

國立臺灣大學 電機資訊學院 資訊網路與多媒體研究所  
碩士論文



Graduate Institute of Networking and Multimedia  
College of Electrical Engineering and Computer Science  
National Taiwan University  
Master Thesis

基於進階 HQEMU 之動態二進制碼向量化

Dynamic Binary Vectorization in Enhanced HQEMU

林致民

Chih-Min Lin

指導教授：徐慰中 教授

Advisor: Wei-Chung Hsu, Ph.D.

中華民國一百零八年六月

June, 2019

國立臺灣大學碩士學位論文  
口試委員會審定書

基於進階 HQEMU 之動態二進制碼向量化

Dynamic Binary Vectorization in Enhanced HQEMU

本論文係林致民君（學號 R06944005）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國一百零八年六月十七日承下列考試委員審查通過及口試及格，特此證明

口試委員：

徐皓中

(簽名)

(指導教授)

洪鼎詠

吳直頁

徐皓中

廖世偉

黃敬群

楊佳玲

所長：



## 基於進階 HQEMU 之動態二進制碼向量化

### 摘要

在平行處理中，自動向量化技術已被編譯器用來善用資料層級的平行化。然而，因為處理器架構一直在提升向量處理 (Vector) 或單指令多資料流 (SIMD) 的能力，所以舊的應用程式沒辦法善用新的 Vector/SIMD 的能力。例如舊的 ARMv7 二進制執行檔不能從 ARMv8 的雙精度浮點運算的 SIMD 得到好處，舊的 x86 二進制執行檔沒辦法享受 AVX-512 的新功能。

在這篇論文中，我們探討在跨指令集架構動態二進制碼轉譯器的基礎問題，該如何將非量化的迴圈轉換成 Vector/SIMD 的形式，使得應用程式在新的處理器上可以獲得更高的計算輸出量。核心概念是從這些應用程式的二進制檔還原重要的迴圈資訊，使得迴圈可以被自動向量化。實驗結果顯示，對於不同的 benchmark，我們的方法可以在 ARMv7 到 ARMv8 的動態二進制碼轉譯中相較於 ARMv7 Native 獲得 1.42 倍的效能提升。

**關鍵字：** 動態二進制碼轉譯, 虛擬暫存器推廣化, 單指令多資料流, 向量處理, 自動向量化



# Dynamic Binary Vectorization in Enhanced HQEMU

## Abstract

Auto vectorization techniques have been adopted by compilers to exploit data-level parallelism in parallel processing for decades. However, since processor architectures have kept enhancing with new features to improve vector/SIMD performance, legacy application binaries failed to fully exploit new vector/SIMD capabilities in modern architectures. For example, legacy ARMv7 binaries cannot benefit from ARMv8 SIMD double precision capability, and legacy x86 binaries cannot enjoy the power of AVX-512 extensions.

In this thesis, we study the fundamental issues involved in cross-ISA Dynamic Binary Translation (DBT) to convert non-vectorized loops to vector/SIMD forms to achieve greater computation throughput available in newer processor architectures. The key idea is to recover critical loop information from those application binaries in order to carry out vectorization at runtime. Experiment results show that our approach achieves an average speedup of 1.42x compared to ARMv7 native run across various benchmarks in an ARMv7-to-ARMv8 dynamic binary translation system.

**Keywords:** Dynamic Binary Translation, Virtual Register Promotion, SIMD, Vector, Auto Vectorization



# Contents

	page
摘要	ii
Abstract	iii
Contents	iv
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>5</b>
2.1 Cross-ISA Dynamic Binary Translation .....	5
2.2 Issues of Loop Analysis for Binary .....	6
<b>3 Methodology</b>	<b>9</b>
3.1 Overview .....	9
3.2 Issue of Virtual Register Promotion .....	10
3.3 Algorithm of Virtual Register Promotion .....	13
3.4 Speculative Execution.....	19
<b>4 Performance Evaluation</b>	<b>22</b>
4.1 Overall Performance .....	24
4.2 Evaluation of Virtual Register Promotion.....	25
<b>5 Related Work</b>	<b>29</b>
<b>6 Conclusion and Future Work</b>	<b>32</b>
<b>Reference</b>	<b>34</b>



# List of Figures

	<b>page</b>
2.1 Example of binary translation and vectorization . . . . .	6
3.1 Workflow of optimization approach . . . . .	10
3.2 Example of vectorization with virtual register promotion . . . . .	11
3.3 Runtime Check for Virtual Register Promotion . . . . .	17
3.4 Rewrite Control Flow for Vectorization . . . . .	20
4.1 Performance results of ARMv7-to-ARMv8 Translation . . . . .	24
4.2 Evaluation results of virtual register promotion . . . . .	26



# List of Tables

	<b>page</b>
4.1 Benchmark Kernels from SPEC . . . . .	23
4.2 Number of Loops with Register Spilling in Kernels . . . . .	24



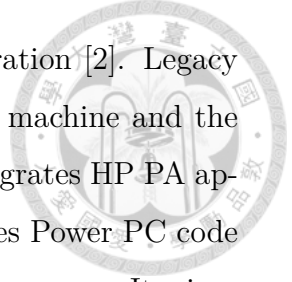
# Chapter 1

## Introduction

Vector processing has been adopted in parallel processing for decades. For vector supercomputers, Cray-1 and Cyber205 were representative in the late 70's, and followed by Cray X-MP/Y-MP, Cray-2, Fujitsu VP200/400, and NEC SX-2 into the '80s. In the late '90s, in order to support real-time gaming and audio/video processing, HP PA added MAX instructions and SUN included VIS instructions in their processors. Intel x86 also started to support the MMX extension and followed with SSE extensions later to provide greater SIMD computing throughput. The type of supporting multimedia processing with SIMD in modern microprocessors is often referred to as Sub-word Level Parallelism (SLP) [1]. Nowadays, almost all microprocessors provide support for SIMD processing. Many of them are exploiting more advanced SIMD features, such as x86 AVX-512 extensions, ARM SVE (Scalable Vector Extension) and RISC-V Vector Extension.

Newer microprocessors will provide stronger and more powerful SIMD capabilities. At the same time, most popular ISAs support backward compatibility in that legacy binaries could run directly on the same processor family, including newer processors. Unlike the scalar counterpart of an ISA, SIMD enhancements often give a more significant performance boost to the data parallel portion of the application. For example, AVX-512 has the potential to increase SIMD performance by 4X over x86 SSE, where the improvement from the scalar ISA enhancements would be much less significant. When running the legacy application binaries on a newer microprocessor, there is a rich opportunity to increase the effective utilization of SIMD resources and boost performance via Dynamic Binary Translation (DBT).

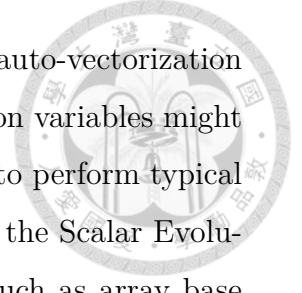




DBT is a common technique used for legacy application migration [2]. Legacy application binary migration is typically applied when the guest machine and the host machine are in different ISAs. For example, HP's Aries [3] migrates HP PA applications to the Itanium architecture, Apple's Rosetta [4] migrates Power PC code to Intel x86 architecture, Intel IA-32/EL migrates x86 binaries to run on Itanium processors, and MAMBO-x64 [5] migrates ARMv7 binaries to ARMv8 processors. However, there are cases where the legacy application binaries can run directly on the host machines, for example, Intel SSE binaries can run directly on AVX-2/AVX-512 based processors, and ARMv7 binaries can run on some ARMv8 based processors with 32-bit support. Under such scenarios, DBT could still be deployed, for the purpose of exploiting new architecture/micro-architecture features, rather than meeting the compatibility requirements. In such a case, this DBT is often referred to as Dynamic Binary Optimizer (DBO) instead of DBT.

To exploit new and more powerful SIMD features in the host machine, a DBO could attempt two obvious directions, one is to exploit wider computational lanes and the other is to vectorize scalar loops in the legacy binary using new SIMD features. For the former case, if an application was compiled for Intel SSE ISA, the binary could only utilize 128 bits of SIMD width. When this binary is running on a new AVX-512 based processor, which supports 512-bit wide computation, the loop could be reorganized to increase benefit from the greater computational throughput. For the latter case, a loop with double precision computation in an application might not get vectorized on ARMv7 due to the lack of double precision NEON support. When this binary is running on an ARMv8 processor, such a scalar loop could possibly be vectorized with the new NEON double precision instructions. These scenarios are becoming more commonplace for the coming future. For instance, some early RISC-V processors do not have vector extension ISA, so many loops could not get vectorized. When newer RISC-V machines equipped with vector units appear on the market, such legacy binaries with the scalar only code could exploit new SIMD capabilities via DBO.

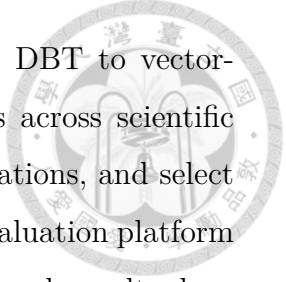
For the two directions of exploiting new SIMD capability via DBO, prior research has reported preliminary results [6], [7]. However, we attempt to address the fundamental issues with the runtime vectorization problems. The first problem is



how to recover loop information from the binary so that prior auto-vectorization compiler techniques could be applied. For example, some induction variables might have been spilled from a register to the stack, making it difficult to perform typical vectorization. Memory accesses instead of registers often confuse the Scalar Evolution Analysis to identify the vectoriable patterns of the loops, such as array base addresses, induction variables, loop boundaries, constant variables, and so on. This is particularly difficult since stripped application binaries lack high-level semantic information and may not have symbol table information. Compile time aliasing analyses are often not applicable here. Although some prior work proposed to apply symbolic promotion to recover such information, yet the proposed approaches are in static binary translation domains. Most legacy binary migrations are based on DBT due to the code discovery and code location issues [2]. The other issue is the possible memory dependence that could prevent a full vectorization of the loop. This is still true even for loops already vectorized in the first place. In order to exploit the increased computational lanes available on the host, the reorganized loop would use a larger strip than the one used in the legacy code. Such a change in strip size might invalidate the data dependence checked at compile time.

We propose virtual register promotion with runtime aliasing checks to ensure effective and safe optimizations. The number one priority for any optimization system is a safe and correct transformation (except for approximate computing where precise computational results are not absolutely required). Our proposed approach ensures safe transformation while exploiting new SIMD capabilities in DBT/DBO. The key contributions of this work are as follows:

- We identify the issues of applying vectorization techniques on binaries in DBT/DBO systems. We address the analysis issues of register spilling when vectorizing loops in legacy binaries and propose virtual register promotion to recover critical loop vectorization information.
- We also identify the required safety issues of virtual register promotion for vectorization in DBT/DBO. To overcome the challenges of memory analyses in DBT/DBO, we propose the new approach to verify the correctness of virtual register promotion with a lightweight aliasing detection to significantly reduce overhead.

- 
- We implemented our approach in a retargetable cross-ISA DBT to vectorize legacy binary code. We choose the benchmark kernels across scientific computing and linear algebra with double precision computations, and select ARMv7 scalar to ARMv8 NEON SIMD translation as our evaluation platform to demonstrate the effectiveness of our approach. The benchmark results show that our approach achieves 1.42X speedup, on average, compared to ARMv7 native runs, on the ARMv8 Cortex-A53 processor.

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of cross-ISA DBTs and explain the conditions of auto vectorization techniques in static compiler and DBTs. In chapter 3, we propose and design an approach of loop transformation in DBT and then report the evaluation results in chapter 4. Chapter 5 describes related work, and chapter 6 concludes with future work.



## Chapter 2

# Background

This chapter briefly reviews Dynamic Binary Translation (DBT) techniques and traditional auto vectorization techniques. We also illustrate the issues of loop analyses for vectorization in DBTs.

## 2.1 Cross-ISA Dynamic Binary Translation

Modern cross-ISA DBTs translate guest binaries in the scope of code fragment to intermediate representation (IR), re-optimize binary code targeting the host machine and maintain guest architectural states at code fragment boundaries to ensure the correct code emulation. Since aggressive optimization in DBTs requires more translation time, part of the runtime, DBT/DBoes typically select a smaller granularity of hot regions to optimize instead of a larger granularity. For instance, Next Executing Tail (NET) [8], adopted in MAMBO-X64 [5] and HQEMU [9], selects a ring (i.e. loops) or a strip of execution path (e.g. a trace) as the optimization candidate.

This work is based on HQEMU, a hybrid QEMU [10] and LLVM [11] Dynamic Binary Translator. A guest binary trace is first translated to LLVM IR. After several phases of optimizations, the trace of IR will go through the LLVM Just-in-Time compiler (MCJIT) to generate host binaries. Figure 2.1 shows the steps of translating the ARMv7 scalar code to ARMv8 double precision NEON SIMD code. Figure 2.1a shows the DAXPY kernel function. This function is compiled to ARMv7 binaries, as shown in Figure 2.1b, and translated via HQEMU to LLVM IRs, shown in Figure 2.1c. Vectorization to LLVM IRs is shown in Figure 2.1d. Finally, the

```

void daxpy(double da,
           double *dx,
           double *dy, int n, ...)
{
    ...
    for (int i = 0; i < n; ++i)
        dy[i] += dx[i] * da;
}

```

(a) C Source Code

```

Loop: vldr d1, [r0] ; dx[i]
      vldr d2, [r1] ; dy[i]
      add r0, r0, #8 ; i=i+1
      vmul d1, d1, d0 ; tmp=dx[i]*da
      vadd d2, d2, d1 ; +dy[i]
      vstr d2, [r1] ; dy[i]=tmp
      add r1, r1, #8 ; i=i+1
      cmp r1, r2 ; i<n
      bne Loop

```

(b) ARMv7 guest binary

```

Entry: load %r0.cpu %r1.cpu %r2 %d0
Loop:
  © %r0 = phi i32 [%r0.cpu], [%r0.a]
  © %r1 = phi i32 [%r1.cpu], [%r1.a]
  Ⓐ %d1 = load double* %r0
  Ⓑ %d2 = load double* %r1
  © %r0.a = add i32 %r0, 8
  Ⓓ %d1.a = fmul double %d1, %d0
  Ⓔ %d2.a = fadd double %d2, %d1.a
  Ⓒ store %d2.a, double* %r1
  © %r1.a = add i32 %r1, 8
  Ⓕ %cmp = cmp ne %r1.a, %r2
  Ⓖ br %cmp, %Loop, %Exit
Exit: store %r0.a %r1.a %d2.a %d1.a

```

(c) Translate ARMv7 binary to IRs

```

Entry: load %r0.cpu %r1.cpu %r2 %d0
Loop:
  © %r0 = phi i32 [%r0.cpu], [%r0.a]
  © %r1 = phi i32 [%r1.cpu], [%r1.a]
  Ⓐ %v1 = load <2 x double>* %r0
  Ⓑ %v2 = load <2 x double>* %r1
  © %r0.a = add i32 %r0, 16
  Ⓓ %v1.a = fmul <2 x double> %v1, %d0
  Ⓔ %v2.a = fadd <2 x double> %v2, %v1.a
  Ⓒ store %v2.a, <2 x double>* %r1
  © %r1.a = add i32 %r1, 16
  Ⓕ %cmp = cmp ne %r1.a, %r2
  Ⓖ br %cmp, %Loop, %Exit
Exit:
  %d1 = extract <2 x double> %v1.a
  %d2 = extract <2 x double> %v2.a
  store %r0.a %r1.a %d1 %d2

```

(d) Vectorized IRs

Figure 2.1: Example of binary translation and vectorization

LLVM IRs are turned into ARMv8 NEON binary with the LLVM-JIT compiler. Our binary vectorization work is between stage 2.1c and 2.1d. In Figure 2.1d, Entry basic block (BB) loads the architectural states from registers (i.e., via register mapping) or memory into virtual registers (i.e., load r0 into %r0.cpu). Here, Loop BB implies the loop body translated from Figure 2.1b, and Exit BB keeps architectural states in sync (i.e., store %r0.a to r0, %r1.a to r1) after executing the loop.

## 2.2 Issues of Loop Analysis for Binary

Scalar Evolution Analysis has been adopted in modern compilers (e.g., GCC, LLVM) to understand loop-oriented expressions and the changes in the value of scalar variables over iterations of the loop. Such analysis approach is commonly used in loop strength reduction, induction variable simplification, loop vectorization, loop access analysis, and dependence analysis, evaluating the evolved values of scalars in a canonical loop. For loop vectorization, illustrated in Figure 2.1b and 2.1c, Scalar Evolution Analysis assists in exposing data dependence (Ⓐ, Ⓑ, and Ⓔ) induction variables and stride distance (Ⓒ and Ⓓ), and loop trip-count analysis (Ⓕ) to ensure

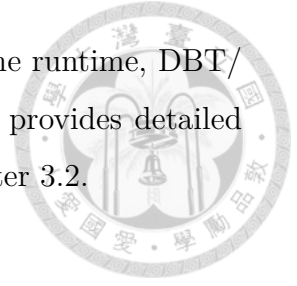
the legality after fusing iterations with vector operations.

Nevertheless, Scalar Evolution Analysis has a critical issue in DBTs. Scalar variables in binaries may not always be in the form of scalars, but in the form of memory references because of register spilling. Register spilling is common in real-world applications containing loop bodies with a large number of variables and longer use-define chains such as Mgrid in SPEC2000 [12], Leslie3d in SPEC2006 [13], and Roms in SPEC2017 [14]. Such spilled variables would make Scalar Evolution Analysis difficult since they would confuse scalars with loop invariants (i.e., memory access to stack) and cripple vectorization capability in DBTs.

Figure 3.2 and 3.2a show a simple case for register spilling in ARMv7 scalar binary of daxpy kernel. ① loads induction variables from stack to register, ③ adds the value for induction variable, and ⑥ stores the variable back to the stack. After translating to IRs, however, ⑥⑦ would be viewed as loop invariant with memory references to stack instead of an induction variable. For another example, ⑦ loads the loop boundary from the stack into register r0, and ⑩ compares loop boundary r0 with induction variables r1. In the form of IRs, ⑦ cause troubles in determining loop trip-count (⑩) in Scalar Evolution Analysis for a similar reason. Furthermore, it is also possible to spill the arbitrary variables from register to memory such as array base address and constant stride in binaries.

Hence, we adopt virtual register promotion, which promotes spilled variables to virtual registers, to simplify loop analysis and enable vectorizations. The loop body in Figure 3.2b would be simplified to the code sequence in Figure 3.2c, and such code avoids confusing memory references and enables DBTs to conduct loop vectorization (Figure 3.2d). For instance, as shown in Figure 3.2c and 3.2d, promoting spilled variables ① and ⑦ to virtual registers %r0 and %r0.b in VRP (i.e., **V**irtual **R**egister **P**romotion) basic block before Loop (i.e., loop body) BB, store the virtual registers back to stack in STVR (i.e., **S**Tore **V**irtual **R**egister) BB, and then transform the loop from scalar to vector form. However, promoting the spilled variables to virtual registers has potential aliasing issues. Spilled variables ⑥⑦ may alias to memory reference instructions (①, ②, and ⑧). Aliasing analysis in DBT is very restricted in that (1) The scope is limited to a trace, not a function or the whole program as in a static compiler, (2) Application binaries are often stripped, no symbol table informa-

tion available, (3) For DBT/DBO, analysis time is also part of the runtime, DBT/DBO cannot afford comprehensive aliasing analysis. This thesis provides detailed solutions to our safe virtual register promotion approach in chapter 3.2.





## Chapter 3

# Methodology

In this chapter, we present our approaches to assist DBTs in vectorization. First, we address the challenges of virtual register promotion, provide the detailed solution of the problems, and prove the validity of this approach after vectorizing the loops. Second, we propose a new approach of loop rewriting to increase the vectorization coverage of binary loop vectorization. Third, we present a new method of runtime checks to ensure the correctness of code emulation after applying above optimizations and vectorization in DBTs. In the following sections, we choose the simplified case of ARMv7 double precision floating point scalar loop to ARMv8 NEON (i.e., 128-bit SIMD register) translation and adopt loop-based vectorization with contiguous memory accesses to illustrate our optimization approach.

### 3.1 Overview

Figure 3.1 illustrates the workflow of our rewriting approach, which targets a translated code fragment of the innermost loop containing neither divergent code paths nor indirect memory accesses in the loop body (step 1). Our translator detects whether the loop contains spilled variables (step 2). If the loop has register spilling, the translator conducts virtual register promotion to assist loop analysis for vectorization (step 3 and step 4). Next, our translator has to determine which variables need to check against aliasing (step 5) if the loop is able to be vectorized. Then, the translator creates two versions of the loop into a scalar-loop and a vector-loop, and then insert runtime checking codes before entering the vector-loop (step 6). Finally, the translator applies vectorization (step 7), rewrites the control flow, and



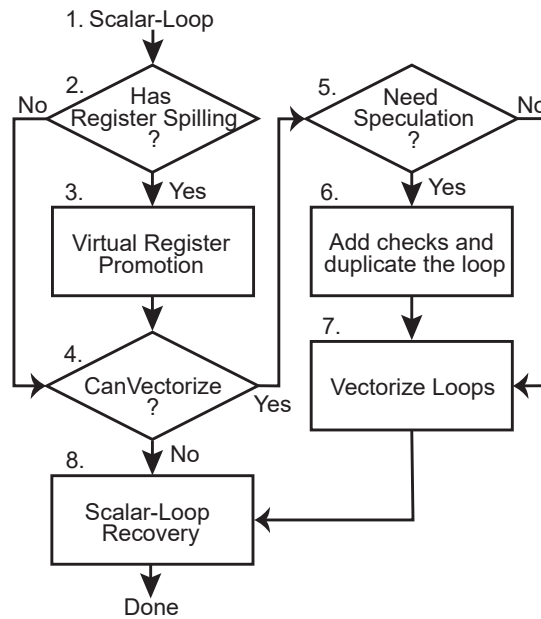


Figure 3.1: Workflow of optimization approach

recovers the scalar-loop back to non-promoted form (step 8) to ensure the validity of code emulation. The following sections would provide more details for each step to rewrite the loop.

## 3.2 Issue of Virtual Register Promotion

This section focus on the challenges and solutions of safe virtual register promotion. There are two types of spilled variables. One is Program Counter (PC) relative data, and the other one is stack variable. The former is a simple case of spilled variables since such type of variables is usually read-only for constant data and embedded in the code section, and DBTs are capable of handling the modified constant variables by re-translating guest binaries. The latter, however, is more challenging for virtual register promotion since such variables can be modified by arbitrary memory reference instructions. For example, as illustrated in Figure 3.2b and 3.2c, ⑥⑦ are stack variables promoted to virtual registers %r0, %r0.b, and %r0.c, respectively. Such spilled variables may be aliased to memory locations referenced by instructions ①②⑧ because it is possible for the store instruction ⑧ to modify the spilled variables ⑥⑦ in memory address %sp (i.e., stack pointer register) and %sp+4 such that the data may not be consistent between virtual registers



```

Loop: ldr r0, [sp] ; load i from stack ①
      vldr d1, [r0] ; dx[i] ②
      vldr d2, [r1] ; dy[i] ③
      add r0, r0, #8 ; i=i+1 ④
      vmul d1, d1, d0 ; tmp=dx[i]*da ⑤
      vadd d2, d2, d1 ; +dy[i] ⑥
      str r0, [sp] ; store i to stack ⑦
      ldr r0, [sp, #4] ; load n from stack ⑧
      vstr d2, [r1] ; dy[i]=tmp ⑨
      add r1, r1, #8 ; i=i+1 ⑩
      cmp r0, r1 ; i < n ⑪
      bne Loop

```

(a) ARMv7 guest binary with register spilling

```

Entry: load %sp %r1.cpu %d0
Loop: ① %r1 = phi [%r1.cpu], [%r1.a]
      ② %r0 = load i32* %sp
      ③ %d1 = load double* %r0
      ④ %d2 = load double* %r1
      ⑤ %r0.a = add i32 %r0, 8
      ⑥ %d1.a = fmul %d1, %d0
      ⑦ %d2.a = fadd %d1, %d1.a
      ⑧ store %r0.a, i32* %sp
      ⑨ %r0.b = load i32* [%sp+4]
      ⑩ store %d2.a, double* %r1
      ⑪ %r1.a = add i32 %r1, 8
      ⑫ %cmp = cmp ne %r0.b, %r1.a
      br %cmp, %Loop, %Exit
Exit: store %r0.b %r1.a %d2.b %d1.a

```

(b) IRs with register spilling

```

Entry: load %sp %r1.cpu %d0
VRP: ① %r0 = load i32* %sp
      ② %r0.b = load i32* [%sp+4]
Loop: ③ %r0.c = phi [%r0], [r0.a]
      ④ %r1 = phi [%r1.cpu], [%r1.a]
      ⑤ %d1 = load double* %r0.c
      ⑥ %d2 = load double* %r1
      ⑦ %r0.a = add i32 %r0.c, 8
      ⑧ %d1.a = fmul %d1, %d0
      ⑨ %d2.a = fadd %d1, %d1.a
      ⑩ store %d2.a, double* %r1
      ⑪ %r1.a = add i32 %r1, 8
      ⑫ %cmp = cmp ne %r0.b, %r1.a
      br %cmp, %Loop, %STVR
STVR: ⑬ store %r0.a, i32* %sp
Exit: store %r0.b %r1.a %d2.b %d1.a

```

(c) IRs with virtual register promotion

```

Entry: load %sp %r1.cpu %d0
VRP: ① %r0 = load i32* %sp
      ② %r0.b = load i32* [%sp+4]
Loop: ③ %r0.c = phi [%r0], [r0.a]
      ④ %r1 = phi [%r1.cpu], [%r1.a]
      ⑤ %v1 = load <2 x double>* %r0.c
      ⑥ %v2 = load <2 x double>* %r1
      ⑦ %r0.a = add i32 %r0.c, 16
      ⑧ %v1.a = fmul <2 x double> %v1, %d0
      ⑨ %v2.a = fadd <2 x double> %v1, %v1.a
      ⑩ store %v2.a, <2 x double>* %r1
      ⑪ %r1.a = add i32 %r1, 16
      ⑫ %cmp = cmp ne %r0.b, %r1.a
      br %cmp, %Loop, %STVR
STVR: ⑬ store %r0.a, i32* %sp
Exit: %d1 = extract <2 x double>* %v1.a
      %d2 = extract <2 x double>* %v2.a
      store %r0.b %r1.a %d1 %d2

```

(d) IRs with promotion and vectorization

Figure 3.2: Example of vectorization with virtual register promotion

and memory. As another example, the spilled variable ⑥ in address  $\%sp$  may be aliased to memory locations referenced by the loads ①②. In this case, when ⑥ updates the value for virtual register  $\%r0.c$  with PHI node, the data in memory  $\%r0.c$  or  $\%r1$  would be inconsistent.

To solve the problems, DBTs should ensure no aliasing between spilled variables and other memory references. Nevertheless, DBTs cannot determine the aliasing when translating guest binaries since both the address of memory accesses (e.g.,  $r1$  and  $r0$ ) and the value in stack pointer register (i.e.,  $sp$ ) can only be identified at run time. Hence, runtime aliasing check is required to prove the validity of virtual register promotion before entering the loop with promoted spilled variables. A naive approach for aliasing check is to insert the checking code before each potential aliasing instructions. However, such an approach is not feasible after the loop is vectorized. Loop-based vectorization changes the order of instructions across iterations and fuses scalar operations into a vector such that DBTs cannot ensure the legality of virtual register promotion after the loop is vectorized.

For instance, as illustrated in Figure 3.2d, if ①② have no aliasing but the store ⑧ aliases to the memory address  $\%sp$  with spilled variable ⑥, DBT should rollback and recover the states for instructions ① ~ ⑦ because ⑥ is an induction variable leading the loads ①② with memory location  $\%r0.c$  and  $\%r1$ . In this case, DBTs should re-execute the instructions in scalar-loop and reload the data from memory to recover the states. Nevertheless, a loop does not always contain only one store instruction. If a valid store instruction is successfully executed before the store ⑧ but the spilled variable ⑥ still alias to ⑧, DBTs cannot recover the original states because incorrect results have already been written back to memory.

One possible solution is recording all the changes in both registers and memory before storing the data back to memory. However, recording the data changes in memory usually needs hardware to assist DBTs in state recovery (e.g., Transmeta CMS [15]), and most modern processors do not provide such supports. Another possible solution is moving all checking codes from memory reference instructions to the head of the loop body and check all possible aliasing pairs between virtual registers and memory references before executing any instruction in each iteration. Nevertheless, such a solution would cause higher overhead when detecting the aliasing in

each iteration and impede the performance improvement. Consequently, overhead reduction of runtime aliasing check is crucial for virtual register promotion.



### 3.3 Algorithm of Virtual Register Promotion

A spilled variable may be viewed as a loop invariant stored on the stack. The stack location is usually a constant address, which is calculated as stack pointer plus offset (i.e., sp of ⑥⑦ in Figure 3.2). DBTs are capable of detecting whether the value of stack pointer is modified in the loop. Therefore, it is possible to validate the correctness of virtual register promotion after vectorization. Runtime aliasing check ensures the virtual register promotion performed is legal. The checking conditions are given below:

$addr(m)$  : address of memory reference  $m$

$size(m)$  : access type of memory reference  $m$

$ss_i$  : store  $i$ th of spilled variable  $sl_i$  : load  $i$ th of spilled variable

$st_i$  : store  $i$ th of non-spilled variable  $tc_i$  :  $i$ th of loop trip-count

$ld_i$  : load  $i$ th of non-spilled variable  $d_i$  :  $i$ th of stride distance

$nss$  : number of  $ss_i$   $nsl$  : number of  $sl_i$   $nl$  : number of  $ld_i$

$ns$  : number of  $st_i$   $SS$  : set of  $ss_i$  address  $SL$  : set of  $sl_i$  address

$ST$  : set of  $st_i$  address  $LD$  : set of  $ld_i$  address

$A$  : set of aliasing address

On the field  $(Z^+)$

$$SS := \{a | addr(ss_i) \leq a < addr(ss_i) + size(ss_i), 1 \leq i \leq nss\} \quad (3.1)$$

$$SL := \{a | addr(sl_i) \leq a < addr(sl_i) + size(sl_i), 1 \leq i \leq nsl\} \quad (3.2)$$

$$ST := \{a | addr(st_i) \leq a < addr(st_i) + tc_i \times d_i, 1 \leq i \leq ns\} \quad (3.3)$$

$$LD := \{a | addr(ld_i) \leq a < addr(ld_i) + tc_i \times d_i, 1 \leq i \leq nl\} \quad (3.4)$$

$$A = (SS \cap (ST \cup LD)) \cup (SL \cap ST) \quad (3.5)$$

$$\text{Legality} = \begin{cases} true, & \text{if } A = \phi \\ false, & \text{otherwise} \end{cases} \quad (3.6)$$

Runtime aliasing check ensures no spilled variables alias to regular memory reference (i.e., load/store non-spilled variables) across all iterations. First, eq 3.1 and 3.2 respectively scan the addresses of all spilled variables stored to stack (SS) and loaded from stack (SL) with the specific size of data type (e.g., size of double is 8 bytes, float is 4 bytes, etc). Second, identify the address range of non-spilled variables across all iterations. Because Scalar Evolution Analysis can expose loop trip-count and stride distance, DBTs can employ such information to calculate the loop boundary (i.e.,  $tc_i \times d_i$ ) in step 3.3 and 3.4. Besides, the loop trip-count and stride distance may be different among all memory references (i.e., vector loads/stores v.s. loop invariants). Finally, the checking in eq 3.5 and 3.6 guarantees the address set of stored spilled variables (SS) do not overlap with the address set of both non-spilled loads (LD) and stores (ST), and all spilled loads do not overlap with the non-spilled stores (ST).

The complete description of virtual register promotion is shown in Algorithm 1. The input VRP, Loop, and STVR represent the three marked basic blocks in Figure 3.2. In the initial state, VRP and STVR BB do not contain any instruction. VRP BB is used to preload the spilled variables to virtual registers, and STVR BB stores the data back to the stack. Loop BB is the loop body containing spilled variables. The output RTCheck is used to record which addresses need runtime aliasing check, and one address maps to one instruction accessing it. All BBs (VRP, Loop, Loop) would be modified by virtual register promotion. The intermediate variable *SVLookup* (**S**pilled **V**ariable **L**ookup **T**able) is a table used to record which instruction first accesses the address in stack, and which one access as a key-value pair ( $address \leftarrow (first\ instruction, last\ instruction)$ ). *UpdateOperand* tracks the data changes among the spilled variables and annotates which instruction should be updated.

The following steps show how our DBT promotes spilled variables to virtual registers, and we follow the format of LLVM IR to illustrate the algorithm. The algorithm tracks the data flow of spilled variables at the same position in the stack. *PromoteSpilledVariable(VRP, Loop, STVR)* (lines 20 to 38) is the main function of this optimization. *BuildSpilledVariableTable* (line 21 and line 1) creates the table of spilled variables to determine which operand should be replaced (*UpdateOperand*),

and record which variable needs runtime aliasing check (*RTCheck*). We consider several cases of register spilling to specific stack address: **a)** loads only, **b)** loads first then stores, **c)** stores first then loads, and **d)** stores only. The cases of memory references to spilled variables have multiple combinations, so DBTs need a more general approach to conquer the problem. First, our DBT scans all instructions in Loop (line 2) BB and determines whether an instruction is memory access to the stack (line 3). If current instruction ( $Inst_i$ ) loads the data from the stack, our DBT would determine whether the memory address is referenced by prior instructions. If such an address does not appear before, the current load should be promoted (lines 11 to 12). Otherwise, the operands containing current instruction should be replaced with the last modified operand (line 9). On the other hand, if the store moves the data to the stack, our DBT should mark the address touched by store instruction even though such position is accessed by prior loads (line 15) and extract the operand to update the last operands corresponding to the target address (line 17 and 19).

It is possible for the application binary to update variables spilled to stack. Here is a simple case shown in Figure 3.2b, ③⑥ is an example that an induction variable loads the data from address  $sp$ , increments the value, and then stores the data back to the same position in the stack. In this case, DBTs should update the data changes in the loop. For the loads first accessing the stack address, we preload the data in VRP BB before entering Loop BB (line 24). If DBTs detect data changes in such position, DBTs would create a Phi Node in Loop BB to update the value from both VRP BB and Loop BB (lines 25 to 27). Figure 3.2c shows the value update in the induction variable (③⑥) with a Phi Node after promotion. Lastly, update all operands in each instruction where such operand is filled in UpdateOperand (lines 35 to 38), remove the spilled loads which do not need to be promoted (line 31), and then store the spilled variables back to the stack in STVR BB when exiting the Loop BB (line 34). After successfully vectorizing the loop (Figure 3.2d), DBTs would insert the checking code according to *RTCheck* table and follow the rules we present in the previous paragraph. The elements in set *SS* and *SL* will be defined by *RTCheck*.  $SS = \{a | addr \leq a < addr + size(ss), (addr, ss) \in RTCheck, ss \in Store\}$ ,  $SL = \{a | addr \leq a < addr + size(sl), (addr, sl) \in RTCheck, sl \in Load\}$

Consequently, our DBT needs to insert  $(nss \times (nl + ns) + nsl \times ns)$  checking pairs to validate the legality of promotion since the access range of each spilled variable is independent. Nevertheless, the more spilled variables a DBT promotes, the more checking pairs the DBT should insert. Even though DBTs only check aliasing once before entering Loop BB, such aliasing check still could be costly when there are many checking pairs inserted. To further reduce such overhead, we present a trade-off aliasing detection between checking overhead and checking granularity. We know that a stack variable access consists of stack pointer register and an offset, so a DBT is able to check whether operands are both constant beforehand. Hence, a DBT can only check the access range of spilled variables from the lowest to the highest address, where a DBT checks only one pair for each regular memory reference. The following equation shows the modified checking function of virtual register promotion:

$$SV = \{a | \min\{SS \cup SL\} \leq a \leq \max\{SS \cup SL\}\} \quad (3.7)$$

$$\text{Legality} = \begin{cases} true, & \text{if } SV \cap (LD \cup ST) = \phi \\ false, & \text{otherwise} \end{cases} \quad (3.8)$$

We define a new access set  $SV$  for spilled variables (eq. 3.7) and the legality of virtual register promotion (eq. 3.8). The new-defined validation (eq. 3.8) is more restricted than the original one (eq. 3.6) because set  $A$  (eq. 3.5) is a subset of  $SV$ . False positives may exist. For example, if non-aliased memory access to a position in this range (eq. 3.8), it will fail the legality check even if this reference is harmless. However, in practice, the checking range is usually small because the compilers spill variables to a small, consecutive area. Therefore, the new and improved rule is more cost effective as our benchmark results shown.

Figure 3.3 shows an example of two different approaches of aliasing check. Assume we have two non-spilled loads and one non-spilled store (i.e., memory reference from  $addr$  to  $addr + tc$ ), and six 4-byte spilled variables with two stores and four loads. Figure 3.3a shows a DBT needs to do 10 pairs of aliasing check for each spilled variable to satisfy the original rule in equation 3.6. Although such aliasing check satisfies the requirements of validation but causes more too much checking overhead. On the other hand, Figure 3.3b shows a DBT only needs to do 3 pairs using the new rule since we are able to figure out the maximum and minimum ad-

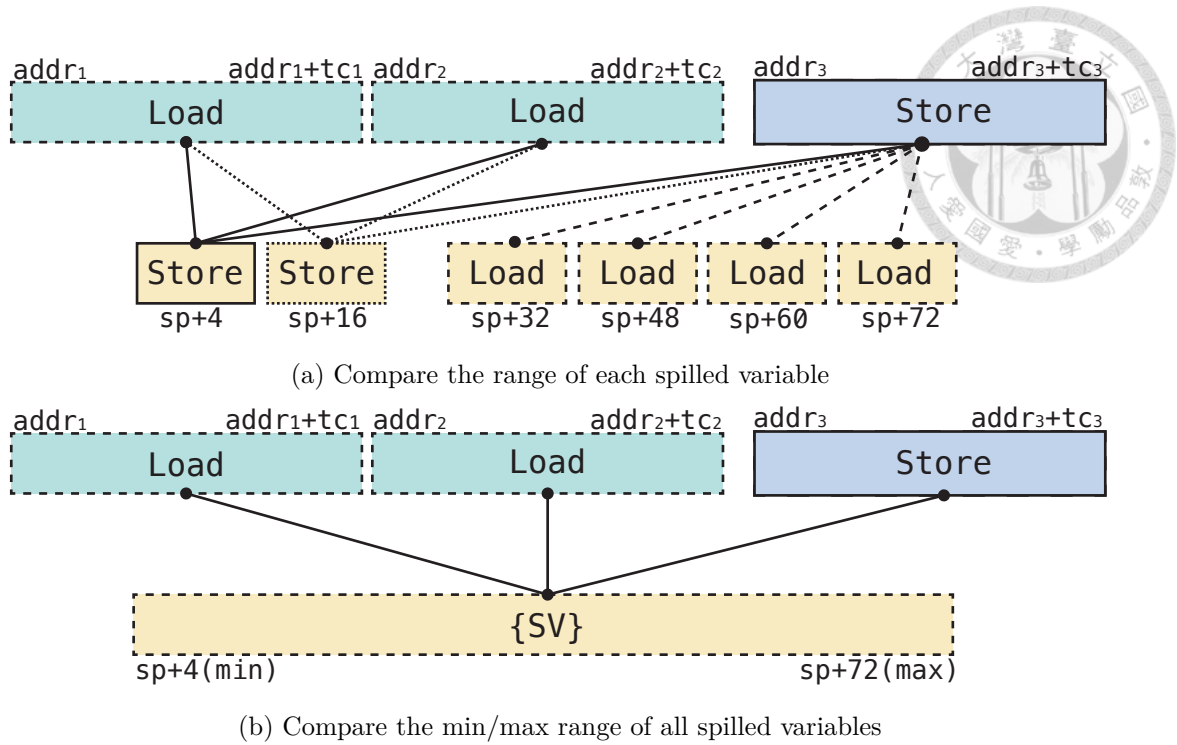


Figure 3.3: Runtime Check for Virtual Register Promotion

dress of both spilled loads and stores from  $sp+4$  to  $sp+72$ , and thus we check such access ranges for max/min address of all spilled variables rather than each individual ones. Such aliasing detection (eq. 3.8) is lightweight but more conservative than Figure 3.3a. Although there is a corner case of unknown offset values, a DBT can directly compare such memory access to all non-spilled loads and stores.



**Algorithm 1: Virtual Register Promotion**

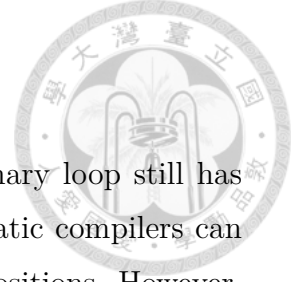

---

```

input : VRP, Loop, STVR
output: RTCheck, VRP, Loop, STVR
1 Function BuildSpilledVariableTable(Loop)
2   for  $i \leftarrow 1$  to  $Loop.Size()$ ,  $\forall Inst_i \in Loop$  do
3     if  $Inst_i \notin SpilledVariables$  then
4       continue
5      $Addr \leftarrow GetLoadStoreAddress(Inst_i)$ 
6      $(BeginInst, LastInst) \leftarrow SVLookup[Addr]$ 
7     if  $Inst_i \in Load$  then
8       if  $Addr \in SVLookup$  then
9          $UpdateOperand[Inst_i] \leftarrow LastInst$ 
10      else
11         $SVLookup[Addr] \leftarrow (Inst_i, Inst_i)$ 
12         $RTCheck[Addr] \leftarrow Inst_i$ 
13      else if  $Inst_i \in Store$  then
14         $Operand \leftarrow GetStoreOperand(Inst_i)$ 
15         $RTCheck[Addr] \leftarrow Inst_i$ 
16        if  $Addr \in SVLookup$  then
17           $SVLookup[Addr] \leftarrow (BeginInst, Operand)$ 
18        else
19           $SVLookup[Addr] \leftarrow (Inst_i, Operand)$ 
20 Function PromoteSpilledVariable(VRP, Loop, STVR)
21   BuildSpilledVariableTable(Loop)
22   foreach  $(\sim, (BeginInst, LastInst)) \in SVLookup$  do
23     if  $BeginInst \in Load$  then
24        $BeginInst.MoveTo(VRP)$ 
25       if  $BeginInst \neq LastInst$  then
26          $Phi \leftarrow CreatePHI(Loop, BeginInst, LastInst)$ 
27          $UpdateOperand[BeginInst] \leftarrow Phi$ 
28   for  $i \leftarrow 1$  to  $Loop.Size()$ ,  $\forall Inst_i \in Loop$  do
29     if  $Inst_i \in SpilledVariables$  then
30       if  $Inst_i \in Load$  then
31          $Inst_i.Remove()$ 
32         continue
33       else if  $Inst_i \in Store$  then
34          $Inst_i.MoveTo(STVR)$ 
35     foreach  $Operand \in Inst_i.GetAllOperands()$  do
36        $NewOperand \leftarrow UpdateOperand[Operand]$ 
37       if  $NewOperand \neq \phi$  then
38          $Operand \leftarrow NewOperand$ 

```

---



### 3.4 Speculative Execution

In addition to the issue of spilled variables, vectorizing a binary loop still has other issues of memory aliasing. With high-level information, static compilers can identify the array base address of memory references to specific positions. However, DBTs cannot obtain such information when translating guest binaries because such addresses are generally put in registers, where the access positions can only be determined at run time. For instance, as illustrated in Figure 2.1c, the compilers usually place the addresses of memory reference  $\textcircled{A}$  $\textcircled{B}$  in registers (r0 and r1). If DBTs vectorize the loop, such memory accesses following induction variables may be aliased to each other since DBTs cannot figure out whether the access ranges of vector memory references overlap with the other one. Therefore, DBTs should check the dependency at run time among memory references. The equation below shows the complete runtime check of such memory aliasing, and we follow the same definition for some variables mentioned in the previous section:

VL: vector length

$$Vec(I) = [addr(I), addr(I) + VL \times size(I)], \forall I \in \text{Vectors}$$

$$Inv(I) = [addr(I), addr(I) + size(I)], \forall I \in \text{Loop Invariants}$$

*AliasVector* =

$$\bigcup_{\substack{i=ns-1, j=ns, k=nl \\ i=1, j>i, k=1}}^{i=ns-1, j=ns, k=nl} \left( Vec(st_i) \cap (Vec(st_j) \cup Vec(ld_k)) \right) \quad (3.9)$$

*AliasInv* =

$$\bigcup_{i=1}^{i=ns} \left( Inv(st_i) \cap (ST \cup LD) \right) \cup \bigcup_{i=1}^{i=nl} \left( Inv(ld_i) \cap ST \right) \quad (3.10)$$

$$\text{Legality} = \begin{cases} true, & \text{if } AliasVector \cup AliasInv = \phi \\ false, & \text{otherwise} \end{cases} \quad (3.11)$$

We consider two cases of memory aliasing, respectively shown in eq. 3.9 and 3.10. One is aliasing between memory references with vector operations, and the other is aliasing between loop invariants and other memory references. The former is a simple case, where DBTs can take advantage of vector properties to check if memory accesses with vector are overlapping after vectorizing the loop. The latter,

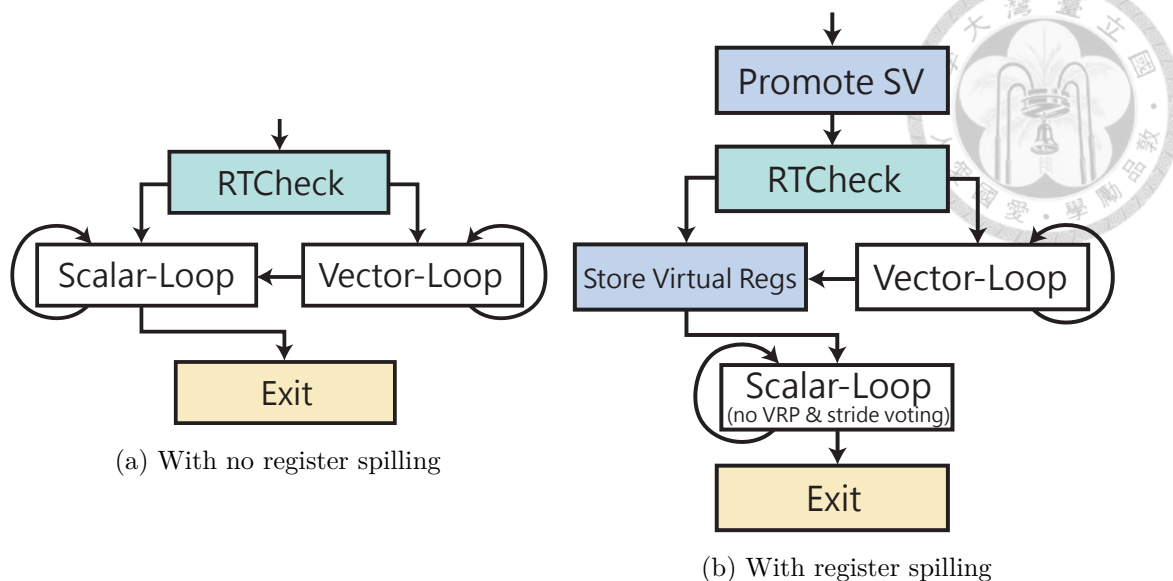


Figure 3.4: Rewrite Control Flow for Vectorization

however, DBTs should guarantee the data in such positions cannot be modified by other memory references. In this case, DBTs would check whether such positions are touched across all iterations and ensure no aliasing exists when executing the vectorized loop.

Moreover, modern compilers (e.g., GCC) sometimes do not encode constant stride within instructions but move it to the registers or stack (i.e, register spilling), such as Roms in SPEC2017. Fortunately, we still have opportunities to determine the distance of strides because the strides are not always placed in registers or stack. To pursue more vectorization capability, we adopt "stride distance voting," voting for the step distance of memory references following the induction variables, and DBTs would check whether the stride distance matches the real one at runtime after successfully vectorizing the loops. For instance, as illustrated in Figure 3.2c, assume the constant stride '8' is spilled to stack, and a DBT has already promoted such a constant variable to a virtual register. Since the constant value in such a register can only be determined at run time, DBTs would create a voting box for other induction variables (e.g., ⑥) to figure out which stride distance is in majority. Then, DBTs would patch the unknown stride to the voting result and insert the checking code to verify whether the voting result is the same as the constant value in registers.

To guarantee the correctness of code execution, DBTs need to verify the legal-

ity of virtual register promotion, detect data dependence among general memory references, and compare the voted stride at run time. Next, DBTs would employ both runtime check and loop vectorization to rewrite the control flow composed of the basic blocks in Figure 3.4 to execute the vectorized loop, and then go through the remaining iterations in the scalar loop. This is because DBTs need to synchronize architecture states in Exit BB. For a simplified implementation, we employ the mechanism of state synchronization of scalar loop rather than extract the elements from vector register in the vectorized loop to maintain consistent states. In addition, we define two different vectorization approaches in DBTs. If there is no register spilling in a loop, DBTs just need to transform the scalar loop into Figure 3.4a, which only checks the data dependence and voted stride distance. If a loop contains spilled variables, we provide another transformation approach shown in Figure 3.4b. The rewriting steps would be: **a)** Promote spilled variables (SV) to virtual registers. **b)** Check all possible aliasing mentioned in the previous section at run time. If the verification failed, then just store the virtual register back to the stack and execute the scalar loop with no promoted variables. **c)** If the validation is successful, just execute the vectorized loop with register promotion. **d)** After the execution of the vectorized loop is done, store the data from virtual registers back to the stack and then execute the remaining iterations in the scalar loop.



## Chapter 4

# Performance Evaluation

We implement our optimization approach in HQEMU, a cross-ISA DBT system based on the LLVM 6.0 JIT compiler, and evaluate the effectiveness with ARMv7 scalar to ARMv8 vector (NEON with 128-bit SIMD registers) translation with double precision computation. Although ARMv7 and ARMv8 may sound like a compatible processor family, so a DBO may seem more appropriate. In fact, ARMv7 and ARMv8 are incompatible ISAs. So a DBT is selected for our testbed. HQEMU first translates ARMv7 scalar binary to LLVM IR, transformed by our optimization approach to apply vectorization, and then LLVM backend generates the host ARMv8 binary. The host system for ARMv7 to ARMv8 translation is Odroid-C2 with ARM Cortex-A53 CPU at 1.5Ghz with 2GB of RAM running Linux 3.14.79, and such platform supports legacy 32-bit (AArch32) execution mode for ARMv7 binaries. Therefore, we could run both ARMv7 and ARMv8 binaries on the same platform.

We select several benchmark kernels, in which the time-consuming loops are vectorizable, from various benchmark suites across scientific computing and linear algebra, such as Livermore Loops (LL), PolyBench (POLY), Netlib BLAS (BLAS), SPEC2000, SPEC2006, and SPEC2017 (SPEC), to evaluate the vectorization capability and performance improvement in our DBT. Register spilling often occurs in real-world applications containing big loops with longer define-use chains. We observe such loops in SPEC suites and show the percentage of execution time and the ratio of loops with register spilling for those kernels in Table 4.1 and 4.2, where the large applications contain more register spilling in innermost loops. All bench-

Table 4.1: Benchmark Kernels from SPEC

Kernel Name	SPEC Version	Benchmark Name	Execution Time(%)	Source
SPEC-psinv	2000	172.mgrid	24.53%	mgrid.f: PSINV()
SPEC-resid	2000	172.mgrid	54.88%	mgrid.f: RESID()
SPEC-calc2	2000	171.swim	32.40%	swim.f: CALC2()
SPEC-fluxi	2006	437.lesli3d	11.97%	tml.f: FLUXI()
SPEC-fluxj	2006	437.lesli3d	13.26%	tml.f: FLUXJ()
SPEC-fluxk	2006	437.lesli3d	16.83%	tml.f: FLUXK()
SPEC-extrapi	2006	437.lesli3d	14.55%	tml.f: EXTRAPI()
SPEC-extrapj	2006	437.lesli3d	13.80%	tml.f: EXTRAPJ()
SPEC-extrapk	2006	437.lesli3d	9.55%	tml.f: EXTRAPK()
SPEC-setbc	2006	437.lesli3d	13.94%	tml.f: SETBC()
SPEC-step2d	2017	654.roms_r	28.78%	step2d_LF_AM3.h: lines:989-2284
SPEC-uv3dmix2	2017	654.roms_r	6.23%	uv3dmix_2.h: uv3dmix2_tile()

marks were compiled using GCC 5.4.1 with flags "-O2 -ftree-vectorize -ffast-math." The ARMv7 binaries do not contain vector operations because ARMv7 does not support NEON SIMD with double precision computation, so the compilers would not vectorize the loops. In contrast, ARMv8 has such support, so compilers could exploit NEON SIMD with double precision. When translating ARMv7 scalar binaries to ARMv8, we vectorize the binaries at the same time. The theoretical performance speedup would achieve 2x with register width from 64-bits (scalar) to 128-bits (NEON SIMD).

Table 4.2: Number of Loops with Register Spilling in Kernels

Kernel Name	Total Num of Loops	Num of Loops with Register Spilling
BLAS-symm	2	1
LL-adi_integ	1	1
LL-diff_pred	1	1
LL-2d_frag	3	1
SPEC-psinv	1	1
SPEC-resid	1	1
SPEC-calc2	1	1
SPEC-fluxi	9	4
SPEC-fluxj	10	4
SPEC-fluxk	10	3
SPEC-setbc	6	6
SPEC-step2d	20	6
SPEC-uv3dmix2	4	4

## 4.1 Overall Performance

We report the performance results in Figure 4.1 to show our vectorization capability in our DBT, and the baseline is normalized to ARMv7 native run (ARMv7-Scalar-Native) with scalar operations. DBT-Vec means the best performance improvement of our approach, which adopts lightweight aliasing detection (i.e., Figure 3.3b), and ARMv8-Vec-native is ARMv8 binaries natively run on the host with NEON SIMD. We choose ARMv8 NEON native run as our reference of maximum performance speedup since ARMv8 NEON supports 128-bit SIMD width.

On average, DBT-Vec could achieve 1.42x performance speedup in comparison

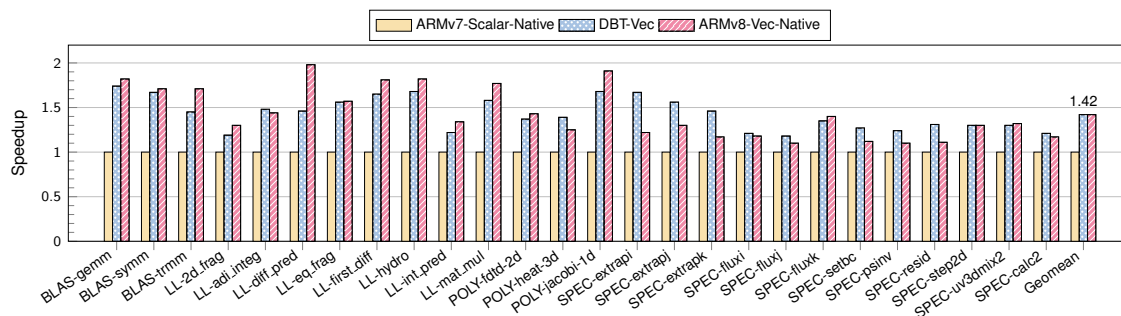


Figure 4.1: Performance results of ARMv7-to-ARMv8 Translation

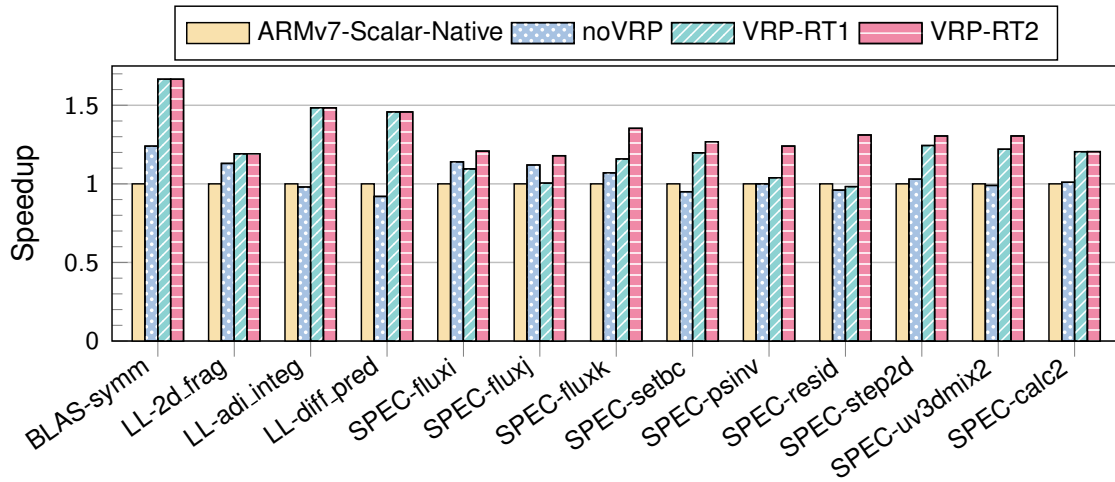
with ARMv7-Scalar-Native and is close to ARMv8-Vec-Native. Besides, DBT-Vec has even more performance improvement than ARMv8-Vec-Native in several kernels, such as the kernels in leslie3d. For kernel SPEC-extrapj and SPEC-extrapk, ARMv8-Vec-Native obtains less performance gain than DBT-Vec. This is because ARMv8-Vec-Native contains more spilled variables than DBT-Vec does and causes more memory reference overhead for data movement between registers and stack. The rules of register spilling follow register allocation in different compiler backends, and such rules in GCC and LLVM backends are different. As for another reason, the granularity of code segment also affects the ratio of register spilling in loops. For ARMv8-Vec-Native, the GCC backend needs to consider the life range of variables across basic blocks in a function for register allocation. For DBT-Vec, however, the LLVM backend only needs to consider the scope of the innermost loop with a smaller range to allocate registers. Such behaviors could also be found in SPEC-calc2 for ARMv8-Vec-Native, which has performance loss due to unwanted register spilling.

For SPEC-setbc and POLY-head3d, GCC does not successfully vectorize the loops in such kernels for ARMv8-Vec-Native because the vectorizable loops do not cross the default threshold of cost model in GCC. In this case, if we force GCC to vectorize the loops by passing the flag "-fvect-cost-model=unlimited," the performance improvement of ARMv8-Vec-Native would be close to DBT-Vec. As for benchmark LL-diff\_pred, however, DBT-Vec has a performance gap between ARMv8-Vec-native because of runtime check overhead for virtual register promotion, and we would report such detailed information in chapter 4.2.

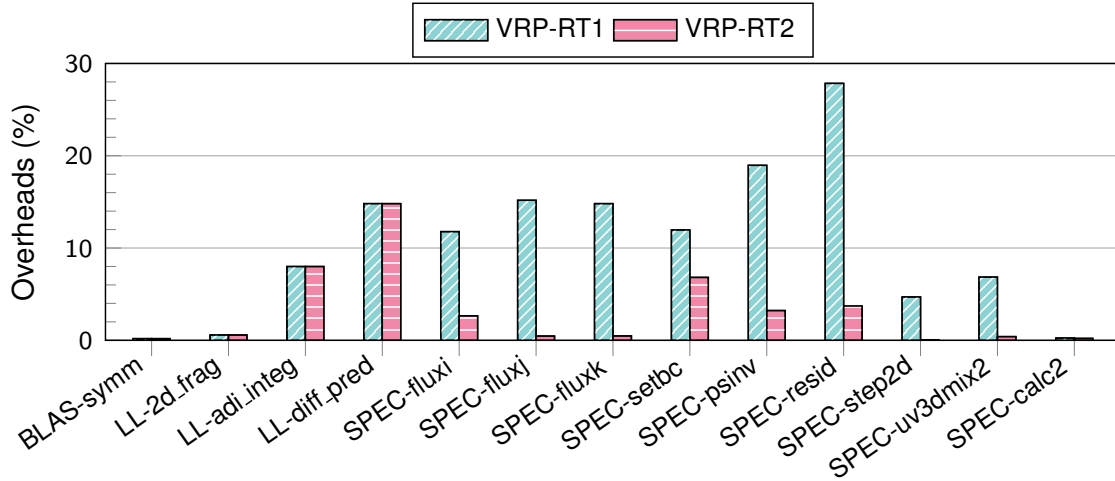
## 4.2 Evaluation of Virtual Register Promotion

There are 13 benchmark kernels containing spilled variables, and thus we further evaluate the performance improvement and overheads for virtual register promotion. Figure 4.2a shows the performance results for different vectorization approaches. Such improvement results would correspond to the number of loops containing spilled variables as shown in Table 4.2. For noVRP, our DBT does not promote spilled variables to virtual registers, so the loops containing spilled variables would not be vectorized. On the other hand, VRP-RT1 checks aliasing for each spilled





(a) Performance of virtual register promotion



(b) Overheads (%) of two runtime aliasing checks

Figure 4.2: Evaluation results of virtual register promotion

variables (i.e., Figure 3.3a), and VRP-RT2 is a trade-off approach of runtime aliasing check (i.e., Figure 3.3b). The overhead report of those different aliasing check policies is also shown in Figure 4.2b. (i.e., VRP-RT2 in Figure 4.2 is the same result as DBT-Vec in Figure 4.1.)

Because of registers spilling in guest binaries, noVRP cannot vectorize the loops with spilled variables. Therefore, VRP-RT1 generally achieves greater performance gain than noVRP since VRP-RT1 has a higher vectorization ratio. However, SPEC-fluxi and SPEC-fluxj still have lower performance improvement than noVRP does. The reason is our DBT has to check aliasing for a large number of spilled variables at run time, and the checking overhead degrades the performance. The results are shown in Figure 4.2b, SPEC-fluxi and SPEC-fluxj respectively spend 11% and 15% of execution time to perform alias checking. Hence, lightweight aliasing detection (VRP-RT2) is crucial for virtual register promotion. As the results show, the overhead of SPEC-fluxi and SPEC-fluxj in VRP-RT2 has been respectively decreased to 2.6% and 0.46%, and the performance improvement of VRP-RT2 is more than noVRP.

For other benchmark kernels, SPEC-fluxk, SPEC-psinv, and SPEC-resid, VRP-RT2 achieves greater performance gain than VRP-RT1. Especially for benchmark kernel SPEC-resid, VRP-RT2 reduces the overhead from 27% down to 3.7% and achieves nearly 1.3x speedup in comparison with VRP-RT1. This is because VRP-RT1 inserts 233 aliasing check pairs to satisfy the rule in Figure 4a. In contrast, VRP-RT2 only creates 29 checking pairs for the same loop with much reduced checking overhead. For kernels LL-2d\_frag, SPEC-calc2, LL-adi\_integ, and LL-diff\_pred, the difference in performance improvement and overheads are not notable because our DBT checks fewer spilled variables for those kernels. As for kernel LL-diff\_pred, mentioned in chapter 4.1, DBT-Vec has a gap of performance improvement between ARMv8 native run because DBTs should pay 15% checking overhead to detect aliasing (VRP-RT2 in Figure 4.2b) for spilled variables. On the other hand, the constant stride in kernels SPEC-step2d and SPEC-uv3dmix are commonly allocated to registers or stack rather than encoded in instructions. In this case, DBTs would patch the unknown constant stride and attempt to vectorize the loops. As a result, our DBT successfully vectorizes the loops and obtain significant

#### 4.2. EVALUATION OF VIRTUAL REGISTER PROMOTION

performance gain in these kernels.





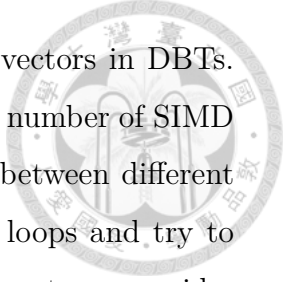
## Chapter 5

# Related Work

For cross-ISA DBTs, QEMU [10] emulates both user space program and full system including operating system, where DBT translates guest binaries to TCG-IR and then generate host binaries for various target machines. MAMBO-X64 [5] is an ARMv7 to ARMv8 DBT, migrating legacy ARMv7 applications to the ARMv8 platform with no 32-bit hardware support. Transmeta CMS [15] could migrate IA-32 applications to their Crusoe VLIW architecture and exploit hardware to solve the issues of memory ordering after executing aggressive re-optimized code. On the other hand, PIN [16] and Valgrind [17] conduct dynamic binary instrumentation to collect profiles and program detailed behavior information. Dynamo [18], DynamoRIO [19] and Adore [20] exploit software and hardware profiling to figure out the time-consuming code segments and rewrite the binary for better performance.

Auto vectorization could be classified in two categories: loop-based vectorization [21]–[23] and basic block vectorization [24]–[26]. The former exploits loop information to widen the computation across iterations and combine them with vector operations. Inspired by these approaches, such techniques could also be used to migrate scalar binaries to hosts with SIMD capability. Unlike loop-based vectorization, the latter combines multiple statements with isomorphic operations in basic blocks into SIMD operations, such as SLP [24], PSLP [27], and Partial SIMD parallelism [28].

For binary level vectorization, several approaches have been proposed to improve DLP for binaries without high-level information. Hong. [6] and Liu. [7] presented short-vector to long-vector to exploit longer SIMD lanes in cross-ISA DBTs. The



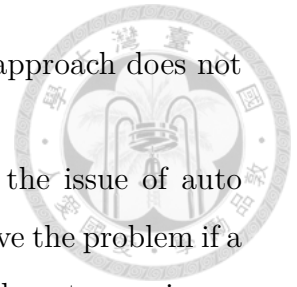
former re-vectorizes the loops containing short vectors to longer vectors in DBTs. The latter translates guest codes to hosts with the wider but fewer number of SIMD registers to resolve the issues of asymmetric SIMD configuration between different ISAs. Besides, they consider input binaries with unrolled SIMD loops and try to reduce SIMD register spilling in the translated host codes. In contrast, we consider register spilling in the input binaries prohibiting DBT vectorization. The other related works are [29], [30], translating non-vectorized x86 binaries to SIMD with both SSE and AVX2 on same-ISA DBT. The former employs McSema [31] to translate x86 binaries to LLVM IR and then exploit the Polly framework [32] to re-vectorize the code. However, they do not illustrate how to deal with data dependency when applying vectorization on binary loops with Polly. The Janus Triad [30] uses static binary analysis to determine whether runtime check is needed to enable the vectorization of a loop. In addition, none of these works mentioned the issues of register spilling caused failed vectorization.

As for register promotion, Li. [33] promotes stack variables to additional general-purpose registers in order to exploit the increased number of registers when migrating the legacy x86 applications from IA32 machines to x64 machines. This work employs shared memory protection mechanisms with a two thread execution model to detect possible aliasing on the stack. When true aliasing is detected, a DBT would recover the architecture states and resume execution. Such an approach employs hardware support to achieve low overhead for aliasing detection. Nevertheless, the mechanism of data recovery cannot be applied on loop vectorization because loop vectorization would aggressively reorder the instructions across iterations and require a different approach for aliasing check.

SecondWrite [34] is a static binary analysis framework presenting a software-based aliasing check for symbol promotion of stack variables to ensure the correctness of code execution and binary rewriting, where such framework synchronizes the promoted stack variables before/after memory instructions whose access range is not statically determinable. In contrast, our approach exploits loop informations to assist DBTs in aliasing detection before entering the loops so that DBTs could achieve lower overheads of runtime check and perform vectorization. Besides, SecondWrite is based on the static binary translation, difficult to handle the self-

modified code, code discovery, and code location problems. Our approach does not have such limitations because it is based on DBT.

For auto parallelization for binaries, [35], [36] also address the issue of auto parallelization in a static binary translator, and they also try to solve the problem if a affine loop does not contain symbolic information. However, they do not cover issues of parallelization or vectorization capability if the binary loops have register spilling. In contract, our approach is auto vectorization in dynamic binary translation, and we carefully consider the issues of register spilling when conducting Scalar Evolution Analysis for auto vectorization.





## Chapter 6

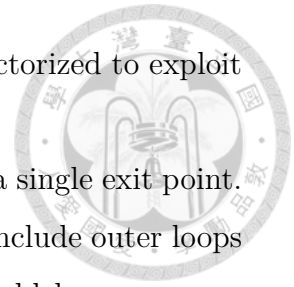
# Conclusion and Future Work

SIMD processing capability is increasingly important in microprocessor architectures. New microprocessors often come up with enhanced and more powerful SIMD capabilities. However, legacy application binaries may not benefit from such newly delivered performance. Dynamic binary translation or optimization could apply some transformations to turn scalar loops in legacy application binaries into vector or SIMD forms. Prior work has been successful in expanding short SIMD to longer SIMD. However, vectorizing scalar loops in legacy binaries has not been very successful. One of the main reasons is that some critical information for vectorization has been lost due to register spilling. One way to recover such important information is to promote spilled variables back to register and enable effective vectorization. This is rather challenging since the binary translator must deal with possible memory aliasing to the stack such as local scalar and array accesses. Although runtime checking could be applied to ensure the correctness of virtual register promotion, previously proposed checking could often require excessive overhead and diminish the benefit of vectorization.

This work proposes using virtual register promotion to recover critical information for scalar loop vectorization. It also comes up with a cost-effective aliasing check for stack variables and spilled variables so that the runtime checking would not block the way to efficient SIMD execution. The evaluation of our approach is based on HQEMU, and our DBT translates ARMv7 application binaries to run on the ARMv8 Cortex-A53 processor. The set of benchmark kernels have shown 1.42x speedup over native runs of ARMv7 binaries on ARMv8. We have success-

fully demonstrated that scalar legacy code could effectively be vectorized to exploit new SIMD capabilities available on the host machine.

Currently, we only consider the cases of innermost loops with a single exit point. In the future, we would like to extend the vectorization scope to include outer loops or unrolled loops where the register spilling caused troubles would be even more common in preventing scalar loops in legacy application binaries to be vectorized via DBT.

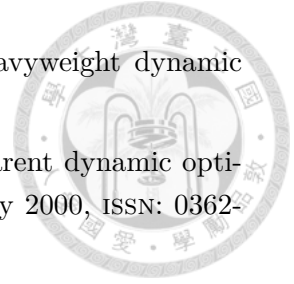




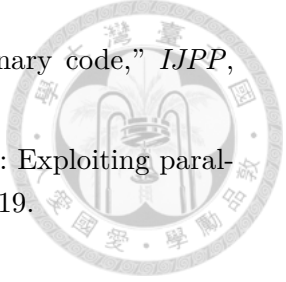


## Reference

- [1] R. B. Lee, “Subword parallelism with max-2,” *IEEE Micro*, 1996.
- [2] J. E. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Jan. 2005.
- [3] C. Zheng and C. Thompson, “Pa-risc to ia-64: Transparent execution, no recompilation,” *Computer*, 2000.
- [4] *Apple’s rosetta*, <https://www.apple.com/rosetta/index.html>, 2006.
- [5] A. D’Antras, C. Gorgovan, J. Garside, and M. Luján, “Low overhead dynamic binary translation on arm,” ser. PLDI’17, 2017.
- [6] D. Hong *et al.*, “Exploiting longer simd lanes in dynamic binary translation,” in *ICPADS*, 2016.
- [7] Y. Liu *et al.*, “Exploiting asymmetric simd register configurations in arm-to-x86 dynamic binary translation,” ser. PACT, 2017.
- [8] E. Duesterwald and V. Bala, “Software profiling for hot path prediction: Less is more,” *SIGPLAN Not.*, 2000.
- [9] D. Hong *et al.*, “Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores,” ser. CGO’12, 2012.
- [10] F. Bellard, “Qemu, a fast and portable dynamic translator,” ser. USENIX ATC’05, 2005.
- [11] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” ser. CGO ’04, Mar. 2004.
- [12] J. L. Henning, “Spec cpu2000: Measuring cpu performance in the new millennium,” *Computer*, vol. 33, no. 7, pp. 28–35, Jul. 2000, ISSN: 0018-9162.
- [13] J. L. Henning, “Spec cpu2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006, ISSN: 0163-5964.
- [14] *Spec cpu 2017*, <https://www.spec.org/cpu2017/>, 2017.
- [15] J. Dehnert *et al.*, “The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges,” ser. CGO’03, 2003.
- [16] C. Luk *et al.*, “Pin: Building customized program analysis tools with dynamic instrumentation,” ser. PLDI ’05, 2005.



- [17] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” *SIGPLAN Not.*, 2007.
- [18] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: A transparent dynamic optimization system,” *SIGPLAN Not.*, vol. 35, no. 5, pp. 1–12, May 2000, ISSN: 0362-1340.
- [19] D. Bruening, E. Duesterwald, and S. Amarasinghe, “Design and implementation of a dynamic optimization framework for windows,” Jan. 2002.
- [20] J. Lu *et al.*, “Design and implementation of a lightweight dynamic optimization system,” *JILP*, 2004.
- [21] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, “Automatic intra-register vectorization for the intel architecture,” *Int. J. Parallel Program.*, vol. 30, no. 2, pp. 65–98, Apr. 2002, ISSN: 0885-7458. [Online]. Available: <http://dl.acm.org/citation.cfm?id=586554.586555>.
- [22] D. Naishlos, “Autovectorization in gcc,” *Proceedings of the 2004 GCC Developers Summit*, pp. 105–118, 2004. [Online]. Available: <ftp://gcc.gnu.org/pub/gcc/summit/2004/Autovectorization.pdf>.
- [23] N. Sreeram and R. Govindarajan, “A vectorizing compiler for multimedia extensions,” *Int. J. Parallel Program.*, vol. 28, no. 4, pp. 363–400, Aug. 2000, ISSN: 0885-7458. DOI: 10.1023/A:1007559022013. [Online]. Available: <http://dx.doi.org/10.1023/A:1007559022013>.
- [24] 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*, ser. PLDI '00, Vancouver, British Columbia, Canada: ACM, 2000, pp. 145–156, ISBN: 1-58113-199-2. DOI: 10.1145/349299.349320. [Online]. Available: <http://doi.acm.org/10.1145/349299.349320>.
- [25] R. L. Leupers and S. Bashford, “Graph-based code selection techniques for embedded processors,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 4, pp. 794–814, Oct. 2000, ISSN: 1084-4309. DOI: 10.1145/362652.362661. [Online]. Available: <http://doi.acm.org/10.1145/362652.362661>.
- [26] S. Kral, F. Franchetti, J. Lorenz, and C. Ueberhuber, “Simd vectorization of straight line fft code, euro-par 2003. parallel processing, 9th international euro-par conference, klagenfurt, austria, august 26-29, 2003. proceedings,” vol. 2790, Aug. 2003, pp. 251–260. DOI: 10.1007/978-3-540-45209-6\_39.
- [27] V. Porpodas, A. Magni, and T. M. Jones, “Pslp: Padded slp automatic vectorization,” in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2015, pp. 190–201. DOI: 10.1109/CGO.2015.7054199.
- [28] H. Zhou and J. Xue, “A compiler approach for exploiting partial simd parallelism,” *ACM Trans. Archit. Code Optim.*, 2016.

- 
- [29] N. Hallou *et al.*, “Runtime vectorization transformations of binary code,” *IJPP*, 2017.
- [30] R. Zhou, G. Wort, M. Erdős, and T. M. Jones, “The janus triad: Exploiting parallelism through dynamic binary modification,” ser. VEE 2019, 2019.
- [31] *Mcsema*, <https://github.com/trailofbits/mcsema>, 2014.
- [32] C. Lengauer, “Polly—performing polyhedral optimizations on a low-level intermediate representation,” *Parallel Processing Letters*, 2012.
- [33] J. Li *et al.*, “Dynamic register promotion of stack variables,” ser. CGO’11, 2011.
- [34] K. Anand *et al.*, “A compiler-level intermediate representation based binary analysis and rewriting system,” ser. EuroSys ’13, 2013.
- [35] A. Kotha *et al.*, “Automatic parallelization in a binary rewriter,” ser. MICRO’43, 2010.
- [36] A. Kotha, K. Anand, T. Creech, K. Elwazeer, M. Smithson, and R. Barua, “Affine parallelization of loops with run-time dependent bounds from binaries,” ser. EOSP’14, Apr. 2014.