The Janus Triad: Exploiting Parallelism Through Dynamic Binary Modification

Ruoyu Zhou, George Wort, Márton Erdős, Timothy M. Jones
University of Cambridge, UK
{ruoyu.zhou,*,me412,timothy.jones}@cl.cam.ac.uk
*georgewort11@gmail.com

Abstract

We present a unified approach for exploiting thread-level, data-level, and memory-level parallelism through a same-ISA dynamic binary modifier guided by static binary analysis. A static binary analyser first examines an executable and determines the operations required to extract parallelism at runtime, encoding them as a series of rewrite rules that a dynamic binary modifier uses to perform binary transformation. We demonstrate this framework by exploiting three different kinds of parallelism to perform automatic vectorisation, software prefetching, and automatic parallelisation together on legacy application binaries. Software prefetch insertion alone achieves an average speedup of 1.2×, comparing favourably with an automatic compiler pass. Automatic vectorisation brings speedups of 2.7× on the TSVC benchmarks, significantly beating a compiler approach for some workloads. Finally, combining prefetching, vectorisation, and parallelisation realises a speedup of 3.8× on a representative application loop.

CCS Concepts • **Software and its engineering → Runtime environments**; *Retargetable compilers*; *Dynamic compilers*.

Keywords binary translation, binary optimisation, vectorization, software prefetch

ACM Reference Format:
Ruoyu Zhou, George Wort, Márton Erdős, Timothy M. Jones. 2019. The Janus Triad: Exploiting Parallelism Through Dynamic Binary Modification. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3313808.3313812

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313812

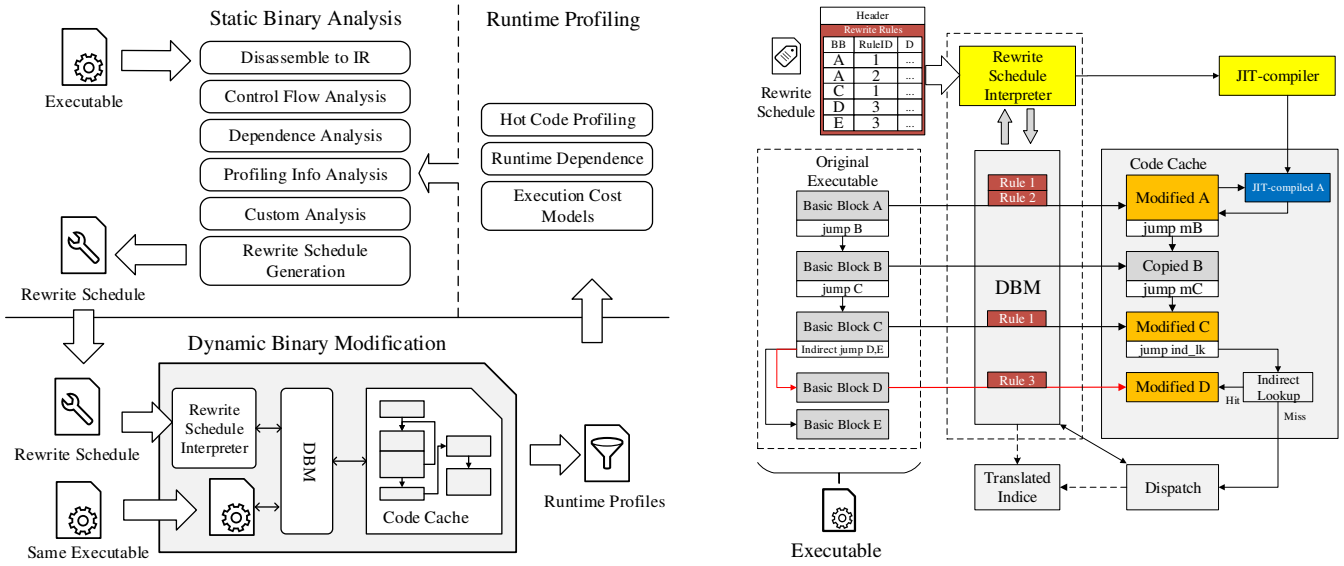
1 Introduction

Multicore processors support a variety of different kinds parallelism to meet the varying needs of a wide range of workloads. Among them, thread-level parallelism (TLP) relies on decomposing workloads into threads at a coarse-grained level and running them on different cores. SIMD processing exploits data-level parallelism (DLP) whereby the same operation is executed on multiple data points simultaneously. Meanwhile, memory-level parallelism (MLP) is sometimes considered a form of instruction-level parallelism (ILP), but comes into its own when exploited using coarse-grained software prefetching techniques to boost cache performance and memory port utilisation.

However, exploiting these various forms of parallelism is workload and processor-specific. SIMD instruction set architectures (ISAs) change regularly (for example, x86 architectures support MMX, SSE and AVX vector ISAs developed by Intel, not to mention those created by AMD), the number of cores on chip is growing slowly, and the configuration of the memory hierarchy has a profound effect on whether software prefetching is useful or not. As such, it is increasingly difficult for users of proprietary and legacy software to specialise their codes for different chips, and even for open-source software, end users often receive their applications in binary form or link against pre-compiled libraries. In many cases the applications can only be compiled with generic optimisations that do not take full advantage of the capabilities of the underlying hardware.

Therefore, optimising applications through binary modification that targets the same ISA becomes a seductive proposition. There are many static binary modification tools (for example, Vulcan [22], BOLT [20], Sun BCO [14], SecondWrite [2], Ispike [17]) that can covert a generic executable to a specialised binary, some relying on profiling or instrumentation. However, most focus on control-flow and peephole optimisations and therefore cannot handle complicated and stripped binaries. Without additional symbolic information, they are not able to analyse and perform sophisticated transformations that maintain the original program semantics.

Dynamic binary modification (DBM) normally sets up a virtualised environment and executes the application just-in-time in its sandboxed code caches. This is flexible and can specialise to different hardware, however it suffers runtime code-cache warm-up overhead and has to use runtime



(a) Flow of Janus' binary modification

(b) Overview of Janus' rewrite-rule interpretation

Figure 1. Overview of the Janus binary modification framework.

optimisation to amortise this cost. Therefore some, such as Dynamo [4] and starDBT [23], focus on a specific type of application with a high fraction of hot code, whereas others, such as DynamoRIO [6] and PIN [16], focus on program analysis and instrumentation, where overheads are tolerated. The overarching problem is that these tools lack a global understanding of the applications that they modify.

A combination of static analysis and dynamic binary modification together builds on each other's strengths. Janus is an open source same-ISA dynamic binary modifier augmented with static binary analysis [25]. Janus performs a static analysis of binary executables and determines the transformations required to optimise them. From this, it produces a set of domain-specific rewrite rules, which are steps that the dynamic binary modifier should follow to perform the transformation at runtime. The dynamic component, based on DynamoRIO, reads these rewrite rules and carries out the transformation when it encounters the relevant code.

The Janus framework provides a system for extracting different levels of parallelism within a dynamic binary modifier. Originally developed for automatic parallelisation of binaries (i.e., extracting TLP) [25], this paper augments Janus with techniques to extract DLP and MLP as well. Using a combination of static analysis and dynamic modification, we show the performance of Janus when extracting each of these types of parallelism in isolation and combined. Our evaluation shows that inserting software prefetches alone provides an average speedup of 1.2× and automatic vectorization brings 2.7× Combined, Janus achieves 3.8× on a workload amenable to all forms of parallelism.

2 The Janus Framework

Janus is a same-ISA binary modification framework that was initially developed for automatic parallelisation [25]. It combines static binary analysis and dynamic binary modification, controlled through a domain-specific rewrite schedule.

Figure 1(a) shows an overview of the Janus framework. It starts by analysing an executable statically to identify regions for optimisation, augmenting this with profiling to refine its cost models. It then determines how to modify the code to perform the desired transformations and encodes the steps into rewrite rules, contained within a rewrite schedule. The dynamic binary modifier reads the executable and rewrite schedule and performs specific modifications according to the rules. Janus can also perform profile-guided optimisation, where different rewrite schedule are used to direct the dynamic binary modifier to perform different analysis or modification at each pass.

2.1 Rewrite-Schedule Interface

The rewrite schedule is based on the insight that any complex transformation to a binary can be decomposed into a series of coarse-grained dynamic operations, each of which can be implemented as a fixed package called a *rewrite rule*. The rewrite rules specify isolated modifications to make locally to each dynamic basic block encountered, to produce a global transformation, so that the power of the rewrite schedule is greater than the sum of its parts.

The rewrite rules are persistent, platform independent and are only associated with the corresponding input binary and instruction set architecture. The rewrite schedule can be reused for multiple runs and cumulatively optimised.

2.2 Rewrite-Rule Interpretation

Using a rewrite schedule enables Janus to overcome the limitations of a purely static or dynamic approach. The rewrite schedule controls binary modification and conveys static information to the DBM, removing the need for dynamic program analysis. Yet Janus also builds on the strengths of dynamic binary modification, by specialising code for different hardware, correctly handling signals and faults, and dealing with code that is not discoverable ahead of time.

Figure 1(b) shows the interpretation process of the rewrite rules. To execute application instructions, the baseline DBM first translates them, mangles them if they could cause it to lose control of the running program, then stores them in a code cache. Before the DBM copies each newly discovered basic block to its code cache, it consults the hash table to determine whether there are any rewrite rules associated with the block. If there are, then the DBM invokes the modification handlers dictated by the rewrite rules on the basic block before encoding back to the code cache.

2.3 Janus Applications

Using Janus, we can perform a variety of binary transformations, under control of the static analyser. Janus could also transform the application for program analysis or other optimisation targets, such as security or reliability. The rewrite rule interface can easily be expanded to support new optimisations that are not easily accomplished in a purely static or dynamic system, to further exploit parallelism. The initial Janus framework has already demonstrated its application for extracting thread-level parallelism [25]. In this paper, we show how three kinds of parallelism (TLP, DLP, and MLP) are exploited using a combination of static and dynamic techniques through custom-defined rewrite rules, demonstrating the simplicity and scalability of our approach.

3 Memory-Level Parallelism

Many dynamic binary modifiers focus on dynamic trace optimisations [4, 23] where the frequently executed basic blocks are predicted and concatenated into a trace or superblock to avoid branch stalls and improve pipeline efficiency. However, another significant source of stalls that exists even in native execution affects applications with irregular memory accesses in the trace. These accesses are difficult, if not impossible, to predict in advance, causing high memory-access latencies to fetch data into the cache hierarchy from DRAM.

Prefetching is an effective technique to take advantage of memory-level parallelism by overlapping computation and memory accessing through either hardware or software. Regular stride-based prefetchers are already fully supported by commercial processors. Irregular prefetches can be performed in compilers by inserting extra address-prediction code within memory-bound hot loops to initiate the memory accesses in advance.

`INSTR_CLONE`	Duplicate instructions and insert at a given location
`INSTR_UPDATE`	Update a specified operand of an instruction
`INSTR_NOP`	Insert *n* no-op instructions
`MEM_PREFETCH`	Insert a prefetch for a memory operand and offset
`MEM_SPILL_REG`	Spill a set of registers to private storage
`MEM_RECOVER_REG`	Recover a set of registers from private storage

Figure 2. Major rewrite rules used for prefetching in Janus.

However, software prefetching is not normally added at default compiler optimisation levels (e.g., O3) due to the difficulty in getting them correct. The compiler must balance the trade-off that occurs from generating the prefetches: code must be added to calculate the prefetch addresses but too much creates overhead that swamps the benefits. Moreover, the optimal prefetch distance can vary across different microarchitectures, meaning the compiler must compromise when the target processor is not known.

3.1 Prefetching Approach

We implement the prefetching approach from Ainsworth and Jones [1], originally developed in LLVM to automatically generate software prefetches for indirect memory accesses. The algorithm retrieves and duplicates program slices to calculate prefetch addresses for a given prefetch offset.

We designed six major rewrite rules to support the software prefetching in Janus, as shown in figure 2. Each rewrite rule performs a specific modification to the incoming basic block that will be copied to the code cache. The prime rewrite rule `MEM_PREFETCH` can direct the DBM to insert a software-prefetch instruction at a specified location based on a specific memory operand. An example is shown in figure 3, where the rule `MEM_PREFETCH` directs the DBM to insert the prefetcht0 instruction based on the existing memory access.

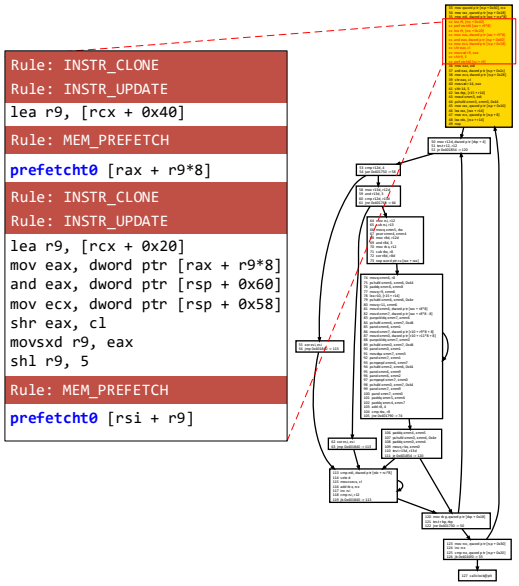
Figure 3(a) shows the rewrite-srule generation pass in Janus' static analyser to enable architecture-dependent rule generation. Janus first scans all the indirect loads in a loop and creates the program slice leading to each address calculation. A prefetch is only generated when the program slice has an input from the induction variable.

Prefetch Address Calculation The prefetched address is a prediction of the dynamic address that will be accessed *n* iterations in the future, where *n* is the prefetch distance. For strided accesses in a hot loop, the prefetch offset can be easily determined as a fixed immediate value. In the DBM, the offset can be directly re-encoded into the prefetch memory operand with a larger displacement value.

However, for indirect memory accesses, variable strides, or complicated address calculations, the address for the *n*th future iteration has to be calculated through an extra code snippet before the prefetch rule. The code snippet can normally be duplicated from existing instructions for the given address in a loop. The `INSTR_CLONE` rule directs the DBM to duplicate the instructions from a range within a basic block.

```
for loop in janus.hotLoops:

  for mem in loop.memReads:
    if isIndirectLoad(mem):
      slice = getPrefetchSlice(mem)
      if slice.inputs.contain(loop.indvars):
        loop.prefetches.insert(slice)

  for slice in loop.prefetches:
    insertRule(INSTR_CLONE, slice)
    insertRule(MEM_PREFETCH, slice.mem, offset)

    for input in slice.inputs:
      insertRule(INSTR_UPDATE, input + offset)

    for mem in slice.memReads:
      insertRule(MEM_PREFETCH, mem, offset*2)

    if slice.needScratchRegister:
      insertRule(MEM_SPILL_REG, slice.scratchSet, loop.init)
      insertRule(MEM_RECOVER_REG, slice.scratchSet, loop.exit)
```

(a) Pseduo-code for prefetch rewrite-rule generation



(b) Example of MEM_PREFETCH interpretation

Figure 3. Prefetch generation in Janus.

The INSTR_UPDATE rule alters the instruction operands so that the slice can take different inputs. A combination of INSTR_CLONE, INSTR_UPDATE and MEM_PREFETCH can then achieve the prefetch for complicated memory accesses.

Static Rule Generation We use the Janus static analyser to detect regions that can be optimised through prefetching within the input binary. We implement the detection analysis in Janus' static analyser based on the algorithm by Ainsworth and Jones [1] and implemented in their LLVM pass. This detection method finds patterns of indirect memory accesses within loops and identifies all instructions that are required for address calculation for each iteration.

Following detection the Janus static analyser generates a combination of INSTR_CLONE and MEM_PREFETCH rules for each opportunity it detects. In some cases, additional scratch registers have to be used for the address calculation. Two rewrite rules, MEM_SPILL_REG and MEM_RECOVER_REG, are generated to direct the dynamic modifier to spill and recover registers determined by the static analyser to be least harmful to performance. With a global view of the loop, the static analyser can detect dead registers and avoid register spilling in the frequently executed code path.

Error Handling The LLVM compiler pass must insert extra bounds-checking code around intermediate loads so as to avoid introducing faults caused by out-of-bounds address calculation. However, Janus reaps the benefits of both static analysis and dynamic transformation, which we use to avoid generating this additional code. After each inserted prefetch instruction we added a specific rewrite rule, INSTR_NOP, to generate a bit pattern which, for x86 code, is 0x90909090

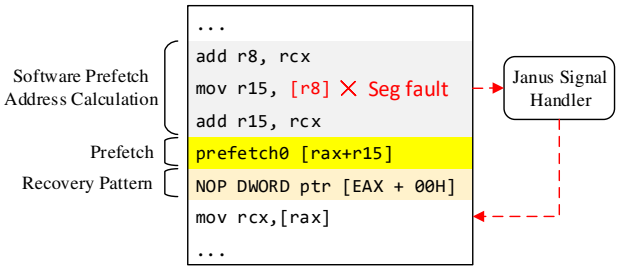


Figure 4. Dynamic error handling in software prefetch.

(four no-ops). If a fault occurs, control is passed to Janus' signal handler which searches ahead in the basic block from the position of the fault, looking for this bit pattern. If it finds the recover pattern, then Janus assumes the fault occurred within code inserted for prefetch optimisation, so it skips to the end of all inserted prefetch code. If there are register spills involved, it rolls back to the start of the sequence of spill-recovery instructions and resumes normal execution. During correct (i.e., fault-free) execution, these no-ops have a negligible effect on performance and avoid the overheads from bounds checking.

Dynamic Adaptation An optimisation is to interpret Janus' rewrite rules only in hot loops, removing cache pollution caused by prefetches in the cold code path. Whenever a trace is formed in DynamoRIO, we can retrieve the rewrite rules and perform software prefetching, rather than doing it for all loops. Our software-prefetch optimisation can also easily adapt to the underlying hardware. For example, if it detects

VECT_BROADCAST	Broadcast a scalar to all SIMD lanes	VECT_BOUND_CHECK	Perform a bound check.
VECT_REDUCE_INIT	Initialise reduction SIMD lanes	VECT_REDUCE_AFTER	Merge reduction SIMD lanes.
VECT_CONVERT	Convert a scalar instruction to SIMD version	VECT_DEP_CHECK	Dynamic dependence check
VECT_LOOP_PEEL	Duplicate the original loop code with specified trip count	VECT_LOOP_UNROLL	Unroll the loop based on the dynamic scale

Figure 5. Major rewrite rules used in automatic vectorisation in Janus.

the processor does not support the prefetch instruction then it avoids generating any prefetch code. This is realised by adding an extra guard in all prefetch-related rewrite rules. With compiler-based prefetch insertion, two copies of the code would have to be generated (one with prefetches, one without) to avoid incurring the overheads of additional instruction execution when running on processors without prefetch support, typically resulting in excessively large executables.

4 Data-Level Parallelism

Many processors support a range of SIMD instruction sets, yet when an application is compiled to run on a variety of different systems, compilers must generate binaries that target the lowest common denominator. This then limits the applications to only run on machines that contain that specific instruction set, making the whole program incompatible with machines containing different instruction sets.

Previous attempts to perform automatic vectorisation in a same-ISA dynamic binary modifier [9, 24] focused on purely dynamic vectorisation in a trace. Due to lack of comprehensive static and alias analysis, this technique can only vectorise loops containing a single basic block and with regular memory accesses. In this section, we describe how generic vectorisation can be achieved using rewrite rules in Janus.

4.1 Binary Vectorisation

In Janus, eight rewrite rules are designed to support vectorisation, as shown figure 5.

Scalar to SIMD Translation The VECT_CONVERT rule is associated with every scalar machine instruction that needs to be translated to a SIMD counterpart. Within the conversion handler, the size of each original scalar register is adjusted to the corresponding widest SIMD register, as determined by the static analyser. For example, register R8 is translated to XMM8 or YMM8 based on the specification of the running hardware, as demonstrated in figure 6(b).

The size of the designated memory operands are also converted to the corresponding size in the instruction. A check of the instruction's opcode is then performed and the appropriate instruction handler called, dealing with either single or double precision, and rewriting to its corresponding opcode and operands for the vector counterpart.

Loop Unrolling and Peeling Loop unrolling is directed by rule VECT_LOOP_UNROLL, which is associated with instructions that update the induction variable in a loop. The static rewrite rule describes the original word width. stride length and number of SIMD lanes for the input binary. Dynamically, Janus obtains the runtime word width from the underlying hardware, calculates the peeling distance and unroll scale, and adjusts memory operands.

However, both the SIMD width and loop trip count might only be known at runtime. Dealing with this requires Janus to calculate the remainder of the loop iterations before the loop's execution, where the rule VECT_LOOP_PEEL is inserted. If the dynamic trip count is not divisible by the lane count, the handler creates a new code cache that duplicates the original loop and modifies the loop bound based on the peeling distance. If the peeling distance is an immediate value, Janus inlines the code into the basic block before the loop execution. Otherwise, if the distance is symbolic, it inserts a runtime check to determine whether to jump to the peeling code cache or not.

Static and Dynamic Alignment Check Janus provides information in the rewrite schedule, gained from the static analyser, on whether the memory accesses are definitely aligned, definitely unaligned, or whether a check must be carried out when the vector size is known. The first two options correspond to when an instruction has no memory access and when unalignment must be assumed given the lack of information available statically. The instruction opcode is then replaced by the appropriate aligned or unaligned, single or double, original- or VEX-encoded instruction, for example VMOVDQA or VMOVDQU.

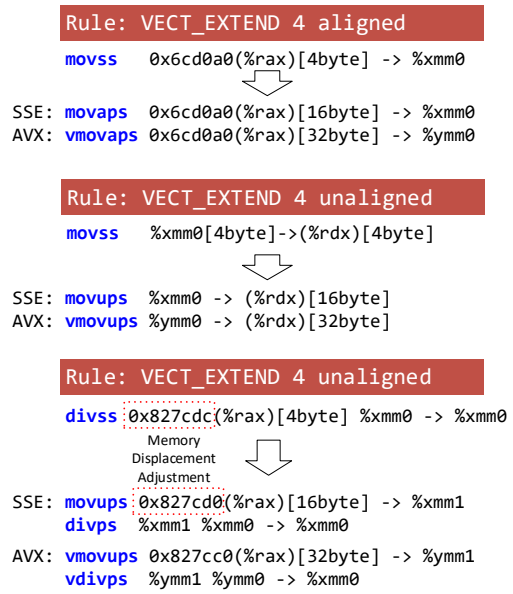
Alignment of memory accesses with the vector size is an important consideration, as loading unaligned memory into vector registers is slow, and arithmetic vector instructions require their memory accesses to be aligned. In order to deal with this, where possible, the initial memory-access address is calculated and stored alongside the VECT_CONVERT rule to allow the calculation of whether or not the instruction is aligned, which will be performed dynamically once the vector size is known.

An arithmetic vector instruction cannot perform a memory access that is not aligned with the vector size. This means that an additional instruction must be inserted to load unaligned memory before performing the operation. The memory operand is loaded into a free register, which then replaces the memory operand in the vector instruction.

Initialisation and Reduction The VECT_BROADCAST rule tags a scalar register, memory operand, or existing SIMD operand to be expanded to a full SIMD register. Depending

```
for loop in janus.hotLoops:
  checks = getRuntimeCheckPairs(loop)
  aligned = alignmentAnalysis(loop)
  for pair in checks:
    insertRule(VECT_BOUND_CHECK, pair, loop.init)

  if needPrePeeling(loop):
    insertRule(VECT_LOOP_PEEL, loop.peel, loop.init)

  for var in loop.inputVariables:
    insertRule(VECT_BROADCAST, var, loop.init)

  for var in loop.reductionVariables:
    insertRule(VECT_REDUCE_INIT, var, loop.init)
    insertRule(VECT_REDUCE_MERGE, var, loop.exit)

  for inst in loop.instructions:
    if needConvert(inst):
      insertRule(VECT_CONVERT, inst, aligned, stride)

  for iter in loop.iterators:
    insertRule(VECT_LOOP_UNROLL, iter, iter.update)

  if needPostPeeling(loop):
    insertRule(VECT_LOOP_PEEL, loop.peel, loop.exit)
```

(a) Pseudo code for vectorisation static rule generation



(b) Runtime interpretation of VECT_CONVERT

Figure 6. Automatic vectorisation in Janus.

on the ISA supported, different broadcast instructions are generated. For zero initialisation, XORing with itself provides a quick solution.

The VECT_REDUCE_MERGE and VECT_REDUCE_INIT rules are partnered when handling reduction variables. The first, VECT_REDUCE_INIT generates the correct initial value in the reduction register. Based on the dynamic lane count and ISA supported, VECT_REDUCE_MERGE generates different code to reduce the multiple reduction variables across the SIMD lanes to one reduction variable in the first lane. Currently only add and multiply reduction is supported.

Runtime Bound Check Once again, the unmodified loop body is required, but this time to provide the original loop in the case that it is deemed unsafe to vectorise at runtime. A VECT_BOUND_CHECK check can be performed on a register's value using the condition of it being equal to a provided value, or on it being a positive value. The check to be used is determined by whether the rule data contains a value to be compared against. If the condition is met then the vectorised loop is executed, otherwise, the original loop is executed. The loop body is bookmarked by the compare instruction and conditional jump over the loop body before it, as well as an unconditional jump to the to-be-vectorised loop's compare instruction following it. The compare instruction jumped to will produce a result that will cause the subsequent vectorised loop's jump to the start of the loop to be ignored, as the vectorised loop ends in the same state as the original.

Runtime Symbolic Resolution The VECT_DEP_CHECK is inserted whenever two memory accesses are identified as "may-alias" due to ambiguities caused by inputs or function arguments. For example, consider this loop:

```
1 for (int i = 0; i < LEN; i++) {
2     a[i] = a[i+k] + b[i];
3 }
```

If the value of k is greater than zero, then Janus can vectorise the loop in a simple manner. For negative values of k, the loop cannot be vectorised without reversing the access order. The rule VECT_DEP_CHECK directs Janus to generate compare and jump instructions before the loop, once the symbolic value of k in a register or stack is available at runtime.

4.2 Vectorisation Rule Generation

Figure 6(a) shows the basic procedure in static rule generation. Loops are initially filtered using Janus' existing static analysis and statically guided profiling using profiling-specific rewrite rules. Loops with function calls, undecided indirect memory accesses, low iteration count, and irregular dependencies are disabled from further rule generation. Janus also has the option to generate profiling rewrite rules to guide DynamoRIO to obtain the hot-loop information in prior runs with training inputs. The resulting rewrite schedule is then only applicable to those hot loops.

For each target hot loop, the static analyser generates rewrite rules by pattern matching the constraints on the static program dependence graph. Induction, reduction and

iterator variables are identified and their dependency and consistency are handled by their corresponding set of rewrite rules. Runtime-check rewrite rules are only generated when there are static ambiguities and the checks can be addressed by simple runtime symbolic resolution. If the symbolic expression is too complicated (e.g., contains phi node or expression too long) or there are an excessive number of runtime checks, Janus would disable vectorisation for this loop.

5 Hybrid Parallelism in Janus

Thread-level parallelism has been explored in Janus in prior work [25] using a set of twelve rewrite rules to perform automatic parallelisation for DoAll loops. In this section, we explore the possibilities for mixing the three sets of rewrite rules to enable the extraction of MLP, DLP, and TLP within the same rewrite schedule.

Modular Rule Interpretation The principle of Janus' modification is to decompose the extraction of each form of parallelism into different sets of fine-grained dynamic operations (rewrite rules), where each set is designed to be self-contained. In Janus, three separate rewrite-rule-generation analyses are performed on a binary for prefetching (MLP), vectorisation (DLP), and parallelisation (TLP), while the final rewrite schedule contains the combination of the three. Interleaving rewrite rules for different optimisations corresponds to the combination of all transformations because all rewrite rules are modular.

Parallelism Applicability Generally the three kinds of parallelism correspond to different phases of computation and therefore different loops. Janus profiles loops to find those that are hot and then selects the kinds of parallelism to extract based on their characteristics. Prefetching is applicable to loops with indirect memory accesses. We avoid loops with only regular strided accesses, assuming that the hardware prefetcher will easily pick up the access pattern. Vectorisation is suited to loops with regular memory accesses and sufficient repeated computation. Parallelisation is for loops without cross-iteration data dependencies and can be applied to loops that also contain MLP or DLP.

Ordered Rule Interpretation Based on the semantics of the rewrite rules, we divide Janus' rewrite rules into three groups:

Atomic performs a self-contained modification that does not affect the prerequisite of later rewrite rules. For example, a runtime bounds check or a scalar-to-vector conversion.

Pairwise rules must be interpreted in pairs. For example, rewrite rules for spilling registers and recovering registers.

Local rewrite rules affect the semantics of later rules, such as loop peeling and unrolling. Therefore rewrite rules have to be interpreted in a specific order.

However, there are a fraction of rewrite rules that need to be handled specifically whenever multiple rewrite rules annotate the same instruction in the same basic block. We carefully define the rewrite-rule interpretation order by setting the priority of the rule so that the output of the previous rewrite rule remains consistent with the prerequisites of the later rewrite rule. For example, the rewrite rule `VECT_LOOP_PEEL` peels a fraction of the total loop iterations to perform aligned vectorisation. Meanwhile, the parallelisation rewrite rule `PARA_LOOP_INIT` modifies the loop starting context for each thread in its thread-private code caches. The vectorisation rule `VECT_LOOP_PEEL` must be interpreted after the parallelisation initialisation rule `PARA_LOOP_INIT` so that the peeling is carried out on the per-thread private version of the loop context. Similarly, the `VECT_REDUCE_AFTER` rule merges a reduction variable into one thread-private copy. This must be executed before the parallelisation rule `PARA_LOOP_FINISH` so that all threaded copies are merged into the main thread.

However there are cases when two rewrite rules exhibit an accumulated modification, where the order of modification does not matter. For example, the vector loop-unrolling rule `VECT_LOOP_UNROLL` can be interleaved with the parallel rule `PARA_LOOP_UNROLL` in any order.

Modification Bookkeeping There are rewrite rules that must be executed in pairs, for example, a spill must be paired with a recover. The effect of the free register lasts between the two rewrite rules. Therefore Janus maintains a runtime bookkeeping variable to keep track of changes made by pairwise rules. For example, all rewrite-rule interpreters must check whether the last-spilled register conflicts with their prerequisites. If a conflict does occur, the handler can regenerate a recover and spill rewrite-rule pair before and after the interrupted rewrite rule. So if a rewrite rule wants to use the original value in register rax that was spilled by a previous rule, it needs to recover the value first.

Refreshable Rule Interpretation DynamoRIO's code cache may occasionally be flushed to maximise the dispatch efficiency in its hash-table lookup. This requires the Janus rewrite-rule handlers to generate the same modifications after code flushing when they encounter the code once more. It requires the rule handlers to examine whether the runtime context it used for just-in-time recompilation has changed. To ensure safety, a better strategy for the handlers is to encode the absolute address of the runtime variable instead of the absolute value.

Summary In Janus, we designed six rewrite rules for software prefetching and a further eight rewrite rules to enable vectorisation in a dynamic binary modifier. These, together with the twelve rewrite rules for parallelisation, constitute the method of using static analysis to enable runtime exploitation of three types of parallelism.

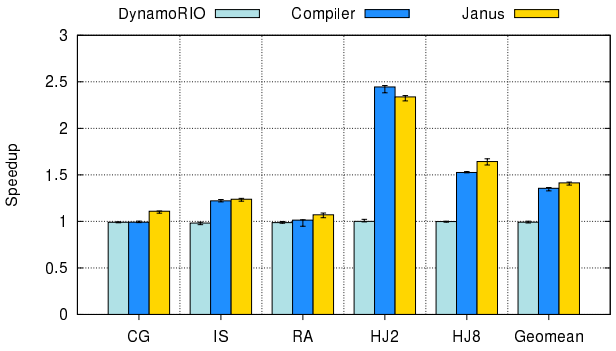


Figure 7. Software prefetch performance comparison between Janus and pre-compiled executables.

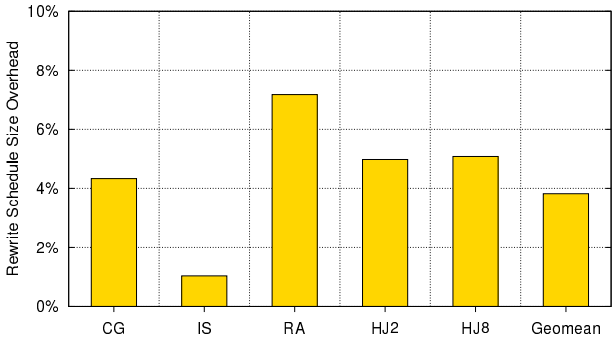


Figure 8. Size of the rewrite schedule for each executable when prefetching, normalised to the size of the executable.

6 Evaluation

We evaluated the extraction of three kinds of parallelism in Janus on an Intel Haswell i5-4570 CPU running Ubuntu 16.04 that contains four cores (8 threads), a 6MB L3 cache and runs at a frequency of 3.2GHz.

6.1 Prefetching

To evaluate the performance of software prefetching in Janus, we evaluated the same benchmarks from the NAS suite [3] as Ainsworth and Jones [1] by compiling a set of binaries using their compiler software-prefetching pass and another without. The baseline binary was compiled by clang 3.8 with optimisation level -O3 (Ainsworth's prefetching pass is written for LLVM). Our evaluation executes Janus on the same binary with a rewrite schedule for prefetching. We report the median, minimum and maximum time from ten runs using the same inputs as published work [1].

Figure 7 shows the performance improvement of just-in-time prefetch in Janus normalised to the baseline of native execution without prefetching. The bar labeled "DynamoRIO" refers to running the original executable under DynamoRIO without performing any modification, reflecting the overhead of the dynamic modifier, which is negligible for the benchmarks. The bar labeled "Compiler" shows the performance of the applications when software-prefetch instructions are added by the LLVM pass. Finally, the bar labeled "Janus" shows that execution using Janus to insert prefetch instructions achieves a significant performance improvement that is comparable to what compilers can extract.

Janus-based prefetching performs similarly to the LLVM pass for IS, and out-performs the compiler scheme for CG, RA and HJ8. There are two reasons for this. First, the hot loop in the binary without prefetching is unrolled, giving better performance, whereas in the binary with compiler-generated prefetching the insertion of prefetches limits unrolling. This shows the benefits of performing certain optimisations, such as prefetching, dynamically at runtime. Second, the no-ops used for error checking in Janus are more optimal than the bounds checking inserted by the compiler.

For HJ2, the LLVM pass achieves higher performance. Janus' modifications for HJ2 require an additional scratch register to be used. Although the amount of spilling required is low, due to inserting spilling only when required, this leads to worse performance than the compiler, since the compiler has full control over register allocation.

Figure 8 shows the size of the rewrite schedule for each application to encode the rewrite rules for prefetching, normalised to the size of the corresponding binary. It is clear that the rewrite schedules are small, around 4%, being at most 7% of the size of the input executable.

6.2 Automatic Vectorisation

To evaluate the performance of automatic vectorisation in Janus, we use the *Test Suite for Vectorising Compilers* (TSVC) benchmarks [7, 18]. The baseline binaries were compiled by gcc 5.4 with optimisation level -O3 and -fno-tree-vectorize to disable auto-vectorisation. The reference binaries were generated with -O3 with auto-vectorisation enabled by default.

6.2.1 Vectorisation Coverage

TSVC consists of 151 benchmarks, each containing a single loop, 28 of which are not vectorisable due to their designated dependence patterns. Each benchmark produces a checksum, which may lose precision if the manipulation of values required results in more frequent rounding or prevents the underlying use of greater precision. Janus proves itself safe and correct by rejecting all loops that it cannot handle within the TSVC test suite and producing checksums which never lose more accuracy than gcc's vectorisation. The double-precision benchmarks never lose any precision.

gcc manages to vectorise and produce a speedup of at least 5% for 58 and 55 loops for single- and double-precision respectively, when using AVX, while Janus is able to speed-up 39 and 33. For single precision, that corresponds to 47.2% and 31.7% of the 123 vectorisable loops in TSVC for gcc and Janus respectively, as shown in figure 9.

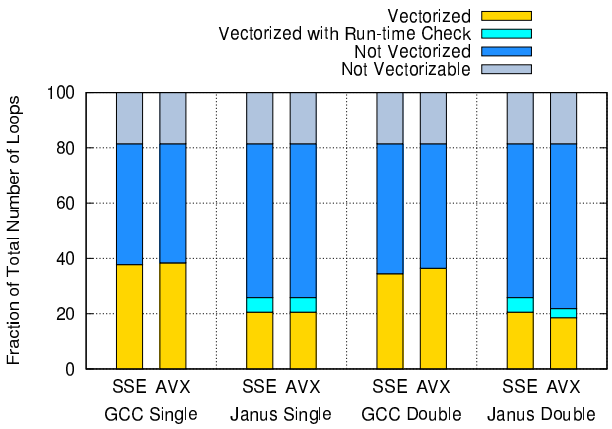


Figure 9. Number of vectorised loop compared to gcc 5.4 auto vectorisation

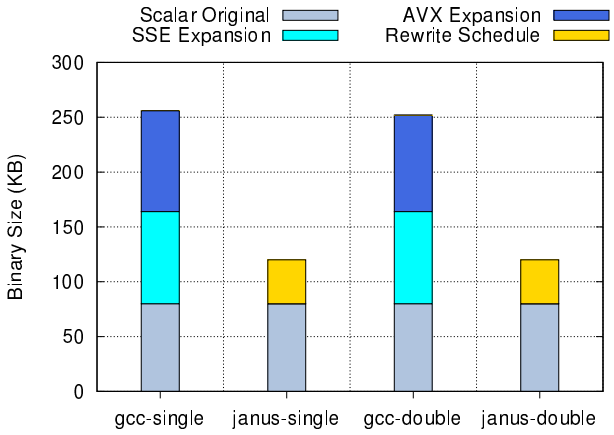


Figure 10. Average size of Janus' rewrite schedule compared to gcc's binary size

6.2.2 Static Analysis Disparity

Janus' static binary analysis rejects 26 loops in TSVC that gcc is able to vectorise. The lack of symbolic information in Janus' memory-related analysis constitutes the reason for the disparity compared to the compiler. These are broken down in table 1.

Undecided memory accesses are those that do not fit an affine expression, which is required for dependence checking, hence Janus is not able to verify the dependence relation with other memory accesses. The discrepancy comes from Janus' failure to perform a full alias analysis at the binary level. Additional barriers in alias analysis are caused by the compiler code generators that use more indirect accesses and other architecture-specific optimisations, thwarting our current analysis.

Highly optimised binaries also contain code from loop unrolling, interchange, and using machine-specific optimisations such as conditional instructions and pointer arithmetic

Table 1. Major reasons in for rejecting loop during vectorisation in Janus.

Reason for rejection	Loop count
Loop trip count not immediate nor symbolic	6
Undecided memory access	5
Incompatible instructions	5
Decrementing induction variable	4
Complex control flow in loop	2
Induction variables with different strides	2
Memory dependency	2

in multi-dimensional array accesses. These either create complicated control flow or cause the induction variable to be updated with different strides in a loop. The strength reduction optimisation also introduces additional cross-iteration dependencies. Although these can be handled by Janus in most case, it complicates the analysis if the reduction is spilled to heap memory. The rest of the reasons for rejection can be resolved by continuing efforts in implementation in Janus' static analysis and the corresponding dynamic handlers.

6.2.3 Storage Benefit

The advantage of Janus is that vectorisation can be tailored to different machines using only a single rewrite schedule without modifying the original binary or specifically targeting either ISA. This is unlike gcc, which would require multiple separate binaries or multiple runtime versions to be produced in order to cover all possible cases. This dynamic adaption of the binary along with runtime checks clearly displays the advantages of Janus over static compilation or modification.

Moreover, the vectorised executables generated by gcc are typically larger than their scalar counterparts for x86 binaries. The storage advantage of only requiring an additional rewrite schedule is shown in figure 10. The size of the rewrite schedule required to achieve similar performance as gcc uses only up to half of the vectorised executable size.

6.2.4 Vectorisation Performance

Figure 11 shows the performance of TSVC aligned loops that are amenable to Janus' vectorisation. The baseline binary runs natively and the execution time for Janus refers to the execution time to run dynamic binary modification on the same baseline binary. We force Janus to vectorise this binary using either the SSE instruction set (128-bit SIMD lanes) or the AVX instruction set (256-bit SIMD lanes).

Two reference binaries were also compiled natively by gcc 5.4 using additional flags -msse4.2 and -mavx, representing state-of-the-art auto-vectorisation performance. From figure 11, we can conclude that Janus is able to produce execution times comparable to the vectorised versions of the binaries produced by gcc. Using SSE, Janus attains 98.7% of gcc's performance where both can vectorise the loop. For

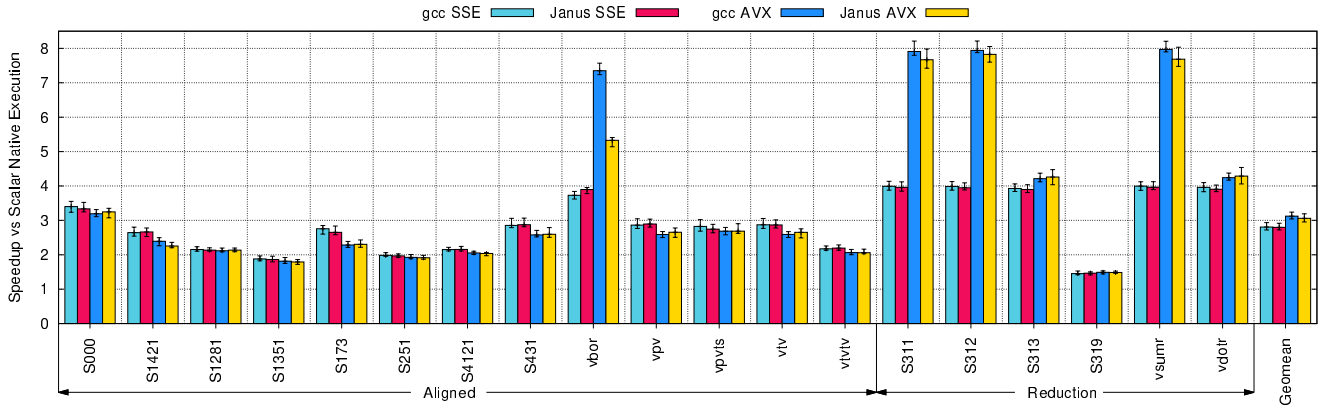


Figure 11. Performance of vectorised aligned loops in single precision TSVC workloads.

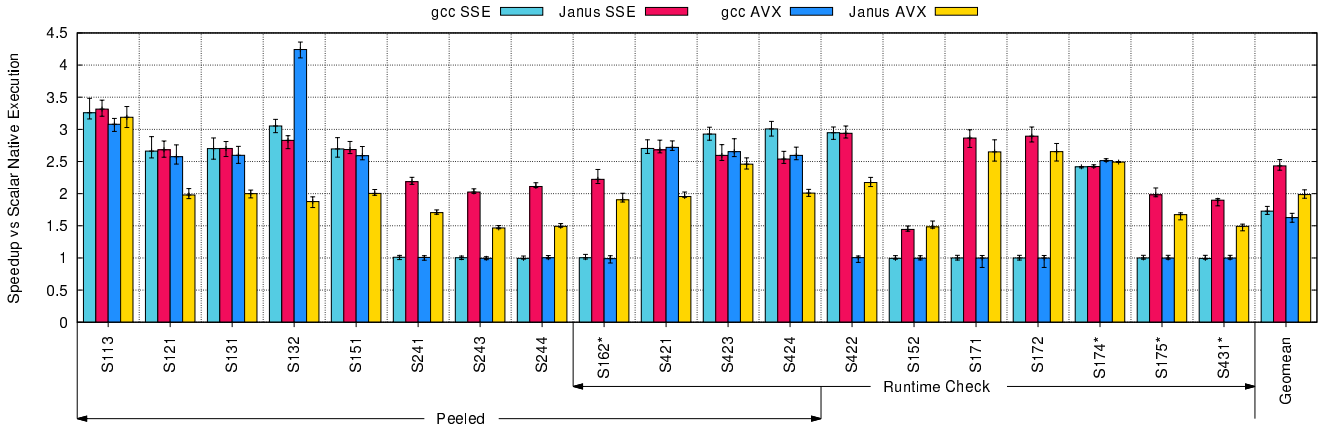


Figure 12. Performance of vectorised peeled and runtime-checked loops in single precision TSVC workloads. Loops marked with * means dynamic symbolic resolution check.

AVX, Janus gains 93.1% of gcc's performance. We divide the benchmarks into four sections and analyse separately.

Aligned The first cluster of loops contain aligned accesses where the iteration count is divisible by the number of SIMD lanes and all memory accesses are aligned. In this regard, Janus achieves almost comparable performance to gcc. One exception is vbor because its loop contains a relatively low iteration count. The short running time does not amortise the sampling overhead in DynamoRIO trace creation and code-cache warm-up.

Peeled This group of loops exhibit unaligned memory accesses or the loop trip count is not divisible by the number of SIMD lanes. Eight out of the nine loops that have a significantly worse running time for Janus contain unaligned accesses and are peeled. The peeling overhead is caused by the Janus auxiliary control to maintain peeling correctness. When duplicating the peeled loop code, additional registers might be used to maintain the peeling and distance.

The performance penalty in AVX is also due to the mixture of SSE and AVX instructions in the generated code. Janus only performs AVX extension on the loops that are annotated by the rewrite rules. The remaining code still contains SSE instructions, used for scalar floating point computation. This causes frequent transitioning between 256-bit AVX instructions and SSE instructions within the same binary, causing performance penalties because the hardware must save and restore the upper 128 bits of the ymm registers internally. Even with no SSE instructions used, DynamoRIO's internal dispatch routine still spills SSE registers instead of AVX registers during its context switch.

Preloaded Vectorisation Janus also vectorises three other loops that gcc cannot: S241, S243 and S244. Based on the gcc report, it identifies memory dependencies whereby in the IR a read after write dependency exists.

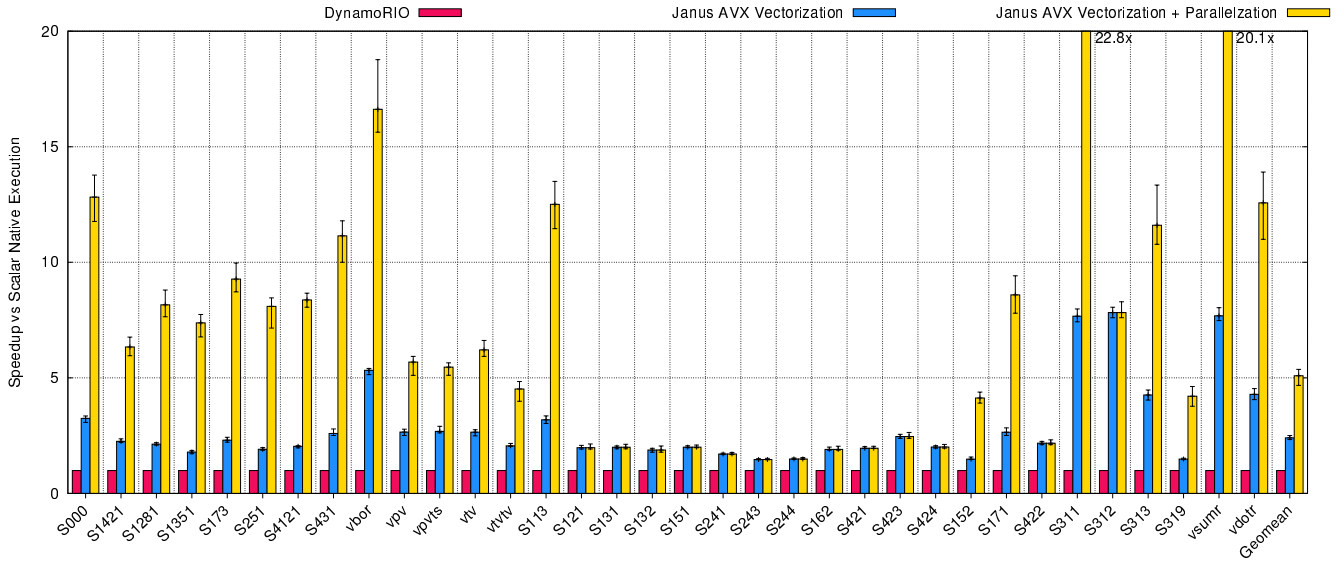


Figure 13. Performance of vectorised and parallelised (4 threads) TSVC workloads.

```
1 for (int i = 0; i < LEN-1; i++) {
2     a[i] = b[i] * c[i  ] * d[i];
3     b[i] = a[i] * a[i+1] * d[i];
4 }
```

At the binary level, the gcc optimisers load the values of a[i] and a[i+1] into registers, essentially pre-loading the read values. Therefore the dependence is not present at the binary level, meaning Janus can vectorise straight away. However, in the compiler analysis, the loop memory access is still considered a memory dependency pattern.

Runtime Check Janus can also insert runtime bound checks to enable vectorisation for loops from S162 to S422. This benefits Janus, enabling it to vectorise six more loops than gcc as well as overcoming the static ambiguity, preventing information loss at binary level.

Reduction The reduction performance achieved by Janus is the same as gcc as can seen from S311 to vdotr.

6.3 Hybrid Parallelisation

We next turn out attention to combining parallelism extraction in Janus.

DLP+TLP Automatic parallelisation and vectorisation can be combined to achieve significant performance, as shown in figure 13. The baseline is the native execution of the scalar binaries compiled by gcc 5.4. The parallelisation analysis is applied to all loops but only vectorised loops are shown in figure 13. Twelve loops are not parallelised due to cross-iteration dependencies found during static analysis. For vectorisation, forward cross-iteration dependencies are allowed using preloaded optimisation. In contrast, a DoAll parallelisation requires no cross-iteration dependencies in all cases.

For the remaining 22 loops, the hybrid of vectorisation and parallelisation achieves a speedup of 8.8× on average that further boosts performance from 2× using vectorisation alone.

DLP+TLP+MLP There are few opportunities to extract all three types of parallelism from a single loop. This is because the prefetch optimisation normally requires indirect memory accesses that could challenge the dependence analysis to enable parallelisation and vectorisation.

Loop S4112 from TSVC shows one such example, where this pattern is commonly found in sparse matrix multiplication (SpMV) applications.

```
1 for (int i = 0; i < LEN; i++) {
2     a[i] += b[ip[i]] * s;
3 }
```

The indirect array accesses b[ip[i]] causes the loop to be memory bound and it can be optimised by software prefetch. Meanwhile, vectorisation is also possible if the hardware supports the gather instruction within its SIMD ISA. The indirect accesses are only reads so there are no cross-iteration write dependencies, meaning parallelisation is also safe.

Since DynamoRIO does not support the AVX512 gather instruction (and the AVX2 gather does not give good performance), we could not evaluate this loop in Janus automatically. Instead we manually wrote an assembly version and encoded the loop into raw binary code snippets. The code snippets were then loaded into Janus' code cache for dynamic execution. For parallelisation, they are copied to each thread's private code cache.

Figure 14 shows that prefetching, vectorisation, and parallelisation (on 4 threads) by themselves achieve 1.16×, 1.21×

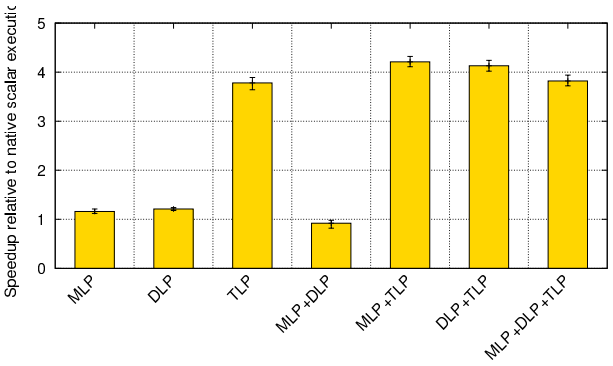


Figure 14. Performance of s4112 with different combinations of parallelism.

and 3.78× speedups respectively, indicating the effectiveness of the transformations made by Janus. Parallelisation achieves the largest speedup due to the embarrassingly DoAll parallelism. The combination of prefetching and parallelisation can further improve the performance by 12% from the 3.78× speedup, due to these transformations being orthogonal. In effect, prefetching boosts performance by 12% whether used alone or in combination with parallelisation. Similarly, the combination of vectorisation and parallelisation can further improve the parallelisation performance by 9%, where these transformations both exploit parallelism across loop iterations.

After prefetch optimisation, the loop becomes compute bound whereas after vectorisation, the loop becomes memory bound. However, combining prefetching and vectorisation results in slowdown. This is due to the parallelism provided by a vector memory load being swamped by the extra overhead of inserting duplicated address calculation for prefetching. This problem could be addressed by having a vector version of a prefetch gather instruction, something that is only available in the Xeon Phi ISA VGATHERPF0. This also impacts the overall overhead of the combination of the three forms of parallelism, achieving only 3.8× speedup.

6.4 Summary

Thanks to the combination of static analysis, profiling, and runtime checks seamlessly controlled by 26 rewrite rules, we can achieve substantial performance through the Janus triad: MLP, DLP and TLP. Extracting most of these forms of parallelism achieves similar performance to a compiler approach with an abundance of source code information. Combining them allows extraction of three forms of parallelism simultaneously from within a binary.

7 Related Work

Janus is the first system to propose a unified platform for extracting three kinds of parallelism using a DBM. We organise the related work based on the individual types of parallelism extracted.

Automatic Binary Prefetching ADORE [15] uses Itanium hardware counters to identify hotspots and phases and to apply memory prefetching optimisations. However, ADORE relies on the compiler reserving a scratch register that is needed for the inserted address calculation. Beyler et al. [5] is a pure software approach that uses a helper thread to monitor load states and insert prefetch when needed.

In contrast, Janus works on any binary, with no assumptions made about the way it has been compiled with no helper thread. Prefetching in Janus is directed by rewrite rules, obtained through static analysis and preliminary profiling. Scratch registers are spilled outside the loop, hence minimising the spilling overhead. Janus only performs modification to indirect accesses, assuming that direct array accesses are handled by hardware prefetchers.

Automatic Binary Vectorisation Hallou et al. [9] implemented auto-vectorisation in a same-ISA dynamic binary translator (DBT) called Padrone [21], which represents the closest work to our loop vectorisation in Janus. They integrated a lightweight static analysis within the DBT to perform vectorisation whenever hot code is identified. However, their system cannot handle complicated control-flow modification, such as loop peeling, and they can only work on loops with a single basic block. Without offloading static analysis from runtime, they also suffer a significant overhead from performing time-consuming analysis (e.g., alias analysis) during program execution. Janus does not suffer these overheads, thanks to the rewrite schedule that communicates statically derived transformations to the dynamic modifier.

Hong et al. [11, 13] proposed using a machine-independent IR layer to achieve cross-ISA SIMD transformation implemented in QEMU. Li et al. [12] implemented vectorisation from x86 to Itanium architecture. Vapor SIMD [19] used a custom designed IR derived source code and perform JIT compilation. Their target is cross-ISA translation, a different domain to Janus. Since it is cross-ISA translation, they get lower performance compared to Janus.

Yardımcı and Franz [24] proposed a binary parallelisation and vectorisation scheme for PowerPC binaries, which combines static analysis and dynamic binary parallelisation in their dynamic software layer. However, they do not fully capitalise of the strengths of combining static and dynamic components, such as runtime checks, error handling, and trace optimisation. Other work has studied the upper limit on the parallelism extractable through a dynamic binary translator [8], but failed to consider the parallelism freed after removing apparent data dependencies.

Dynamic Binary Optimisation DynamoRIO [6] is a robust open-source runtime code manipulation system which originates from the well-known high-performance binary translator, Dynamo [4]. Other dynamic modification tools, such as Pin [16], are closed source, and, like DynInst [10],

are more focused on binary instrumentation. The Sun Studio Binary Code Optimiser [14] and Microsoft Vulcan [22] are well-known commercial tools for rewriting binaries for better single-threaded performance, but both rely on instrumentation to collect profiling information.

8 Conclusion

We have presented the Janus triad, a unified approach for exploiting three kinds of parallelism seamlessly controlled by domain-specific rewrite rules in a same-ISA dynamic binary modifier. Rule generation and interpretation enables easy automation and portability. Through the Janus triad, we demonstrate substantial performance from this framework comparable to compiler counterparts.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through grant references EP/K026399/1, EP/P020011/1 and EP/N509620/1. Additional data related to this publication is available in the data repository at https://doi.org/10.17863/CAM.37523 and Janus can be obtained at https://github.com/JanusDBM/Janus.

References

[1] Sam Ainsworth and Timothy M. Jones. 2017. Software Prefetching for Indirect Memory Accesses. In *CGO*.

[2] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. 2010. *Decompilation to compiler high IR in a binary rewriter*. Technical Report. University of Maryland.

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks summary and preliminary results. In *SC*.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. In *PLDI*.

[5] Jean Christophe Beyler and Philippe Clauss. 2007. Performance driven data cache prefetching in a dynamic software optimization system. In *SC*.

[6] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *CGO*.

[7] David Callahan, Jack Dongarra, and David Levine. 1988. Vectorizing compilers: A test suite and results. In *SC*.

[8] Tobias J. K. Edler von Koch and Björn Franke. 2013. Limits of Region-based Dynamic Binary Parallelization. In *VEE*.

[9] Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. 2015. Dynamic re-vectorization of binary code. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*.

[10] Jeffrey K Hollingsworth, Barton Paul Miller, and Jon Cargille. 1994. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference*.

[11] Ding-Yong Hong, Sheng-Yu Fu, Yu-Ping Liu, Jan-Jan Wu, and Wei-Chung Hsu. 2016. Exploiting longer SIMD lanes in dynamic binary translation. In *ICPADS*.

[12] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. 2006. Optimizing dynamic binary translation for SIMD instructions. In *CGO*.

[13] Yu-Ping Liu, Ding-Yong Hong, Jan-Jan Wu, Sheng-Yu Fu, and Wei-Chung Hsu. 2017. Exploiting Asymmetric SIMD Register Configurations in ARM-to-x86 Dynamic Binary Translation. In *PACT*.

[14] Sheldon Lobo. 1999. The Sun Studio Binary Code Optimizer. http://www.oracle.com/technetwork/server-storage/solaris/binopt-136601.html.

[15] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. 2003. The performance of runtime data cache prefetching in a dynamic optimization system. In *MICRO*.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*.

[17] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. 2004. Ispike: A post-link optimizer for the Intel® Itanium® architecture. In *CGO*.

[18] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. 2011. An evaluation of vectorizing compilers. In *PACT*.

[19] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-vectorize once, run everywhere. In *CGO*.

[20] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. (2019).

[21] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. 2014. Padrone: a platform for online profiling, analysis, and optimization. In *International Workshop on Dynamic Compilation Everywhere*.

[22] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. 2001. *Vulcan: Binary Transformation in a Distributed Environment*. Technical Report MSR-TR-2001-50. Microsoft Research.

[23] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. 2007. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*.

[24] Efe Yardımcı and Michael Franz. 2006. Dynamic Parallelization and Mapping of Binary Executables on Hierarchical Platforms. In *CF*.

[25] Ruoyu Zhou and Timothy M. Jones. 2019. Janus: Statically-Driven and Profile-Guided Automatic Dynamic Binary Parallelization. In *CGO*.