of which both the children are part of (this case is not shown in the algorithm). Thus, the scalar cost of node is computed from several matching rules and the **lowest cost** from all these rules is deemed to be the best $scost$ for n. The particular configuration or matching that yields the best cost is stored for later use.

After all the $scost$s of the nodes at a level are computed, we compute the $vcost$s of the tiles. For this, each tile t consisting of isomorphic operations on o, is subjected to a set of dynamic programming rules that closely resemble the scalar cost computation. To illustrate an important case, refer to Figure 4. In this case assume that the tile whose $vcost$ is being computed is $t = \langle o^1, o^2, \ldots, o^k \rangle$. Also assume we are trying to match this tile with two children tiles lying at levels lower than that of the level of t. Let these tiles be $t_1 = \langle o_1^1, o_1^2 \ldots o_1^k \rangle$ and $t_2 = \langle o_2^1, o_2^2 \ldots o_2^k \rangle$. The perfect match of a tile (which is checked by the function **TilesMatchFully**), is found by matching whether the left-child of o^i is o_1^i and the right-child is o_2^i. Now, the $vcost$ can be computed as $vcost(t) + vcost(t_1) + vcost(t_2)$. There are several variations of these matchings (not shown in the algorithm) which gives rise to various other $vcost$ values. Once again, the smallest of these costs is chosen and stored as the best vector cost estimate for tile t.

Our algorithm **Find_KTileSimpleBestCost** scans all the levels starting at level 1 and ending at the $maxLevel$. Once completed, the best $scost$s and $vcost$s of all the nodes are available. Also,

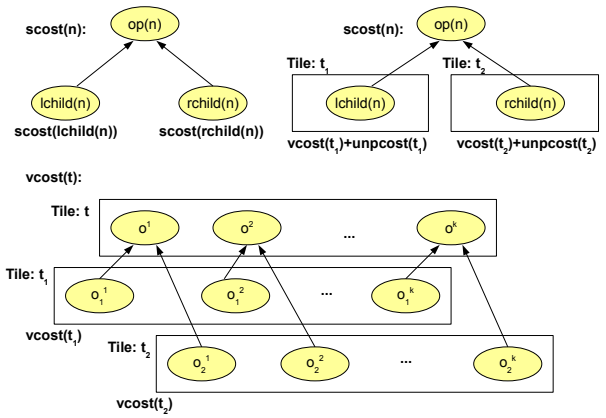


Figure 4. $scost$ and $vcost$ computation

we do not use all the dynamic programming rules provided in [9] because - a) applying all the rules increase the compile-time and b) some of these rules may only be useful in rare situations.

After the costing is done, we have to do the actual mapping of the DAG nodes to tiles (or scalars). Hence a top-down pass starting at the $maxLevel$ and descending to the leaf level is carried out similar to the BZS algorithm. At the highest level it picks those tiles that satisfy the property that its $vcost$ is better than combined $scost$s of all the constituent nodes. These tiles form the **seed tiles**. The best configuration (stored earlier) from these seed tiles are picked at $maxLevel - 1$ automatically. In addition, if other tiles satisfy the constraint that their $vcost$s are better than the combined $scost$s of the constituents nodes of the tiles, they are also picked as long they do not overlap with already-chosen tiles. When we reach the leaf level we would have picked all the good tiles. These are referred to as the **chosen** tiles. The nodes that are not part of any tile is left aside as scalar operations.

3.4 Phase IV – Code Generation

This is a phase of our implementation which differs radically from the BZS implementation as our algorithm is exercised at the High-Level WHIRL in LNO. Hence, the output of the code generation phase is a vectorized WHIRL tree. This is lowered later to the machine code by the existing code generation pass.

As described in the previous section, once we have figured which nodes are part of tiles and need to be simdized and which are not, we need to generate the SIMD code, the scalar code as well as the glue code that will allow packing and unpacking. For this purpose, we carry out a bottom up pass of the topological levels of the DAG, looking at all the tiles that are chosen. The nodes that are not parts of any tile are left untouched (they may be looked up later if their parents/children are tiled for the purpose of unpacking/packing). Once a tile is chosen for code generation it falls into one of the following classes - a) a regular operator tile like **add** b) a **LDID/ILOAD** tile whose nodes may or may not be adjacent in memory c) a **STID/ISTORE** tile whose nodes may or may not be adjacent in memory d) a tile whose parents are not tiled (i.e. scalars) e) a tile whose children are not tiled (i.e. scalars). In addition for load/store tiles the variables may be live-in or live-out or both, in which case we need to generate additional code outside the loop to set up the vectors (for live-in) or unpack the existing tiles to scalars (for live-out). We will now briefly describe each of the cases a) to e). However, before that we will dwell a bit on how the WHIRL tree can be transformed to reflect a simdized operation.

Assume a chosen 4-tile of 32-bit **ILOAD**s that happen from contiguous memory. Let the tile be denoted as $t = \langle a[i], a[i+1], a[i+2], a[i+3] \rangle$ for simplicity purposes. The WHIRL trees, wt_0, wt_1, wt_2 and wt_3 as shown in Figure 5. The trees are reachable from t via pointers maintained in t.

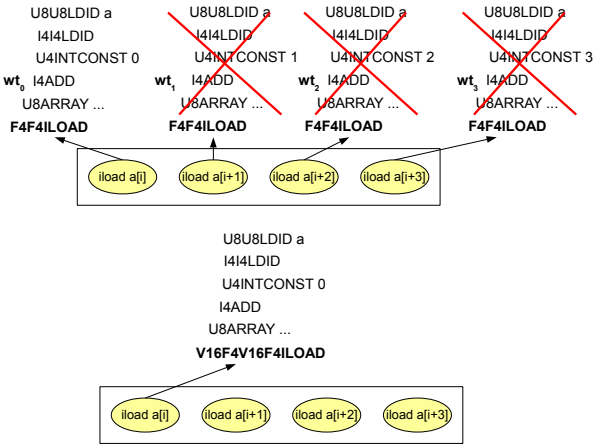


Figure 5. SIMD code generation via WHIRL transformation

If t is a chosen tile, and given that it is composed of contiguous memory accesses, our code generation mechanism transforms these WHIRL trees to a single tree. This is done by **deleting** the WHIRL trees wt_1, wt_2, wt_3 and changing the **desc** and **result** types of wt_0 (refer to [5] for types in WHIRL). The new WHIRL tree has a vector type **V16F4V16F4** where **V16** denotes 128-bit SIMD width. Later lowering phases already understand the vector types and generate the correct **vmov** [1] instruction for loading 4 contiguous 32-bits starting at $\&a[i]$. Similar type change for the first operator o^1 of a tile $t = \langle o^1, \ldots, o^k \rangle$ and removal of the WHIRL trees that correspond to the operators $o^2, \ldots, o^k$ are carried out for arithmetic operations (Case (a)).

3.4.1 Handling LDID/ILOAD tiles

We have already highlighted how the WHIRL tree is transformed for **ILOAD** tiles where the individual accesses are contiguous in memory. We will now discuss the case of **LDID/ILOAD** tiles where the accesses are not contiguous. Take the simple case of a 4-tile where $t = \langle w, x, y, z \rangle$ having four variables w, x, y, z being loaded. Since these are not contiguous in memory we need to load the variables independently and then pack them together as a tile. This cannot be easily accomplished by a simple type transformation. Instead we have to use special **unpck(lps)** and **shuf(ps)** instructions [1] supported by the underlying processor. Also, to carry out this transformation in WHIRL we have to a) associate a simd-temporary that will represent t and b) use Open64 suppported **intrinsic** calls which are finally lowered to the requisite unpck(lps) and shuf(ps) instructions. Figure 6 shows the original and transformed WHIRL trees. The transformed tree uses 1 unpck(lps) and 2 shuf(ps) operations. The first unpck(lps) creates the tile $\langle \_, \_, y, z \rangle$. The corresponding 2 shuf(ps) operations create the tile $\langle w, x, y, z \rangle$.

Later uses of the tile t are replaced by the loads to the new temporary (e.g. in Figure 6 the temporary is **psimd_v16_v1**). Also, all the WHIRL trees wt_0, wt_1, wt_2, and wt_3 are deleted and a new tree is created.

3.4.2 Handling STID/ISTORE tiles

Handling **ISTORE** tiles where the memory accesses are adjacent is analogous to the iload case with only the type of the store operator

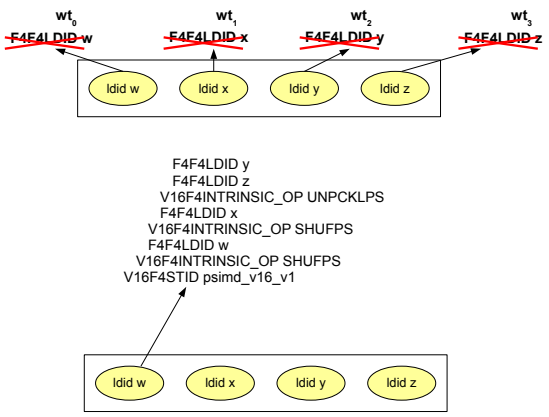
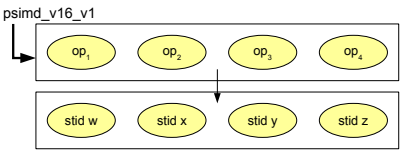


Figure 6. SIMD code generation for non-contiguous load via WHIRL transformation

changed to a SIMD type. However, when we have to take care of **ISTORE/STID** tiles where the accesses are non-contiguous we need to emit **pshufd** instructions (once again via intrinsics) [1]. These instructions shift basic elements of a SIMD register to position them for scalar moves to the variables which require them. Figure 7 shows how a store tile $t = \langle w, x, y, z \rangle$ which is currently packed as a SIMD temporary is stored to its constituent variables.




```
   V16F4<op> ...
V16F4STID psimd_v16_v1 // first split and store the value in a simd temp
// now store the elements individually using pshufd
   V16F4V16F4LDID psimd_v16_v1
   U4INTCONST 3 // shuf constant to position w to the lowest 32-bit
 V16F4INTRINSIC_OP PSHUFD
F4STID w
   V16F4V16F4LDID psimd_v16_v1
   U4INTCONST 2  // shuf constant to position x to the lowest 32-bit
 V16F4INTRINSIC_OP PSHUFD
F4STID x
   V16F4V16F4LDID psimd_v16_v1
   U4INTCONST 1 // shuf constant to position y to the lowest 32-bit
 V16F4INTRINSIC_OP PSHUFD
F4STID y
   V16F4V16F4LDID psimd_v16_v1
F4STID z // z is already at the lowest 32-bit so no shuf required
```

Figure 7. SIMD code generation for non-contiguous store via whirl transformation

It should be noted that before we can use **pshufd** instructions the SIMD operation which feeds into the store tile must be saved to a SIMD temporary during the WHIRL tree transformation. The temporary can then be loaded, and shuffled to position the individual parts for the scalar stores.

3.4.3 Handling tiles whose predecessor(s) or successor(s) are scalars

When a tile's predecessors nodes are scalar, we have to transform the WHIRL to pack these scalars into a tile. This follows a similar pattern of packing **LDID/ILOAD** operations which are not contiguous. The only difference is that we may need to break up existing trees (or make copies of them) and then pack them using a similar methodology as shown in Figure 6.

When a tile's successor tiles are scalar, we have to unpack the tile and feed the scalar values. This follows a similar pattern of unpacking **STID/ISTORE** operations which are not contiguous as shown in Figure 7.

There may be cases when parts of predecessors and/or successors are tiled while the rest are not. In such a case we need to pack/unpack the scalar nodes and use the tiled nodes as is.

3.4.4 Algorithm ApplySimd

ApplySimd is the main algorithm that has been implemented in our work to drive the partial simdization phase in LNO. Starting with the WHIRL tree representation of the inner loop, it builds the DAG and the topological levels using **CreateDAG** and **CreateLevelsForDAG** methods. **CreateVector_KTilesForLevels** is invoked next to build the k-tiles at each level. The dynamic programming framework is applied after that in **Find_KTileSimpleBestCost** which populates the tiles and nodes with their respective scalar and tiling costs. **FindBestVectorScalar_KTiles** traverses the levels and chooses the good tiles for simdization. The non-chosen nodes are left to be operated as scalars. Finally, **GenerateVectorScalarCode_KTile** is applied to generate the simdized WHIRL tree annotated with vector types and intrinsics. The algorithm is outlined in Figure 8.

3.5 Partial simdization of the inner loop of inl1130

Here we illustrate how our algorithm has been applied to the inner loop of inl1130 with significant gains in performance. We will highlight the major points here. We can observe that the loop has a 3-way SIMD parallelism. Hence, we apply 3-tiling to this loop for the best results. The 3-tiling is performed on the DAG of the inner loop. Based on the costs, the dynamic programming algorithm picks up the three non-adjacent accesses of $pos(j3), pos(j3+3), pos(j3+6)$ and pack them together as a non-contiguous **ILOAD** tile $\langle pos(j3), pos(j3+3), pos(j3+6) \rangle$. Similarly, the tiles $\langle pos(j3+1), pos(j3+4), pos(j3+7) \rangle$ and $\langle pos(j3+2), pos(j3+5), pos(j3+8) \rangle$ are formed. This tiling may appear non-intuitive at first because we do not pick the adjacent accesses of $pos(j3), pos(j3+1), pos(j3+2)$ etc. However, detailed cost analysis reveals that such a tiling may result in a semi-optimized schedule. We will see the reason why. The **ILOAD** tiles now feed into the **STID** tiles of the variables $jx1, jy1, \ldots$. For example, the $\langle pos(j3), pos(j3+3), pos(j3+6) \rangle$ tile feeds into the $\langle jx1, jx2, jx3 \rangle$ tile while the other two feed into the $\langle jy1, jy2, jy3 \rangle$ and $\langle jz1, jz2, jz3 \rangle$ tile respectively. Continuing in this fashion the tiles $\langle dx11, dx12, dx13 \rangle$, $\langle dy11, dy12, dy13 \rangle$ and $\langle dz11, dz12, dz13 \rangle$ are created. This results in $\langle rsq11, rsq12, rsq13 \rangle$ also being tiled. It is interesting to note that if instead of the non-contiguous **ILOAD** tiling that we started with, if we would have started with the contiguous tiling of $\langle pos(j3), pos(j3+1), pos(j3+2) \rangle$ and so on, we would have found $\langle dx11, dy11, dz11 \rangle$ in the same tile. This would have caused the problem of calculating the $rsq11 \ldots$ values because we would have needed an instruction like a horizontal add **hadd(ps)**, to sum up $dx11^2 + dy11^2 + dz11^2$ which is required to calculate $rsq11$. On most machines horizontal operations are costly. The costing mechanism of the dynamic programming algorithm avoids costly horizontal operations even it would mean incurring some packing cost(s) for non-contiguous operations.

When the tile $\langle dx11, dx12, dx13 \rangle$ is computed we require the live-in variable $ix1$ to be present in a tile $\langle ix1, ix1, ix1 \rangle$ in a replicated form. This is achieved by means of **pshufd** instructions placed at the outer loop. Similar replicated tiles are created for $\langle iy1, iy1, iy1 \rangle$ and $\langle iz1, iz1, iz1 \rangle$. Figure 9 shows some of the tiles created for the inner loop of inl1130. The tiles for the generated code are shown in a pseudo-form with angled brackets in Figure 9.

```
Input: WN - the WHIRL Tree of the loop
Output: Simdized WHIRL Tree
Procedure ApplySimd() {
1:      // Phase-I
2:      /* Initialization: create the DAG from the WHIRL using data flow analysis. */
3:      _dag = CreateDAG();
4:      /* level creation in DAG: topological ordering of the DAG nodes.*/
5:      _setOfLevels = CreateLevelsForDAG();
6:      // Phase-II
7:      /* possible vector tiles are created for each level. */
8:      _setOfTiles = CreateVector_KTilesForLevels();
9:      // Phase-III
10:     /* scalar and vector costs are computed according to
11:     the defined cost function using dynamic programming */
12:     _setOfTiles< scost, vcost > = Find_KTileSimpleBestCost();
13:     /* choose the best set of tiles for maximal benefits */
14:     _chosenSetOfTiles< scost, vcost > = FindBestVectorScalar_Ktiles();
15:     // Phase-IV
16:     /* generate the simdized code by transforming the WHIRL */
17:     _simdizedWHIRL = GenerateVectorScalarCode_KTile();
}
```

Figure 8. ApplySimd Algorithm.

We handle reduction variables like $vctot$ as a separate pass over the DAG after the simdization has been done. In general, such a pass may expose new simdization opportunities that need to be handled in an iterative manner. This has been postponed to a future work.

```
//Live-In code
l1 = <ix1,ix1,ix1>
l2 = <iy1,iy1,iy1>
l3 = <iz1,iz1,iz1>
...
do k = nj0, nj1
   t1 = <jx1,jx2,jx3>
   t2 = <jy1,jy2,jy3>
   t3 = <jz1,jz2,jz3>
   …
   d1 = l1-t1 // d1 is <dx11,dx12,dx13>
   d2 = l2-t2 // d2 is <dy11,dy12,dy13>
   d3 = l3-t3 // d2 is <dz11,dz12,dz13>
   …
   // r1 is <rsq11,rsq12,rsq13>
   r1 = d1*d1 + d2*d2 + d3*d3
   …
   rinv1 = rsqrt(r1) // rinv1 = <rinv11,rinv12,rinv13>
                     // rsqrt = 1/sqrt
   …

end do
//Live-Out code
  // Unpack vctot etc...
```

Figure 9. SIMD code generation for il1130 - a snapshot

In the latter half of the inner loop of inl1130 there are issues of replicating variables (ex: $fs11$) to enable SIMD operations as well as cases where an original tiling like $\langle dx11, dx12, dx13\rangle$ may need to be unpacked and packed into a new tile of the form $\langle dx11, dy11, dz11\rangle$. To handle the latter we enable special group-matching of tiles which we call **supertiling**. In supertiling, $\langle dx11, dx12, dx13\rangle$, $\langle dy11, dy12, dy13\rangle$ and $\langle dz11, dz12, dz13\rangle$ are unpacked and packed into $\langle dx11, dy11, dz11\rangle$, $\langle dx12, dy12, dz12\rangle$ and $\langle dx13, dy13, dz13\rangle$ using a very compact set of instructions instead of individually unpacking and packing them. Issues like supertiling, handling replicated variables for better partial simdization etc. are handled as separates passes over the DAG.

4. Experiments and Results

Our algorithm has been made available in the Open64 4.5.1 release from AMD. At present we have utilized the flag -LNO:simd=3 to drive this optimization. An initial version of partial simdization was already available under this flag for an earlier Open64 release, hence we have reused the same flag. We have experimented our new compiler primarily on 435.gromacs by using the -LNO:simd=3 flag resulting in substantial improvement of the runtime of this benchmark. inl1130 is the hottest subroutine of 435.gromacs and consumes 65-70% of the runtime. We are able to simdize around 90% of the inner loop of inl1130 that results in the performance improvement reported in this work. The rest of the subroutines are only moderately hot. The details of DAG of the inner loop of inl1130 along with its size (#N), topological levels(#L), number of nodes simdized(#N Simdized) etc. is provided in Table 1. The number of DAG nodes is 763. The number of levels in the DAG is 43. The number of 3-tiles identified is 226 which comprises of 678 DAG nodes, effectively simdizing around 90% of the loop.

Table 1. Simdization Analysis Result of inner loop of inl1130

#N	#L	#N Simdized	#3-tiles Simdized	% Simdized
763	43	678	226	88.8

The performance results for the runs of 435.gromacs are shown in Table 2. We have used the **peak** options of SPEC for running this benchmark, with -LNO:simd=3 turned on. Our experiments use 32 Core $2.5GHz$, 64-bit AMD Bulldozer processors and $128GB$ DDR3 RAM running SLES11 SP1 Linux operating system. The table illustrates the execution times and speedups achieved for the simdized version when compared against the non-simdized version. We have carried 1-copy, 16-copy and 32-copy runs which correspond to both the speed as well as the rate runs of SPEC [4]. Note that the runtimes listed in Table 2 refer to the runtime of the entire benchmark and not just inl1130. It can be seen that we have achieved speedups in the range 21%–22% for the entire benchmark which is indeed impressive.

In our framework we employ partial simdization only when traditional vectorization fails. Also, no other transformations such as loop unrolling etc. were used to facilitate partial simdization. This signifies the importance and usefulness of adopting the partial simdization framework in the Open64 compiler.

Table 2. Comparison of executions of Simdized and Non-simdized version of gromacs

Gromacs Exec.	Non-Simdized	Simdized	Improvement
#copies	Time(sec)	Time(sec)	% time
1	415	341	21.7
16	443	366	21
32	562	462	21.6

5. Conclusion and Future Work

In this paper we have outlined our experience in implementing a dynamic programming based algorithm by Barik et al. [9]. Our work utilizes the Open64 middle-end optimizer infrastructure for partially simdizing inner loops. Our algorithmic implementation differs substantially from the original published work in several key areas. We have run our adopted algorithm on 435.gromacs from the SPEC CPU2006 suite and have achieved significant improvement over the baseline non-simdized code. We have shown that a dynamic programming based approach holds sufficient promise, though we may need to tune aspects of the algorithm for better compile-time overheads.

Our future work involves tuning the algorithm further for lower compile-time overhead. In addition, we are bolstering our algorithm for the iterative formulation to extract the best k value for tiling. We are also investigating other benchmarks for possible application of this new framework. Partial simdization effectiveness may be boosted by unrolling of a loop if it is not unrolled already. In general, finding an optimal unroll-factor for improving partial simdization is an open problem. However, we are already investigating ways and means to unroll by small factors when we find that the existing partial simdization opportunity in a loop is low.

Acknowledgments

The authors would like to thank Tim Wilkens and Michael Berg of AMD for identifying various simdization opportunities and encouraging us to implement it in the compiler.

References

[1] *Using the x86 Open64 Compiler Suite*. http://support.amd.com/us/Processor_TechDocs/26568_APM_v4.pdf. 2011.

[2] *The Software Vectorization Handbook*. Intel Press. 2004.

[3] *AMD64 Architecture Programmer's Manual Volume4 : 128-Bit and 256-Bit Media Instructions*. http://developer.amd.com/tools/open64/Documents/open64.html. 2011.

[4] *SPEC:Standard Performance Evaluation Corporation*. http://www.spec.org.

[5] *Open64 Compiler : Whirl Intermediate Representation*. http://www.mcs.anl.gov/OpenAD/open64A.pdf.

[6] *Code optimization with the IBM XL Compilers*. IBM. 2010.

[7] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9:491–542, October 1987.

[8] M. Alvarez, E. Salamí, A. Ramírez, and M. Valero. Performance impact of unaligned memory operations in simd extensions for video codec applications. In *IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS, pages 62–71, 2007.

[9] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 201–212, Washington, DC, USA, 2010. IEEE Computer Society.

[10] H. Chang and W. Sung. Efficient vectorization of simd programs with non-aligned and irregular data access hardware. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 167–176, New York, NY, USA, 2008. ACM.

[11] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 82–93, New York, NY, USA, 2004. ACM.

[12] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[13] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *International Journal Parallel Program.*, 28:pages 347–361, August 2000.

[14] A. Kudriavtsev and P. Kogge. Generation of permutations for simd processors. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '05, pages 147–156, New York, NY, USA, 2005. ACM.

[15] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[16] S. Larsen, E. Witchel, and S. Amarasinghe. *Techniques for Increasing and Detecting Memory Alignment*. Tech. Report. MIT-LCS-TM-621, 2001.

[17] S. Larsen, E. Witchel, and S. P. Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 18–29, Washington, DC, USA, 2002. IEEE Computer Society.

[18] R. Leupers. Code selection for media processors with simd instructions. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '00, pages 4–8, New York, NY, USA, 2000. ACM.

[19] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 372–382, 2011.

[20] D. Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[21] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[22] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM.

[23] J. Shin. Introducing control flow into vectorized code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 280–291, Washington, DC, USA, 2007. IEEE Computer Society.

[24] J. Shin, M. Hall, and C. Jacqueline. Superword-level parallelism in the presence of control flow. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society.

[25] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal Parallel Program.*, 28:pages 363–400, August 2000.

[26] H. Tanaka, Y. Ota, N. Matsumoto, T. Hieda, Y. Takeuchi, and M. Imai. A new compilation technique for simd code generation across basic

block boundaries. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 101–106, Piscataway, NJ, USA, 2010. IEEE Press.

[27] P. Wu, A. E. Eichenberger, and A. Wang. Efficient simd code generation for runtime alignment and length conversion. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 153–164, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X.