Speculative Dynamic Vectorization to Assist Static Vectorization in a HW/SW Co-designed Environment

Rakesh Kumar[1], Alejandro Martínez[2], Antonio González[1, 2]
[1] Department of Computer Architecture, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain
[2] Intel Barcelona Research Center, Intel Labs, 08034, Barcelona, Spain
rkumar@ac.upc.edu, alejandro.martinez@intel.com, antonio.gonzalez@intel.com

Abstract—**Compiler based static vectorization is used widely to extract data level parallelism from computation intensive applications. Static vectorization is very effective in vectorizing traditional array based applications. However, compilers inability to reorder ambiguous memory references severely limits vectorization opportunities, especially in pointer rich applications. HW/SW co-designed processors provide an excellent opportunity to optimize the applications at runtime. The availability of dynamic application behavior at runtime will help in capturing vectorization opportunities generally missed by the compilers.**

This paper proposes to complement the static vectorization with a speculative dynamic vectorizer in a HW/SW co-design processor. We present a speculative dynamic vectorization algorithm that speculatively reorders ambiguous memory references to uncover vectorization opportunities. The hardware checks for any memory dependence violation due to speculative vectorization and takes corrective action in case of violation. Our experiments show that the combined (static + dynamic) vectorization approach provides 2x performance benefit compared to the static vectorization alone, for SPECFP2006. Moreover, dynamic vectorization scheme is as effective in vectorization of pointer-based applications as for the array-based ones, whereas compilers lose significant vectorization opportunities in pointer-based applications.

Keywords— HW/SW Co-designed processor, Vectorization, Speculation, Dynamic optimizations

I. INTRODUCTION

Single Instruction Multiple Data (SIMD) accelerators form an integral part of modern microprocessors. These can be found in processors from different computing domains like general purpose processors [2] [9] [16], Digital Signal Processors [5], gaming consoles [13] [25] as well as embedded architectures [6]. SIMD accelerators are tailored to exploit data level parallelism from modern multimedia, scientific and throughput computing applications. Since these accelerators perform the same operation on multiple pieces of data, they just require duplicated functional units and a very simple control mechanism. Due to this simplicity, SIMD accelerators grow in size with each new generation. For example, Intel´s MMX [2] had vector length of 64-bits, which was increased to 128-bits in SSE extensions [2]. Intel´s recent SIMD extension AVX [2] and Intel´s Xeon Phi [4] supports 256-bit and 512-bit vector operations respectively.

Code generation for SIMD extensions has always been challenging. In the early days, programmers used to target these extensions mainly using in-line assembly or specialized library calls. Then, automatic generation of SIMD instructions (auto-vectorization) was introduced in compilers [7][18], which borrowed their methodology from vector compilers. These compilers target loops for generating code for SIMD accelerators. Later, S. Larsen et al. [14] introduced Superword Level Parallelism (SLP) in which they target basic blocks instead of whole loops for vectorization. These static approaches to vectorization are effective for traditional applications where memory is referenced through explicit array accesses, whereas modern applications make extensive use of pointers. Due to this, disambiguation of a pair of memory accesses becomes difficult at compile time. Since memory operations form the foundation of vectorization, current static approaches are limited in extracting SIMD parallelism.

In this paper, we propose to have dynamic vectorization as a complimentary optimization to the compiler based static vectorization. It is important to note that we do not propose to eliminate static vectorization altogether because there are several complex and time consuming transformations which are not straightforward to apply at runtime and are too costly like loop distribution, loop interchange, loop peeling, memory layout change, algorithm substitution etc. However, static vectorization alone fails to capture significant vectorization opportunities due to conservative pointer disambiguation analysis. To handle these cases we propose to have a speculative dynamic vectorizer which can speculatively reorder ambiguous memory references to uncover vectorization opportunities. Moreover, in the absence of loops, the scope of vectorization for static vectorization is a single basic block. We propose to vectorize bigger code regions which include multiple basic blocks and can be created at runtime following the biased direction of branches.

Furthermore, we propose a speculative dynamic vectorization algorithm which can be implemented in the software layer of a HW/SW co-designed processor[1]. The proposed algorithm speculatively reorders and vectorizes memory operations. During execution, the hardware checks for any memory dependence violations caused by speculative vectorization. If any violation is detected, the hardware rolls back to a previously saved check-point and executes a non-speculative version of the code. The hardware support required for speculative execution is already provided by co-designed processors like Transmeta Crusoe [10], BOA [23] etc. Therefore, no additional hardware support is needed from speculative vectorization point of view. This hardware support is also one of the reasons for choosing HW/SW co-designed processors over dynamic binary optimizers in our proposals.

Moreover, in the absence of static compiler vectorization, our algorithm can work as a standalone vectorizer also.

[1] Section II BACKGROUND provides the background about HW/SW Co-designed Processors.

Therefore, the legacy code which was not compiled for any SIMD accelerator can be vectorized using the proposed algorithm. The co-designed nature of the processor makes the vectorization portable. As a result, the algorithm can be modified to transparently target a different SIMD accelerator. It is important to note that the proposed algorithm does not require any compiler or operating system support/modification. The main contributions of this paper can be summarized as:

- Proposes a complementary dynamic vectorization to the static compiler vectorization.
- Proposes to increase the vectorization scope utilizing the dynamically discovered control flow: biased branch directions and dynamic loop trip counts.
- A runtime speculative vectorization algorithm :
    - that is equally good in vectorizing array and pointer based applications.
    - that is able to vectorize legacy code.
- Experimental evaluation of the proposed algorithm and it's comparison with GCC vectorizer.

The rest of the paper is organized as follows: Section II provides a background on HW/SW co-designed processors. Section III briefly provides the motivation for the work presented in this paper. Section IV describes the proposed algorithm with an example. Section V explains the speculation and recovery mechanism. Evaluation of the algorithm using SPECFP2006, Physicsbench and UTDSP applications is presented in Section VI. Section VII presents the related work and Section VIII concludes.

II. BACKGROUND

HW/SW Co-designed processors [10][23] employ a software layer that resides between the hardware and the operating system. This software layer allows host and guest ISAs to be completely different, by translating the guest ISA instructions to the host ISA dynamically. We define host ISA as the ISA which is implemented in the hardware, whereas, guest ISA is the one for which applications are compiled. The basic idea behind these processors is to have a simple host ISA to reduce power consumption and complexity.

The software layer translates the guest ISA instructions to the host ISA in multiple phases. Generally, in the first phase, guest ISA instructions are interpreted. In the rest of the phases, guest code in translated and stored in a code cache, after applying several dynamic optimizations, for faster execution. The number of translation phases and optimizations in each phase are implementation dependent.

Hardware support is needed for efficient and correct emulation of the guest ISA instructions. Memory speculation is the key to several optimizations performed by HW/SW co-designed processors. To ensure the correctness of memory speculation, hardware support is provided to detect speculation failure and recover from it. Furthermore, hardware support is necessary for providing precise exceptions and detecting self-modifying code. Moreover, overhead of indirect branches and function returns can be reduced by having some hardware support [15].

III. MOTIVATION

Traditional compile time loop vectorization is effective for applications involving explicit array accesses since memory dependence analysis are relatively easy. Significant performance gains have been reported using compiler vectorization in the past[7][14]. However, one of the major obstacles in vectorization at compile time is memory disambiguation and dependence testing. J. Holewinski et. al. [12] showed that static vectorization fails to extract significant vectorization opportunities especially in pointer-based applications. Furthermore, S. Maleki et al. [17] showed that the modern compilers, including Intel ICC, IBM XLC and GNU GCC, are limited in vectorizing modern applications. Extensive use of pointers and pointer arithmetic in these applications complicate memory disambiguation and dependence testing. Even though research shows that a pair of memory accesses rarely alias until and unless aliasing is obvious [11], compilers generate conservative code to ensure correctness which limits vectorization opportunities [22]. For example, Figure 1a shows a loop that performs pointer arithmetic. During compilation, if the compiler cannot prove that the two pointers always reference different memory locations, this loop cannot be vectorized.

```
void example(double *a, double *b)
{
  int i;
  for (i = 0; i < NUM_ITR; i++)
    a[i] += b[i] * CONST;
}
```

a) An example loop with pointers

```
loop:   I0    ld_64    v2, M [r2 + r1 * 8]
        I1    mulsd    v3, v2, v1
        I2    ld_64    v4, M [r3 + r1 * 8]
        I3    addsd    v5, v4, v3
        I4    st_64    v5, M [r3 + r1 * 8]
        I5    add      r4, r1, 1
        I6    ld_64    v6, M [r2 + r4 * 8]
        I7    mulsd    v7, v6, v1
        I8    ld_64    v8, M [r3 + r4 * 8]
        I9    addsd    xmm0, v8, v7
        I10   st_64    xmm0, M [r3 + r4 * 8]
        I11   add      r1, r4, 1
        I12   cmp      r1, r0
        I13   jne      loop
```

b) Unrolled lower level representation

```
        I0    Pack[2]           v1, v1, v1
loop:   I1    ld_128_spec       v2, M [r2 + r1 * 8]
        I2    mulpd             v3, v2, v1
        I3    ld_128            v4, M [r3 + r1 * 8]
        I4    addpd             v5, v4, v3
        I5    st_128_spec       v5, M [r3 + r1 * 8]
        I6    add               r1, r1, 2
        I7    cmp               r1, r0
        I8    jne               loop
        I9    Unpack            xmm0, v5
```

c) Speculatively vectorized version

Figure 1. An Example Loop with pointer arithmetic.

[2] Pack/Unpack instructions are explained in Section IV B) Vectorization.

As stated before, a recent approach to vectorization, SLP [14], performs vectorization at basic block level. Whereas traditional loop vectorizers vectorize either whole loop or nothing, SLP may vectorize portions of a loop if the whole loop is not vectorizable. SLP starts by identifying adjacent memory accesses and then follows their def-use and use-def chains. Figure 1b shows low level code for the loop of Figure 1a after unrolling it once. In this case even though I0 and I6 are adjacent memory references, they cannot be packed by SLP since I4 and I6 may alias. Thus, memory dependences affect both traditional loop vectorizers as well as modern SLP.

One possible solution that compilers may provide is to generate two versions of the loop: one without vectorization and another vectorized with a runtime test to check for aliasing. However, this solution is not optimal because: 1) runtime test has to be executed every time before executing the loop, thus resulting in performance loss. Moreover, as the number of arrays to be checked for aliasing increases the number of checks to be performed also increases.
2) Having multiple versions of the loop increases the static code footprint of the application, which results in higher instruction cache size requirements.

Another way of vectorizing the example loop is through "__restrict annotation". However, it requires source code modification which is not always possible e.g. unavailability of the source code or any other reason. In contrast, the proposed mechanism does not require any source code modification. Moreover, the "__restrict annotation" can not help in vectorization of the loops with complicated memory dependence. We choose a simple loop in this example to easily explain the proposed vectorization algorithm in Section IV D.

HW/SW Co-designed processors provide an excellent opportunity to handle these cases: instead of generating multiple versions, a single speculatively vectorized version can be generated by the software layer and the hardware can be tailored to execute the vectorized code efficiently and safely. The proposed algorithm speculatively reorders memory operations to expose vectorization opportunities. For the example code of Figure 1b, our algorithm speculatively assumes that I4 and I6 will never alias and reorders them to pack I0 and I6 together, as shown in Figure 1c. Moreover, due to the speculative reordering, I1 is converted to a speculative load and I5 to a speculative store (in vectorized code). If during the execution it turns out that I1 and I5 access overlapping memory locations, the hardware will detect this condition and will take corrective measures. In this example, by vectorizing speculatively we are able to vectorize the whole loop, whereas loop vectorization and SLP could not find vectorization opportunities.

Therefore, having two complementary vectorizing schemes helps to get the best of both the worlds. Static vectorization applies more complex and time consuming loop transformations whereas dynamic vectorization speculatively vectorizes ambiguous memory references and dependent operations.

IV. Vectorization Algorithm

This section provides the details of the proposed vectorization scheme. Before explaining the vectorization algorithm itself, first we explain binary translation/optimization steps of our (a typical) HW/SW co-designed processor. It helps us understand the context in which vectorization is done.

The software layer of our co-designed processor is called Translation Optimization Layer (TOL). TOL operates in three translation modes for generating host code from guest x86 code: Interpretation Mode (IM), Basic Block Translation Mode (BBM) and Superblock Translation Mode (SBM). Vectorization is done in SBM, which is the most aggressive translation/optimization level, after applying several standard compiler optimizations.

A. *Pre-Vectorization Steps*

Before starting with vectorization we create a superblock, apply standard compiler optimizations on the superblock and generate a Data Dependence Graph (DDG). Each of these steps is explained below:

1) Superblock Creation: TOL starts by interpreting guest x86 instruction stream in IM. When a basic block is executed more than a predetermined number of times, TOL switches to BBM. In this mode, the whole basic block is translated and stored in the code cache and the rest of the executions of this basic block are done from the code cache. Moreover, branch profiling information for direction and target of branches is also collected. Once the execution of a basic block exceeds another predetermined threshold, TOL creates a bigger optimization region, called superblock, using the branch profiling information collected during BBM.

A superblock generally includes multiple basic blocks following the biased direction of branches. Moreover, branches inside the superblocks are converted to "asserts" so that a superblock can be treated as a single-entry, single-exit sequence of instructions. This gives the freedom to reorder and vectorize the instructions from multiple basic blocks. "Asserts" are similar to branches in the sense that both checks a condition. Branches determine the next instruction to be executed based on the condition, however asserts have no such effect. If the condition is true assert does nothing. However, if the condition evaluates to false, the assert "fails" and the execution is restarted from a previously saved checkpoint in IM. Furthermore, while creating a superblock, if a loop is detected, it is unrolled. Currently, we unroll loops with a single basic block.

2) Pre-optimizations: A pre-optimization phase applies several conventional compiler optimizations in order to remove the dead code and improve the quality of the code. First of all, the superblock is converted into Static Single Assignment (SSA) form to remove anti and output dependences. Then, the optimizations Constant Propagation, Copy Propagation, Constant Folding, Common Sub-expression Elimination and Dead Code Elimination are applied.

The next step is to generate the Data Dependence Graph (DDG). During DDG creation, we perform memory disambiguation analysis and consecutiveness analysis. If memory disambiguation analysis cannot prove that a pair of memory operations will never/always alias, it is marked as "may alias". In case of reordering, the original memory instructions are converted to speculative memory operations. Similarly, consecutiveness analysis finds adjacent memory accesses and marks them for vectorization. Apart from this, Redundant Load Elimination and Store Forwarding are also applied during DDG phase so that redundant memory operations are removed before vectorization.

B. Vectorization

The vectorizer packs together a number of independent scalar instructions, which perform the same operation, and replaces them with one vector instruction; the number of scalar instructions packed depends on the data-types of scalar instructions. Therefore, vectorization reduces dynamic instruction count and improves performance. Before describing the algorithm, we define a set of conditions that a pair of instructions must satisfy to be included in the same pack:

- The instructions must perform the same operation.
- The instructions must be independent.
- The instructions must not be in another pack.
- If the instructions are load/store, they must be accessing consecutive memory locations.

Vectorization starts by marking all the instructions which are candidates for vectorization. Moreover, we mark First Load and First Store instructions. First Load/Store instructions are those for which there are no other loads/stores from/to adjacently previous memory locations. For example, if there is a 64-bit load instruction I_L that loads from a memory location [M] and there is no 64-bit load instruction that loads from address [M – 8], we call I_L First Load.

Vectorization begins by packing consecutive stores, starting from a First Store. The decision of starting with stores instead of loads is based on the observation that a given kind of operation always has the same number of predecessors, e.g. all the additions always have two predecessors, whereas the number of successors may vary depending on how many instructions consume the result. Consequently, following a bottom-up approach results in a more structured tree traversal than a top-down approach.

Once a pack of stores is created, their predecessors are packed, before packing other stores, if they satisfy the packing conditions. Moreover if the last store in the pack has a next adjacent store, it is marked as First Store so that a new pack can start from it.

Once all the stores are packed and their predecessor/ successors chains have been followed, we check for remaining load instructions that satisfy the packing conditions and pack them in the same way as stores.

While traversing the predecessor/successor chains, if we find out that the predecessors of a pack cannot be vectorized, a Pack instruction is generated. This Pack instruction collects the results of all the predecessors into a single vector register and feeds the current pack. Similarly if all the successors of a pack cannot be vectorized, an Unpack instruction is generated. This Unpack instruction distributes the result of the pack to the scalar successor instructions.

Moreover, Pack instructions are needed if a pack contains an instruction whose input is live-in of the superblock. Similarly, Unpack instructions are needed to put the results from a pack to the architectural registers which are live-outs of the superblock.

C. Static vs Dynamic Vectorization

Loops are the basic program structures that the vectorizers target for extracting parallelism through vectorization. Several loop transformations are sometimes needed to make a loop vectorizable. The transformation like loop distribution, loop interchange, loop peeling, node splitting, memory layout change, algorithm substitution etc are generally applied to make a loop vectorizable. These time consuming transformations are better suited at compile time than at runtime. However, compile time vectorization suffers from several limitation like: 1) limited vectorization opportunities due to conservative memory disambiguation analysis, 2) scope of vectorization is limited to basic blocks if the loops cannot be unrolled e.g. due to complex control flow and 3) legacy code cannot be vectorized.

The proposed speculative dynamic vectorization gets rid of all these limitations. The proposed algorithm relaxes the restrictions on memory disambiguation by speculatively reordering/vectorizing the ambiguous memory references. Since the scope of vectorization for the proposed algorithm is a superblock, it crosses the basic block boundaries to vectorize instructions from multiple basic blocks. Moreover, for the loops, where the number of iterations are not known statically, it is difficult to decide the unroll factor at compile time. The availability of dynamic application behavior, at runtime, allows to detect the loop unroll factor dynamically. Furthermore, since the dynamic vectorization is applied at runtime on the program binary and not at the source code level, the legacy code can also be vectorized. Therefore, the combination of static and dynamic vectorization extracts the maximum vectorization opportunities.

D. Working Through an Example

Figure 2a shows the DDG for the example code of Figure 1b. Since the loop is unrolled once and there is no loop carried dependences, assumed speculatively, the two trees are completely separated from each other. For the sake of simplicity, we do not show loop control code in this figure. Also, pairs of ambiguous memory reference instructions like I4 and I6 are considered independent speculatively. As our algorithm begins with consecutive stores, the stores I4 and I10 are packed together as shown in Figure 2b. Moreover, the new store instruction is speculative one and I6 is also converted to speculative load. Following the predecessor tree, we see that I3 and I9 satisfy the packing conditions and vectorize them. Notice here that I9 writes to a live-out architectural register. As a result, we have to generate an Unpack instruction to write the result to the live-out register. This is shown in Figure 2c.

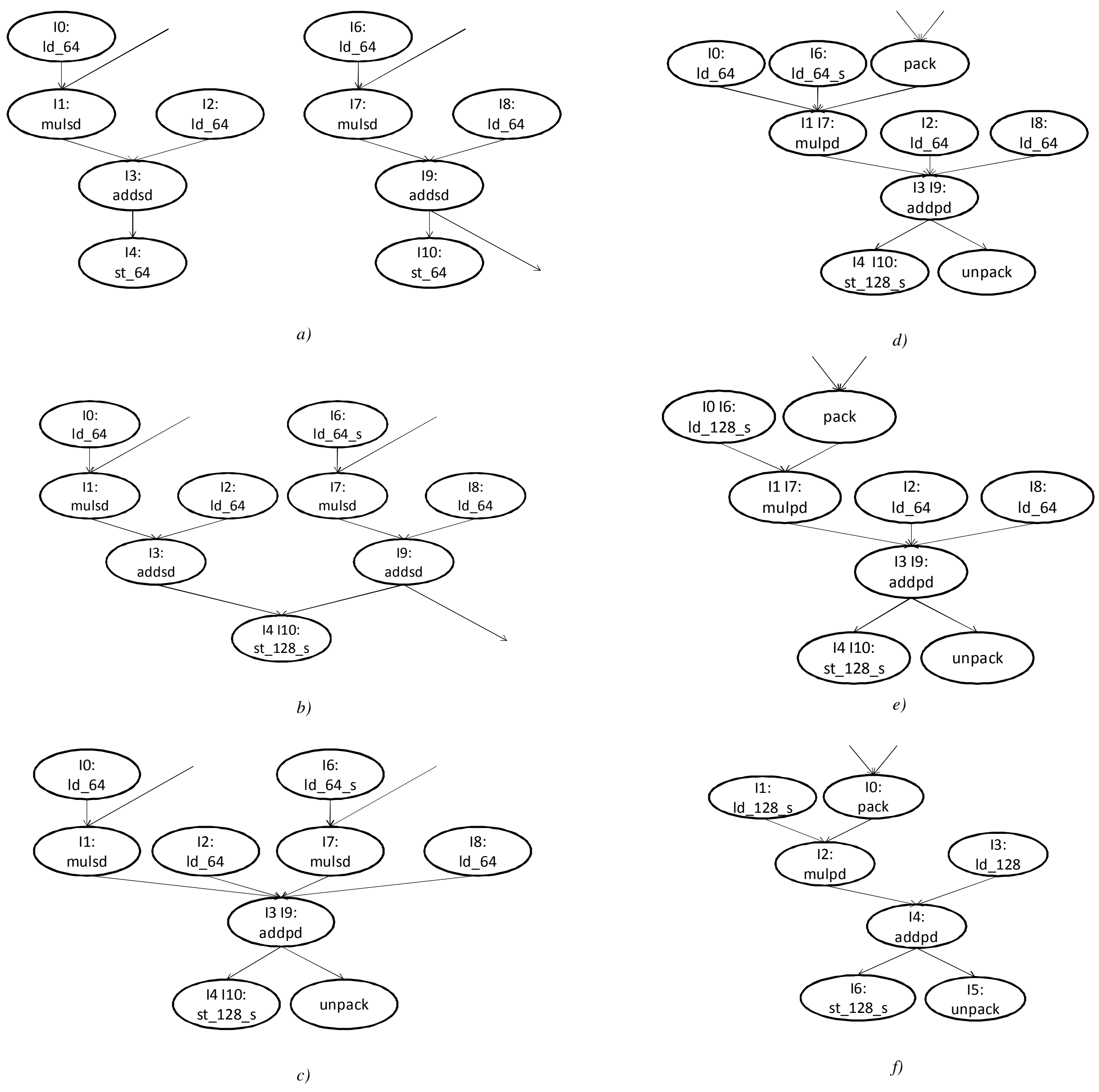

Figure 2. Example for vectorization of the code of Figure 1b. a). Shows the DDG for the loop which is unrolled once. We don´t show loop control code for the sake of simplicity. Since two iterations are completely independent we have two completely separated trees. Two arrows coming in to I1 and I7 represents live-in and arrow going out of I9 represents live-out of the superblock. Also, speculatively, we assume there is no dependence between the memory instructions until its obvious b) Shows the state of DDG after vectorizing consecutive stores, also, the new store instruction is speculative one. c) Then, we follow the predecessor chain and pack addsd instructions. Since I9 writes to an architectural register, we need to unpack the results and write to the architectural register. d) Packs two mulsd instructions and since one of the inputs to both of these instructions is a live-in, a Pack instruction is also generated to pack the inputs in a single vector register. e) and f) pack remaining load instructions and f) Shows the final state.

Traversing up the tree we vectorize multiplication instructions I1 and I7. One of the inputs of the multiplication instructions is a live-in to the superblock. Hence, we generate a Pack instruction to put the live-in values in a vector register as shown in Figure 2d. As explained earlier, before packing the other predecessors of additions (I3 and I9), we traverse the tree up for the predecessors of I1 and I7. We discover that the loads I0 and I6 are independent and consecutive, thus, they are packed next. Also, the new vector load instruction is speculative since I6 was speculative, Figure 2e. Finally, Figure 2f shows the second inputs of additions (I3 and I9): the two load instructions (I2 and I8) are also vectorized. Pack and Unpack instructions generated to read and write architectural registers in this example can be moved outside the loop as loop invariant code, as shown in Figure 1c. This way, we are able to vectorize the whole loop.

V. Speculation and Recovery

Memory speculation is a key optimization to achieve performance in HW/SW co-designed systems. For example, Transmeta Crusoe [10] reports that, on average, suppressing memory reordering causes 10% and 33% performance loss in operating system boots and user applications respectively. Since, memory operations play an important role in vectorization, by freely reordering them, consecutive memory references can be packed together. This not only helps in utilizing memory bandwidth but also in vectorization of their dependent arithmetic operations. Furthermore, it is important to note that HW/SW co-designed processors like Transmeta Crusoe, BOA etc provide hardware support for speculation and recovery even though they do not have any dynamic vectorization scheme. Therefore, we assume this hardware support to be present in our baseline architecture. Hence, from the vectorization point of view, we do not need to add any new hardware support for speculation and recovery. This section briefly explains how the speculation and recovery mechanism works in HW/SW co-designed processors.

A combination of software and hardware mechanisms is used to detect speculation failure and subsequent recovery. The software labels each load/store instruction with a sequence number in the original program order. If a pair of load-store or store-store instructions, that may alias, is reordered, the original load/store instructions are converted to “speculative load/store” instructions.

The hardware has two sets of architectural registers: a working set and a shadow copy. Before starting the execution of speculated code, a copy of the working set is saved into the shadow registers (Saving a checkpoint). During the execution, only the working copy of the registers is updated. In the case of speculation failure, the register state is restored by copying the contents of shadow registers to the working copy. Restoring the memory state is a little more complicated since it is not practical to have two copies of the whole memory state. To track the changes in the memory state, a store buffer is used. During the normal execution, store instructions write to the store buffer instead of directly writing to the memory. In the case of speculation failure, the contents of the store buffer are discarded whereas they are forwarded to the memory if the speculated code executes successfully.

To detect a speculation failure, the hardware maintains a table to record address and size of all the memory locations accessed by “speculative load/store” instructions in the current superblock. Moreover, the sequence number of “speculative load/store” instructions is also recorded in the table. During the execution, if the hardware detects:

- that a speculative memory instruction with higher sequence number is executed before another speculative memory instruction with lower sequence number and
- they access overlapping memory locations,

an exception in raised. In this case, the contents of the store buffer are flushed; register values from the shadow registers are copied to the working set; (This has the effect of restoring the earlier saved checkpoint) and the execution is restarted in Interpretation Mode. On the other hand, in case of successful execution of speculated code, values in the store buffer are forwarded to the memory and the contents of the shadow registers are discarded.

```
Seq Num                          Seq Num
   1    ld_64   v1, M[x]            2   st_64_s   v2, M[y]
   2    st_64   v2, M[y]            1   ld_64_s   v1, M[x]
```

a) Orignal Code Sequenc) *b) Reordered Code Sequence*

```
PC -->  1   ld_64_s    v1, M[x]
```

Seq Num	Address	Size
2	y	8

c) Hardware Table State

Figure 3. Speculation Failure Detection Example.

Figure 3 shows an example of speculation failure detection mechanism. Figure 3a shows the original code sequence with two memory references where the relation between the two memory addresses is unknown. The two instructions are labeled in the program order. Figure 3b shows the reordered code sequence. The instructions maintain their sequence number. However, they are converted to speculated instructions to inform the hardware to check them for speculation failure. Figure 3c shows the hardware table state just before executing the speculated load instruction. The program counter points to the current instructions and the table has entry for the executed speculated store instruction. At this point, since the instruction with higher sequence number(2) has been executed before the instruction with smaller sequence number(1), if the address of the current speculated load instruction overlaps with the address of the speculated store instruction, the hardware will generate an exception and will go the recovery mode.

If the rate of speculation failures exceeds a predetermined limit in a particular superblock, it is recreated without reordering ambiguous memory references.

With this speculation and recovery support available in the baseline architecture, speculatively vectorized code can be executed correctly without any additional hardware support.

VI. Performance Evaluation

A. Experimental Framework

To evaluate the proposals, we use DARCO [21], which is an infrastructure for evaluating HW/SW co-designed virtual machines. DARCO executes guest x86 binary on a PowerPC-like RISC host architecture. Since DARCO emulates floating point code in software, we extended the infrastructure to add floating point scalar and vector operations. The proposed algorithm was implemented in TOL to support vectorization.

In our experiments, we assume that the host architecture supports a vector width of 128-bits. Moreover, we consider

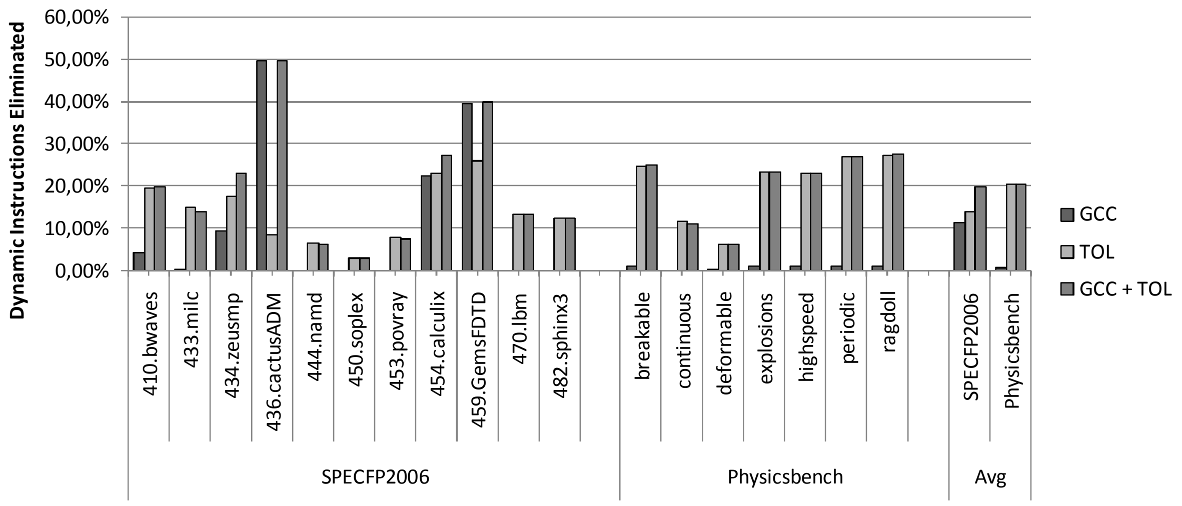

Figure 4 Percentage of Dynamic Instructions Eliminated by GCC, TOL and GCC + TOL Vectorizations

only floating point operations for vectorization (because most SIMD optimizations tend to focus on them) and no integer operation is vectorized. For this reason, we show only floating point instructions in the results presented in this section.

For the speculation and recovery, as discussed in Section V, the hardware maintains a table where it stores the sequence number, direction and size of speculative load/store instructions. We implement this table with 1K entries. Optimal duration/position to take a checkpoint is a different research problem and is out of scope of this paper. For simplicity we take checkpoint in the beginning of every superblock. We implement the store buffer with 1K entries. Moreover to avoid overflow of the store buffer we restrict the number of load/store instructions to be 1K in a superblock. Since we take checkpoint in the beginning of every superblock and a superblock cannot have more than 1K load/store, the store buffer can never overflow.

B. Benchmarks

To measure the success of the proposals we use a set applications from SPECFP2006[1] and Physicsbench[27] benchmarks suites. Furthermore, to measure the effectiveness of the proposed algorithm in vectorizing pointer based applications we use kernels from UTDSP benchmark suite [3]. UTDSP benchmark suite contains array and pointer based version of several signal processing kernels. Both versions provide identical functionality, the only difference being the use of arrays or pointers to traverse the data structures. SPECFP2006 benchmarks operate on double-precision, whereas Physicsbench and UTDSP operate on single-precision floating point values.

All the benchmarks are executed till completion. SPECFP2006 benchmarks are executed using “train” input. Moreover, choose only the benchmarks which have less than 150 billion dynamic instructions to keep the execution time manageable. The benchmarks are compiled with gcc version 4.5.3, optimization flags “-O3 –ffast -math -fomit-frame-pointer” and “-mfpmath=sse -msse3” vectorization flag. To disable vectorization “-fno-tree-vectorize” flag is used.

C. FP Dynamic Instruction Elimination

This section presents the percentage of dynamic instructions eliminated by 1) only GCC, 2) only TOL and 3) GCC+TOL vectorizations, first for SPECFP2006 and Physicsbench benchmarks suites and then for UTDSP Kernels. GCC and TOL represent static and dynamic vectorization respectively. For TOL vectorization the input binary is compiled by GCC but not vectorized. Also, TOL vectorization results show its effectiveness in vectorizing legacy code, since input binary is not vectorized for any SIMD accelerator. For GCC+TOL case, the input binary to TOL is already vectorized by GCC. The results of this case show the vectorization opportunities missed by GCC but captured by TOL.

1) Benchmarks: For SPECFP2006, on average, the combined GCC+TOL approach eliminates approximately twice the number of instructions than only the static GCC vectorization as shows in Figure 4. GCC+TOL vectorization outperforms GCC for all the SPECFP2006 benchmarks except for 436.cactusADM and 459.GemsFDTD. GCC completely vectorizes these benchmarks and hence TOL does not get any further vectorization opportunities. Therefore, instruction elimination is same for GCC and GCC+TOL. It is also important to note that on average, dynamic TOL vectorization itself outperforms static GCC vectorization. Moreover, the only benchmarks where GCC outperforms TOL are again 436.cactusADM and 459.GemsFDTD. The effectiveness of TOL vectorization, to some extent, depends on the quality of the input binary. For example, for 436.cactusADM the input binary to TOL contains GCC unrolled version of the hottest loop. This GCC unrolled loop does not fit in a single superblock due to TOL´s restriction on the maximum number of instructions in a superblock. Therefore, TOL vectorizer could not vectorize it as good as GCC. For 459.GemsFDTD GCC generates significant spill-fill code (to store/retrieve temporary values to/from memory) in the frequently executed loops. This spill-fill code affects TOL´s ability to vectorize this benchmark.

GCC could not vectorize Physicsbench mainly due to the presence of complex control flow in the most frequently executed loops. TOL also is unable to unroll these loops; however, it extracts significant vectorization opportunities through superblock vectorization. Since GCC fails to vectorize anything, GCC+TOL and TOL vectorizations both eliminate 20% of dynamic instruction stream.

2) *Kernels:* Table I shows the vectorization results for UTDSP kernels. As the table shows GCC vectorizes the array based version of FFT, LATNRM and Matrix Multiplication (MULT) but for the pointer based version it is able to vectorize only LATNRM. On the contrary, performance of TOL is same for the array and pointer based versions for all the kernels except for IIR. Pointer based version of IIR contains control flow inside the innermost loop and hence TOL fails to vectorize it. Furthermore, once again a combination of static and dynamic vectorization, GCC+TOL, provides the best solution.

TABLE I. PERCENTAGE OF DYNAMIC INSTRUCTIONS ELIMINATED BY GCC, TOL AND GCC+TOL VECTORIZATIONS

Benchmark	Type	GCC	TOL	GCC + TOL
FFT	Array	43.28%	52.70%	43.28%
	Pointer	0.00%	49.87%	49.87%
FIR	Array	0.00%	0.00%	0.00%
	Pointer	-0.08%	0.00%	-0.08%
IIR	Array	0.00%	32.52%	32.52%
	Pointer	0.00%	0.00%	0.00%
LATNRM	Array	23.48%	7.38%	20.44%
	Pointer	19.43%	17.85%	27.76%
LMSFIR	Array	0.00%	0.00%	0.00%
	Pointer	0.00%	0.00%	0.00%
MULT	Array	64.72%	17.62%	64.72%
	Pointer	0.00%	17.62%	17.62%
Avg	Array	21.91%	18.37%	26.83%
	Pointer	3.23%	14.22%	15.86%

For the array based version, TOL vectorizer outperforms GCC in vectorizing IIR. GCC is unable to resolve loop carried dependences, whereas speculative vectorization helps TOL to provide an instruction reduction of 32%. On the other hand, GCC surpasses TOL vectorization for LATNRM and Matrix Multiplication (MULT). In the current version of TOL vectorizer, reductions are not implemented. Both LATNRM and MULT have reductions, which TOL fails to vectorize. Moreover, MULT has non-unit stride memory accesses, since only one dimension of the matrix (either row or column) can be accessed in the unit-stride manner. Compliers apply optimizations like “memory layout change”, “data coping” etc to convert non-unit stride accesses to unit-stride. However, these optimizations are not directly applicable at runtime. This adds to the loss of vectorization opportunities for TOL vectorizer.

None of the vectorization schemes is able to extract benefit for FIR and LMSFIR, mainly because of the presence of control flow inside the innermost loop. Moreover, in these benchmarks, the number of independent instructions in the basic blocks (and even in superblocks) is not enough to enable vectorization. It is also interesting to note that TOL eliminates 53% of instructions from array version of FFT, whereas GCC+TOL eliminate only 43% (as does GCC alone). This is because the input to TOL is completely vectorized by GCC and TOL does not find any vectorization opportunities, therefore the instruction reductions stays at 43% in GCC+TOL case.

D. Vectorization Overhead

Vectorization overhead is the fraction of dynamic instruction stream that corresponds to the vectorization of superblocks by TOL. A high vectorization overhead might offset the benefits of the vectorization. We calculate the vectorization overhead as:

$$= \frac{Total\ overhead_{with\ vectorization} - Total\ overhead_{without\ vectorization}}{Total\ number\ of\ dynamic\ instructions_{without\ vectorization}}$$

Our experimental results show that, on average, the vectorization overhead is less than 0.6% of the dynamic instruction stream, for all the benchmark suites. Hence, the dynamic vectorization overhead is negligible compared to its benefits.

E. Effectiveness of Memory Speculation

One of the main factors in the success of the proposed vectorization scheme is the memory speculation. However, it might backfire if there are lots of speculation failures. A speculation failure results in executing un-optimized (and without TOL vectorization) version of the code and if the rate of speculation failure exceeds a predetermined threshold, recreating the superblock without speculation. However, our results show that, on average, we execute more than 99% of the dynamic code in speculation mode. It reflects the fact that number of speculation failures, and hence the overhead associated with it, is negligible.

F. Performance

For the performance analysis, we model a simple in-order processor, in congruence with the simple hardware design philosophy of the co-designed processors, with issue width of two. Microarchitectural parameters for the modeled processor are given in Table II. For the performance analysis both the floating point and integer instructions are considered, even though TOL vectorizes only the floating point code.

TABLE II. PROCESSOR MICROARCHITECTURAL PARAMETERS

Parameter	Value
L1 I-cache	64KB, 4-way set associative, 64-byte line, 1 cycle hit, LRU
L1 D-cache	64KB, 4-way set associative, 64-byte line, 1 cycle hit, LRU
Unified L2 cache	512KB, 8-way set associative, 64-byte line, 6 cycle hit, LRU
Scalar Functional Units (latency)	2 simple int(1), 2 int mul/div (3/10) 2 simple FP(2), 2 FP mul/div (4/20)
Vector Functional Units (latency)	1 simple int(1), 1 int mul/div (3/10) 1 simple FP(2), 1 FP mul/div (4/20)
Registers	128-Integer, 128-Vector, 32-FP
Main memory Lat	128 Cycles

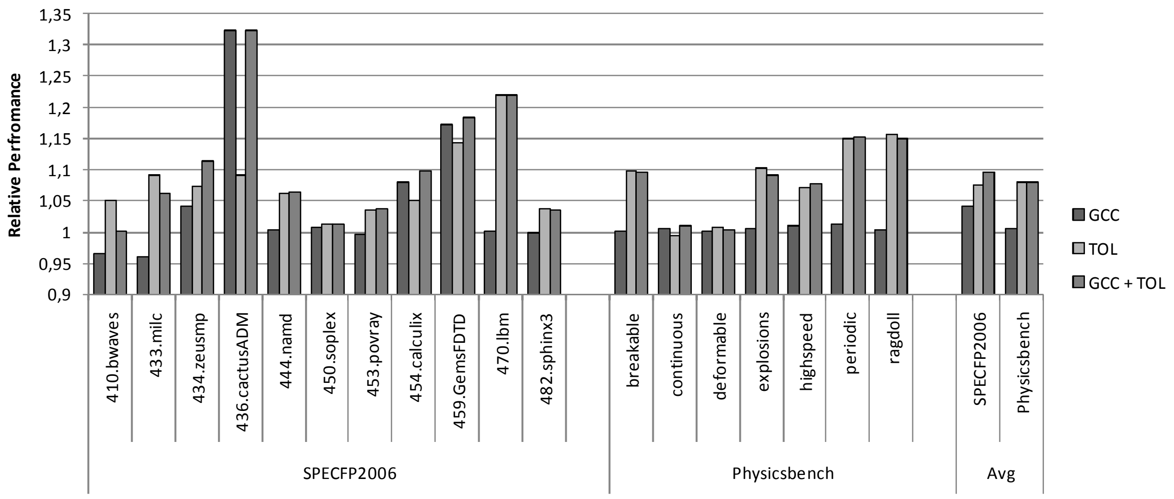

Figure 5 Execution speed for GCC, TOL and GCC + TOL vectorized code relative to unvectorized code. Higher is better.

Figure 5 shows the performance of the vectorized code using the different vectorization schemes relative to the unvectorized code, for SPECFP2006 and Physicsbench. The performance results in the figure conform to the results of Figure 4 for dynamic instruction elimination. For SPECFP2006, GCC+TOL vectorization provides twice the performance benefit than GCC alone (10% compares to 5% of GCC alone). Also, TOL vectorization alone provides better performance than GCC alone. It is interesting to note that for 410.bwaves and 433.milc GCC vectorized code gets a slowdown even though Figure 4 shows dynamic FP instruction elimination. The slowdown comes because of the integer code. GCC adds more integer code than it vectorizes, hence suffers a slowdown. Moreover, for these benchmarks GCC+TOL provides worse performance than TOL alone because GCC+TOL vectorizes GCC vectorized input with extra integer code whereas TOL vectorizes unvectorized code.

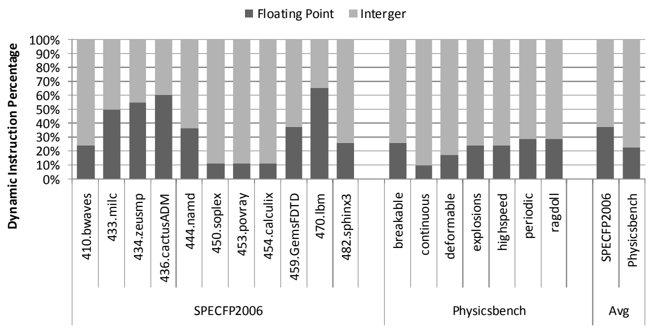

Figure 6 Integer and Floating Point Instruction Distribution in SPECFP2006 and Physicsbench

As GCC fails to vectorize anything in Physicsbench it does not show any performance improvements. However, similar to the results of Figure 4, GCC+TOL and TOL vectorizations provide similar performance benefits for Physicsbench.

An interesting thing to note is that in Figure 4 GCC+TOL vectorization, on average, eliminates 20% of the dynamic instruction stream for both SPECFP2006 and Physicsbench. However, SPECFP2006 gets more speed up than Physicsbench as shown in Figure 5. This is because percentage of floating point code is more in SPECFP2006 than in Physicsbench as shown in Figure 6.

Table III shows the speedup for UTDSP kernels. These results also conform to the results of Table I. For the pointer based version of the kernels GCC loses significant performance compared to the array based version. However, performance is not affected a lot for TOL vectorizer. Furthermore, the combination of static and dynamic vectorizations, GCC+TOL, is able to extract maximum performance out of the kernels.

TABLE III. EXECUTION SPEEDUP RELATIVE TO UNVECTORIZED CODE. HIGHER THE BETTER.

Benchmark	Type	GCC	TOL	GCC + TOL
FFT	Array	1.26	1.50	1.26
	Pointer	1.00	1.50	1.50
FIR	Array	1.00	1.00	1.00
	Pointer	1.05	1.00	1.05
IIR	Array	1.00	1.29	1.29
	Pointer	1.00	1.00	1.00
LATNRM	Array	1.39	1.03	1.33
	Pointer	1.31	1.13	1.39
LMSFIR	Array	1.00	1.00	1.00
	Pointer	1.03	1.00	1.03
MULT	Array	2.33	1.07	2.33
	Pointer	1.17	1.23	1.16
Avg	Array	1.33	1.15	1.37
	Pointer	1.09	1.14	1.19

VII. RELATED WORK

Speculative Dynamic Vectorization is not a much extended topic in literature. There have only been a few proposals like Speculative Dynamic Vectorization [20] and Dynamic Vectorization in Trace Processors [26]. None of them is in the context of HW/SW co-designed processors.

A. Pajuelo et al. [20] proposed to speculatively vectorize the instruction stream in the hardware for superscalar architectures. Several hardware structures are added to support speculative dynamic vectorization, which might not be a power efficient solution, especially in out-of-order superscalar processors where power consumption is already a big issue. They report, more than half of the speculative work is unless due to mispredictions, whereas the rate of speculation failure is negligible in our case. S. Vajapeyam et al. [26] builds a large logical instruction window and converts repetitive dynamic instructions from different iterations of a loop into vector form. The whole loop is vectorized if all iterations of the loop have the same control flow.

Liquid SIMD [8] decouples the SIMD accelerator implementation from the instruction set of the processor by compiler support and a hardware based dynamic translator. Similarly, Vapor SIMD [19] provides a just-in-time compilation solution for targeting different SIMD architectures. Thus, both the solutions eliminate the problem of binary compatibility and software migration. However, both need compiler changes and recompilation. J. Shin et al. [24] extended the SLP in the presence of control flow. They execute both paths of an "if statement" as a vector instruction and then choose the correct result. Our solution, however, requires the execution of only the frequently executed path.

VIII. Conclusion

This paper proposed to assist the static compiler vectorization with a complementary dynamic vectorization. Static vectorization applies complex and time consuming loop transformations at compile time to vectorize a loop. Subsequently at runtime, dynamic vectorization extracts vectorization opportunities missed by static vectorizer due to conservative memory disambiguation analysis and limited vectorization scope. Furthermore, the paper proposed a vectorization algorithm that speculatively reorders ambiguous memory references to facilitate vectorization. The hardware, using the existing speculation and recovery support, checks for any memory dependence violation and takes corrective action in that case.

Our experimental results show that the combined static and dynamic vectorization improves the performance twice compared to static vectorization alone for SPECFP2006. Furthermore, we show that the proposed dynamic vectorization performs as good for pointer based applications as for the array based ones. However, GCC vectorization loses significant opportunities when source code utilizes pointers. Moreover, the overhead of runtime vectorization is only 0.6%.

Acknowldgments

This work is partially supported by the Generalitat de Catalunya under grant 2009SGR-1250, the Spanish Ministry of Education and Science under grant TIN 2010-18368, and Intel Corporation. Rakesh Kumar is supported by an FPI-UPC research grant.

References

[1] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. URL http://www.spec.org/cpu2006/.

[2] Intel Corporation, Intel$^{®}$ 64 and IA-32 Architectures Software Developer´s Manual, Volume 1-3.

[3] UTDSP Benchmarks: www.eecg.toronto.edu/~corinna/

[4] "The Intel® Xeon Phi™ Coprocessor", : http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html

[5] P.D´Arcy and S. Beach, StarCore SC140: A New DSP Architecture for Portable Devices. In *Wireless Symposium*. Motorola, Sept. 1999.

[6] M. Baron. Cortex-A8: High speed, low power. M*icroprocessor Report*,11(14):1–6, 2005.

[7] A. J. C. Bik et al. Automatic intra-register vectorization for the Intel architecture. *International Journal of Paral-lel Programming*, 30(2):65–98, April 2002

[8] N. Clark et al. Liquid SIMD: Abstracting SIMD Hardware using Lightweight Dynamic Mapping. In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07), 216-227

[9] K. Diefendorff, P.K. Dubey, R. Hochsprung, H. Scale. AltiVec extension to PowerPC accelerates media processing, *IEEE Micro,* , vol.20, no.2, pp.85-95, Mar/Apr 2000

[10] J. C. Dehnert et al. The transmeta code morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In Proceedings of CGO '03, pages15-24.

[11] B. Guo et al. Selective Runtime Memory Disambiguation in a Dynamic Binary Translator, In Proceedings of the 15th international conference on Compiler Construction (CC'06), pages 65-79.

[12] J. Holewinski et al. Dynamic trace-based analysis of vectorization potential of applications. In Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '12), pages 371-382.

[13] J. A. Kahle et al. Introduction to the Cell Multiprocessor. In *IBM Journal of Research and Development*, 49(4), pages 589–604, July 2005

[14] S. Larsen et al. Exploiting superword level parallelism with multimedia instruction sets. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00).

[15] Ho-Seop Kim et al. Hardware Support for Control Transfers in Code Caches. In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36). 253-.

[16] R. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51-59, Aug 1996.

[17] S. Maleki et al. An Evaluation of Vectorizing Compilers. In Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11), pages 372-382.

[18] D. Naishlos. Autovectorization in GCC. In *The 2004 GCC Developers' Summit*, pages 105–118,2004.

[19] D. Nuzman et al.Vapor SIMD: Auto-vectorize once, run everywhere. In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11). USA, 151-160.

[20] A. Pajuelo, A. Gonzalez, and M. Valero. 2002. Speculative dynamic vectorization. In Proceedings of the 29th annual international symposium on Computer architecture (ISCA '02), pages 271-280.

[21] D. Pavlou et al. DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines. In Proceedings of the 4th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT'11), held in conjuction with ISCA-38.

[22] G. Ren, P. Wu, D. Padua. An Empirical Study On the Vectorization of Multimedia Applications for Multimedia Extensions. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) Volume 01 (IPDPS '05), Vol. 1, 04-08 2005

[23] S. S. Paul et al. BOA: Targeting multi-gigahertz with binary translation. In Proc. of the 1999 Workshop on Binary Translation, IEEE Computer Society Technical Committee on Computer Architecture Newsletter.

[24] J. Shin et al. Superword-Level Parallelism in the Presence of Control Flow. In Proceedings of the international symposium on Code generation and optimization (CGO '05). 165-175.

[25] M. Sporny, G. Carper, and J. Turner. The Playstation 2 Linux Kit Handbook, 2002.

[26] S. Vajapeyam et al. Dynamic vectorization: a mechanism for exploiting far-flung ILP in ordinary programs. In Proceedings of the 26th annual International Symposium on Computer architecture (ISCA '99)16-27.

[27] T. Y. Yeh et al. Parallax: An architecture for real-time physics. In Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07), pages 232-243.