Speculative Dynamic Vectorization for HW/SW Co-designed Processors

Rakesh Kumar[1], Alejandro Martínez[2], Antonio González[1, 2]
[1] Dept. 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

1. INTRODUCTION

Hardware/Software (HW/SW) co-designed processors have emerged as a promising solution to the power and complexity problems of modern microprocessors. These processors keep their hardware simple in order not to hit the power wall and utilize dynamic optimizations to improve the performance. However, vectorization, one of the most potent optimizations, has not yet received deserved attention. Compiler´s inability to reorder ambiguous memory references severely limits vectorization opportunities, especially in pointer rich applications.

This paper presents 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. The algorithm does not require any compiler or operating system support. Moreover, it can vectorize legacy code which was not compiled for any SIMD accelerator.

2. ALGORITHM

The software layer of our co-designed processor is called Emulation Software Layer (ESL). ESL 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.

4.1 Pre-Vectorization Steps

4.1.1 Superblock Creation

ESL starts by interpreting guest x86 instruction stream in IM. When a basic block is executed more than a predetermined number of times, ESL 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 the branches is also collected. Once the execution of a basic block exceeds another predetermined threshold, ESL 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 or unrolled loops.

4.1.2 Pre-optimizations

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.

4.2 Vectorization

The vectorizer packs together a number of independent scalar instructions, which perform the same operation, and replaces them with one vector instruction. 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 be performing the same operation.
- must be independent.
- must not have been included 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. 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.

Vectorization starting from adjacent loads/stores has an obvious limitation: if a superblock does not have any consecutive loads/stores, nothing can be vectorized. To tackle this problem, after packing all loads/stores and their predecessors/successors, we check if still there are some arithmetic instructions which can be packed together. If yes, we vectorize them and follow their predecessor/successor trees. This allows us to partially vectorize loops with interleaved memory accesses.

While traversing the predecessor/successor tree, if we find out that the predecessors of a pack cannot be vectorized, a Pack instruction is generated. This Pack instruction collects the results

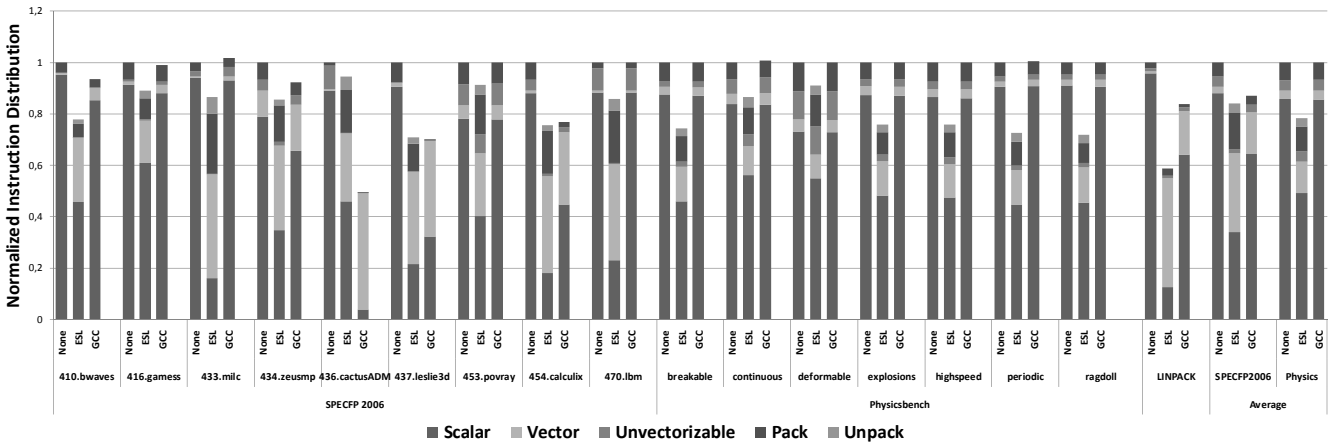

Figure 1 Normalized Dynamic Instruction Distribution: Without Vectorization, ESL Vectorization and GCC Vectorization

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.

3. SPECULATION AND RECOVERY

Memory speculation is a key optimization to achieve performance in HW/SW co-designed systems. For example, Transmeta Crusoe [2] 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.

ESL labels each load/store instruction with a sequence number in original program order. During vectorization or instruction scheduling, 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.

To support the speculative execution, we have two sets of architectural registers in the hardware: the working set and the shadow registers. Before starting the execution of speculative code, a copy of the working set is saved into the shadow registers. Moreover, during the execution of speculative code, store instructions write into a store buffer instead of directly writing to the memory.

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; and the execution is restarted in Interpretation Mode. On the other hand, in case of successful execution of speculative code, values in the store buffer are forwarded to the memory and the contents of the shadow registers are discarded.

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

4. PERFORMANCE EVALUATION

To evaluate the proposed algorithm, we use DARCO [3], which is an infrastructure for evaluating co-designed virtual machines. DARCO executes guest x86 binary on a PowerPC-like RISC host architecture. The proposed algorithm was implemented in Emulation Software Layer (ESL) of DARCO. In our experiments, we assume that the host architecture supports a vector size of 128-bits. Moreover, we vectorize only the floating point operations.

We use a set of applications from SPECFP2006 and Physicsbench benchmarks suites and LINPACK for the evaluation. All the benchmarks are compiled with gcc-4.5.3 with -O3 -fomit-frame-pointer -ffast-math -mfpmath=sse -msse3 flags.

4.1 FP Dynamic Instruction Elimination

Figure 1 compares the dynamic instruction reduction by our algorithm and that of GCC [1]. Dynamic instruction stream includes: Scalar and vector floating point instructions, unvectorizable instructions and Pack/Unpack instructions.

As the figure shows, we are able to reduce, on average, 16%, 22% and 41% of dynamic instructions and compared to GCC, we are able to eliminate, on average, 3%, 22% and 25% more dynamic instructions for SPECFP2006, Physicsbench and LINPACK respectively. The behavior of Physicsbench is really interesting for both vectorization techniques. GCC practically does not find any SIMD parallelism in Physicsbench because of the extensive use of pointers in this benchmark suite. Moreover, these benchmarks consist of complex control flow in the frequently executed loops and GCC fails to vectorize them. However, our algorithm shows a dynamic instruction reduction of 22%. This shows that we are able to find significant vectorization opportunities even in the cases where GCC fails to vectorize anything.

Our experimental results also show that memory speculation doubles the dynamic instruction stream coverage of vectorization.

5 REFRENCES

[1] Auto-vectorization in GCC. URL http://gcc.gnu.org/projects/tree-ssa/vectorization.html

[2] 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-01, pages 15–24, 2003.

[3] D. Pavlou et al. DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines. In *AMAS-BT'11, held in conjunction with ISCA-38*, June 2011.