View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Apollo

Janus: Statically-Driven and Profile-Guided Automatic Dynamic Binary Parallelisation

Ruoyu Zhou
University of Cambridge, UK
ruoyu.zhou@cl.cam.ac.uk

Timothy M. Jones
University of Cambridge, UK
timothy.jones@cl.cam.ac.uk

***Abstract*—We present Janus, a framework that addresses the challenge of automatic binary parallelisation. Janus uses same-ISA dynamic binary modification to optimise application binaries, controlled by static analysis with judicious use of software speculation and runtime checks that ensure the safety of the optimisations. A static binary analyser first examines a binary executable, to determine the loops that are amenable to parallelisation and the transformations required. These are encoded as a series of rewrite rules, the steps needed to convert a serial loop into parallel form. The Janus dynamic binary modifier reads both the original executable and rewrite rules and carries out the transformations on a per-basic-block level just-in-time before execution. Lifting static analysis out of the runtime enables the global and profile-guided views of the application; ambiguities from static binary analysis can in turn be addressed through a combination of dynamic runtime checks and speculation guard against data dependence violations. It allows us to parallelise even those loops containing dynamically discovered code. We demonstrate Janus by parallelising a range of optimised SPEC CPU 2006 benchmarks, achieving average speedups of 2.1× and 6.0× in the best case.**

I. Introduction

Program performance is heavily influenced by the ability to create parallel applications consisting of multiple threads working as independently as possible. The programming language and runtime communities have provided new languages and constructs to aid parallel programming, significantly boosting programmer productivity [1]. Whilst important and useful for new applications, users of single-threaded applications where the source code is lost, unavailable or cannot easily be recompiled, are not able to benefit from the underlying parallel architecture.

Within this context, parallelisation of application binaries becomes a seductive proposition. Regardless of the source languages used to create the program, or the availability of the code, an application can be restructured within its binary form to split off tasks into separate threads, and combine their results back together when required. Although almost impossible to perform effectively by hand, automatic tools have the ability to extract parallelism from sequential applications through analysis of the executable to extract multiple threads that can execute concurrently.

Within the literature there are a number of schemes for binary parallelisation, using either static or dynamic approaches. Static binary parallelisers [2], [3], [4], [5] typically focus on affine loops with known iteration counts. They deal with ambiguity from static analysis by creating multiple versions of code and directing execution to a suitable variant based on runtime conditions. However, these produce large executables, limit flexibility, and are difficult to integrate into stripped binaries and combine with exception and signal handling.

At the other end of the spectrum are purely dynamic approaches, such as RASP [6], which is a simulated dynamic parallelisation approach that relies on hardware transactional memory to speculate on the independence of threads, but does not run on existing multicore systems. However, some have suggested that dynamic binary parallelisation alone can never achieve significant performance [7] without using necessary transformation to remove predictable data dependencies.

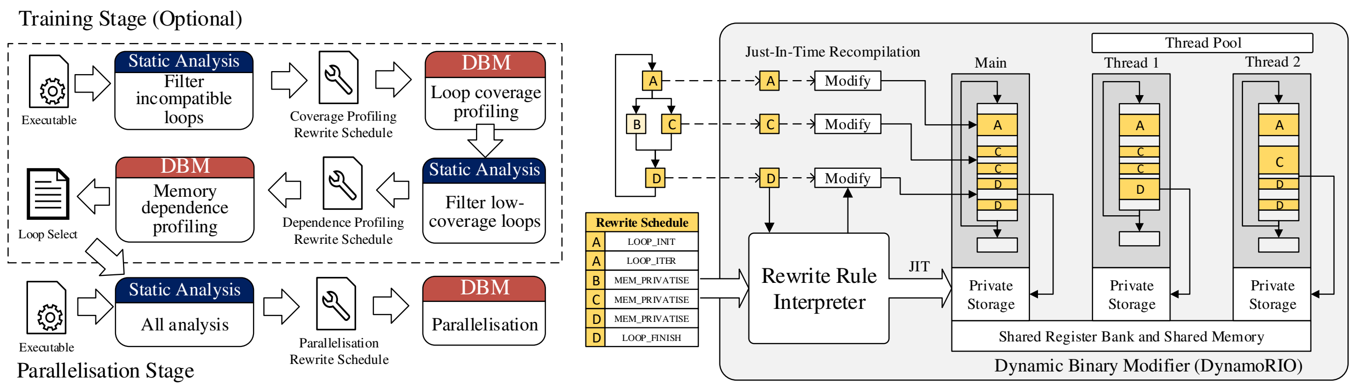
To this end, we present Janus, an open source binary modification framework[1] designed for automatic parallelisation that overcomes the limitations of prior approaches by combining static analysis, profile information, dynamic modification, judicious use of speculation, and runtime checks. Janus uses an intermediate architecture-independent interface between static and dynamic components, called a rewrite schedule, to define, control, and automate binary modification. With the expressive power of the rewrite schedule, Janus is able to perform analysis, profiling, and complicated parallelisation operations. The dynamic binary modifier (DBM), based on DynamoRIO [8], reads the rewrite schedule and carries out parallelisation as instructed at runtime.

Janus supports both x86-64 and AArch64 binaries and requires no user intervention to transform, profile, or optimise applications. Prior techniques contain only some of these features, but not are fully automated, and are therefore restricted practically in the parallelism that they can extract. We evaluate Janus on a x86-64 multicore system, parallelising applications from the SPEC CPU 2006 benchmark suite, achieving speedups of 2.1× on average and 6.0× in the best case when running with eight threads.

II. Janus Binary Parallelisation

Janus parallelises loops from sequential binaries, running groups of iterations on different cores in a round-robin fashion. We currently extract DoAll parallelism because this can already unlock significant performance for some applications.

[1]Janus available at https://github.com/JanusDBM/Janus and data for this publication at https://doi.org/10.17863/CAM.33893



(a) Automatic parallelisation flow (b) Parallel loop execution

Fig. 1. Automatic parallelisation in Janus. Rewrite schedules allow the static analyser to control dynamic modification.

Figure 1(a) shows how rewrite schedules are used by the static analyser to control profiling and parallelisation; figure 1(b) shows how a loop is modified dynamically based on a rewrite schedule, to execute it in parallel.

Janus starts by analysing an executable statically to identify parallelisable loops, and augments this with profiling to find the most profitable loops to parallelise and to identify any cross-iteration memory dependencies that it may contain. It then determines how to parallelise each one and encodes the transformation steps into rewrite rules, contained in a rewrite schedule. The dynamic binary modifier reads the executable and rewrite schedule and performs the parallelisation according to the rules. This includes inserting runtime checks or speculation to deal with potentially unsafe behaviour.

A. *Static-Dynamic Interface*

The rewrite schedule is central to Janus, facilitating communication from the static analyser to the dynamic binary modifier in a way that allows complex transformation and optimisation of the binary application. In effect, it maps the static analyser's global view of the binary down to a series of changes to make at the DBM's local view.

1) Rewrite Schedule: The key insight provided by the rewrite schedule is that complex transformations to a binary can be decomposed into a series of coarse-grained dynamic operations. We define each of these operations to be a *rewrite rule*. Each rewrite rule directs the DBM to perform a specific modification locally to each basic block, and together the rewrite rules make the changes required to implement a global sophisticated optimisation, such as parallelisation. The rewrite schedule and rewrite rules specify simple dynamic transformations that can be easily implemented and verified individually. However considering the combined effect in a runtime context, the power of the rewrite schedule is greater than the sum of all its parts.

Each rewrite schedule contains a header, rewrite rules to control transformations, and data to support them. The header specifies the layout of the rewrite schedule, as well as global and miscellaneous information about the executable. Each rewrite rule is a fixed-length data structure consisting of an address that corresponds to an application location where the rewrite rule should be triggered, a rule ID to describe the transformation to carry out, and a data field that contains rule-specific information (e.g., a register number or immediate).

Using a rewrite schedule enables Janus to overcome the limitations of pure static or dynamic binary modification. The rewrite schedule controls binary modification and conveys static information to the DBM, removing the need for dynamic program analysis. Yet it also builds on the strengths of dynamic binary modification, by allowing Janus to specialise code for each thread, for different hardware, to correctly handle signals and faults, and to deal with code that is not discoverable ahead of time. For example, in the presence of shared library calls, rewrite rules define the boundary between the statically analysable and unknown codes. At runtime, on crossing this boundary, the DBM takes control of deciding which modifications should be made, as it discovers new code to execute, and then hands control back to the rewrite schedule on crossing the boundary again.

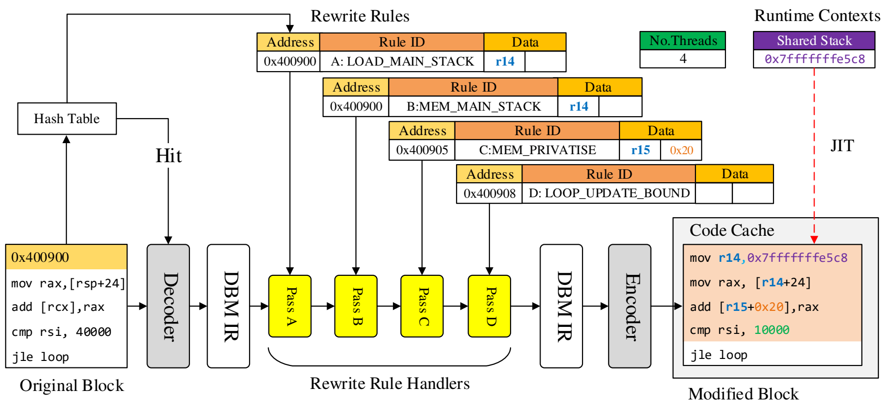
2) Rewrite-Schedule Interpretation: The task of the dynamic binary modifier in Janus is to transform and execute an application under direction from the rewrite schedule provided by the static analyser. Each rewrite rule ID has a corresponding runtime handler within the DBM which is responsible for carrying out the transformation. Each runtime handler is specific to a single rule ID, and understands the meaning of the information carried in the rewrite rule data field. To add more functionality to Janus we simply add new rule IDs to the rewrite schedule and create their corresponding handlers in the DBM.

Figure 2(b) shows an example of the rewrite rule interpretation process. The DBM first takes control of an application at startup and immediately loads its associated rewrite schedule. It inserts each rewrite rule into a hash table, indexed by instruction or basic-block address from the program binary, for fast lookup. To execute application instructions, the baseline DBM (i.e., without considering the rewrite schedule) first translates them, modifies them if they could cause it to lose control of

```
for loop in janus.hotLoops:
  if loop.isDOALL:

    for init in loop.inits:
      insertRule(LOOP_INIT, init)

    for exit in loop.exits:
      insertRule(LOOP_FINISH, exit)

    for bound in loop.bounds:
      insertRule(UPDATE_BOUND, bound)

    for mem in loop.mems:
      if needPrivatise(loop, mem):
        insertRule(MEM_PRIVATISE, mem)
```

Static Binary Analyser

(a) Example of a rewrite rule generation pass



```
mov rax,[rsp+24]
add [rcx],rax
cmp rsi, 40000
jle loop
```

```
mov r14,0x7fffffffe5c8
mov rax, [r14+24]
add [r15+0x20],rax
cmp rsi, 10000
jle loop
```

(b) Interpretation of rewrite rules to privatise variables within a basic block in the DBM

Fig. 2. The rewrite schedule and its interpretation at runtime in the DBM.

the running program, then stores them in a code cache. This process occurs when the DBM encounters instructions it has not seen before, or when it performs trace optimisation on frequently executed code sequences. In Janus, before storing the instructions in the code cache, it checks whether there are any rewrite rules associated with the code and, if so, makes the appropriate transformations.

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, the DBM invokes the corresponding handler to modify the block. In this example, four rewrite rules are found corresponding to the incoming basic block for a thread. The first rewrite rule directs the DBM to insert a new instruction to encode the global shared stack pointer as an immediate value into a register (r14). The second alters the mov instruction to use r14 instead of rsp, essentially loading a read-only variable from the shared stack rather than each per-thread private stack. The third rewrite rule privatises a heap memory operand [rcx] used by the add instruction to the pre-allocated thread-private location [r15+0x20], where r15 contains the base of thread-local storage (TLS). Finally, the fourth rewrite rule alters the loop bounds so that the thread only executes a quarter of the loop iterations. Following this, the modified basic block is encoded to machine code again and immediately executed from the code cache. As shown in figure 2(b), where two or more rewrite rules refer to the same machine instruction, transformations are carried out in the order that they occur in the rewrite schedule, which is defined in the static analyser. The final modified code is immediately executed from the code cache.

B. Rewrite Rules for Automatic Parallelisation

To achieve automatic parallelisation using Janus, we designed six major profiling rewrite rules and twelve rules for parallel transformation, as shown in figure 3. Using these high-level rules, Janus automates the whole parallelisation process. This includes profiling for loop coverage (to select the most profitable loops to parallelise), profiling data dependencies (to identify loops that are likely DOALL, if static analysis cannot prove this), and transforming each loop into parallel form.

C. Statically-Driven Profiling

Janus optionally collects profile information using dynamic binary instrumentation, driven by static analysis through a rewrite schedule. This automates the process of profile gathering in the same DBM and also addresses the shortcomings of a purely dynamic sample-based approach. Moreover, Janus' statically guided profiling is quicker than other binary instrumentation tools because it only instruments the loops of interest and only certain instructions within those loops (e.g., not all loads and stores).

As shown in figure 1(a), each loop is analysed to determine whether it is in a form that is feasible for parallelisation. We reject loops with incompatible instructions, those performing IO operations, those with interrupts, exceptions, system calls or non-returning subroutines, and those where we cannot recognise the induction variables. These loops are not subject to profiling or any further analysis. For feasible loops, we profile with training inputs, counting the number of dynamic instructions executed in each as a proxy for time spent inside each one. Low coverage loops (those with a small fraction of total program instructions) are filtered out. Finally, we perform a further profiling run on all loops that may have cross-iteration data dependencies to identify those that definitely exist and those that may not.

D. Parallelisation Rewrite Schedule Generation

Janus' static analyser reads stripped binary executables as input, disassembles the code section and converts the disassembled instructions into its own IR. Each IR instruction has a one-to-one correspondence with an instruction from the binary's ISA. Although this is low level, it facilitates the generation of rewrite rules in the backend of the tool since it is close to the actual machine code. Janus abstracts all register, stack and absolute memory locations into versioned variables

PROF_LOOP_START	Start profiling a loop.	PROF_LOOP_FINISH	Finish profiling a loop.
PROF_LOOP_ITER	Start another loop iteration.	PROF_EXCALL_START	Start profiling an external call within a profiled loop.
PROF_EXCALL_FINISH	Finish profiling an external call within a profiled loop.	PROF_MEM_ACCESS	Check for data dependencies for a memory access.
THREAD_SCHEDULE	Schedule threads to jump to a code address.	THREAD_YIELD	Send threads back to the thread pool.
LOOP_INIT	Initialize loop context for each thread.	LOOP_FINISH	Combine loop contexts from all threads.
LOOP_UPDATE_BOUND	Update a loop bound for a thread.	MEM_MAIN_STACK	Redirect a stack access to the main stack.
MEM_PRIVATISE	Redirect a memory access to a private address.	MEM_BOUNDS_CHECK	Perform a bounds check on two array bounds.
MEM_SPILL_REG	Spill a set of registers to private storage.	MEM_RECOVER_REG	Recover a set of registers from private storage.
TX_START	Start a software transaction.	TX_FINISH	Validate and commit a software transaction.

Fig. 3. Major rewrite rules used in automatic profiling (blue) and parallelisation (orange) in Janus.

in static single assignment (SSA) form. From here Janus performs standard control-flow and data-flow analysis, including domination, liveness, reaching, dependence and memory-alias analyses.

Loops and function calls are recognised and analysed if the CFG is fully recovered. Each variable and memory address accessed in the loop can be represented as a tree of expressions in the SSA graph. By traversing the abstract tree upwards, we can construct a canonicalised symbolic polynomial where the terms are leaf nodes of the function argument and memory accesses. The loop's iterator is identified by constructing a cyclic expression starting from the phi node of the loop start block in the abstract expression tree. By examining the loop exit conditions, we can solve the range of each loop iterator, symbolically representing it as a start, step and final value of the iterator. The symbolic range of the loop iterator is then propagated back to all the memory accesses within the loop.

Handling optimised binaries: One of the prime difficulties faced by Janus is to parallelise binaries that have been heavily optimised by compilers. Optimised binaries often contain inner loops that have been unrolled, jammed, or had iterations peeled off by the compiler to help with vectorisation. These inner loops may also contain multiple versions of code, with the correct version selected at runtime based on compiler-generated runtime checks. Distinct code paths may compute the same output values using different combinations of operations, which results in unnecessary phi nodes and complicates Janus' dependence analysis. In addition, ISA-specific complex instructions, register spilling, indirect stack accesses, and conditional instructions obfuscate the data-flow graph obtained by Janus.

To alleviate this problem, we evaluate the canonicalised expressions for each phi node in the loop using symbolic execution and range propagation. Therefore if Janus can prove equality for the expressions for all predecessors in the phi node, it flags the path (phi node) as duplicated. For complex instructions, we conservatively simplify them in our analysis-only IR (e.g., for a conditional move, we include both source operands in our analysis).

Alias analysis: Additional alias analysis is performed for the memory reads and writes that belong to the same array base. We calculate the distance vector for every memory read-write and write-write pair and solve the equation when the distance vector is zero. As the range of each pair is already propagated and the bases are the same, a memory alias is detected when the ranges of two pairs overlap. Then we identify all common parts of the polynomial that are considered constant throughout the loop. If there is more than one array base identified, Janus emits a MEM_BOUNDS_CHECK rule at the point in the code when the array base is created (typically at the beginning of the function). This guards the parallel version of the loop, only enabling parallelisation when the checks pass, meaning that all array bases are independent of each other.

Loop characterisation and selection: Based on profiling and static alias analysis, Janus divides the candidate loops into four categories. The fifth category, "incompatible", applies to loops that were never candidates for parallelisation with our current implementation.

Type A: Static DOALL These loops contain no cross-iteration dependencies except through induction and reduction variables with addition and subtraction reduction operations. The number of iterations of the loop may be determined statically and there may be multiple exits from the loop.

Type B: Static Dependence These loops have cross-iteration data dependencies that have been identified statically.

Type C: Dynamic DOALL Here the loop's induction variable can be clearly identified. There are memory accesses that cannot be analysed statically, but profiling shows that they do not alias at runtime.

Type D: Dynamic Dependence This corresponds to the remainder of the loops: those where the induction variable can be clearly identified but there are memory accesses that cause cross-iteration dependencies during profiling.

Loop nests are identified using an inter-procedural control-flow analysis and only one loop in each nest selected for parallelisation. Janus selects the outermost loop of type A and failing that then type C. Within type A we prefer loops where the number of iterations is statically known and there is a single exit from the loop.

Rule generation for selected loops: An example of generating parallelisation rules is shown in figure 2(a). The static analyser abstracts and encapsulates both profiling information and high level objects such as loops and functions. Based on the selected loop, Janus flags all those variables that are "private", "read-only", "first-private", "induction", and "reduction" using rewrite rules, so that each thread can interpret them differently and copy values from the main thread's registers or stack frame.

For the stack accesses that are read-only, Janus emits a MEM_MAIN_STACK rule for each instruction that reads the stack element involved. At runtime, each thread reads the main stack

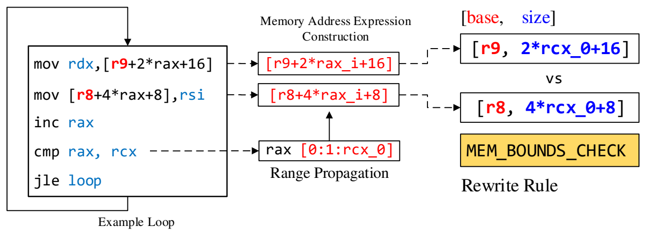


Fig. 4. Generation of MEM_BOUNDS_CHECK based on symbolic range propagation.

instead of its own, for those elements only and avoids copying across stacks. Cross-iteration WAR and WAW dependencies can be removed by privatising the data per thread. To achieve this, Janus emits a MEM_PRIVATISE rule for each instruction that reads or writes these heap locations. The MEM_PRIVATISE rule contains a private storage ID so that on translation the memory access is re-encoded into a direct memory access to a specific private storage location.

E. Parallelised Loop Execution

All loop handling code is generated by the dynamic binary translator just-in-time based on the rewrite schedule, the original code, runtime and machine characteristics. Janus contains two rewrite rules that generate dynamic code to control the creation and deletion of parallel threads. Once threads are created, they wait in a thread pool until required to execute a parallel loop. Two further threading rules, THREAD_SCHEDULE and THREAD_YIELD, direct threads to leave the thread pool and start execution of a specified PC address, then return again afterwards. Once a Janus thread leaves the thread pool, its instructions are controlled by the dynamic binary modifier (meaning thread code is subject to modification through the rewrite schedule). Each thread has associated thread-local storage and a private code cache, as does the main thread, which allows independent interpretation of rewrite rules to specialise computation for each thread. Each thread also obtains its own private stack and copies minimum initial contexts from the main stack just before executing its first iteration.

Janus has the ability to dynamically choose the thread-scheduling policy. By default, if the number of loop iterations can be determined statically or determined through a runtime check just before the loop, each thread runs an equal number of contiguous iterations corresponding to $\#iterations/\#threads$. If the number of loop iterations cannot be determined dynamically, Janus schedules threads to execute a small number of contiguous iterations from the total iteration space in a round-robin fashion.

1) Runtime Array-Base Check: We implement a dynamic check rule to catch dependencies between loop iterations where static analysis cannot prove their absence. These occur before the loop runs, and verify that all arrays written to are distinct from those reads.

Although our static analysis can often determine that accesses to two arrays do not alias if the arrays are different, it cannot always prove the second step and guarantee that the arrays are indeed separate objects. Our dynamic check alleviates this issue by performing the check at runtime before the loop starts execution. Figure 4 shows an example of emitting a MEM_BOUNDS_CHECK rule. Static analysis can identify the base of the array (usually held in a register or on the stack) and the maximum address accessed (calculated from the induction variable and any offset with knowledge of the loop's iteration count). Our runtime check therefore verifies that arrays that are written to are entirely independent of other arrays that are read from or written to. In the example, this means checking there are no overlaps between the ranges [r9, r9+2*rcx_0+16] and [r8, r8+4*rcx_0+8].

The MEM_BOUNDS_CHECK rule is inserted at the least-executed path before the loop execution where the inputs are available, such as beginning of the loop's parent function. It ensures that parallel execution only proceeds if the checks all pass. If not, then the loop is executed sequentially. If the loop is already modified, Janus would flush the modified code cache and reload the original sequential code.

2) Just-In-Time Software Transactional Memory: We also implement a light-weight word-based software transactional memory (STM) with lazy value-based conflict checking, similar to JudoSTM [9]. Instead of having static STM API routines, Janus' STM consists only of dynamic handlers that rewrite the original memory accesses to inlined thread-private and speculative accesses. A TX_START rewrite rule is used to enable Janus' STM. On execution, handlers create code that copies the machine context (registers) into a buffer and then sets a runtime flag to indicate transactional execution, meaning that all subsequent heap accesses and out-of-frame stack accesses use Janus' STM. This means they are modified to record the addresses read and written, and to buffer stored data. A subsequent TX_FINISH rule generates code to reset the runtime flag. In addition, once a thread has finished its set of contiguous iterations and is the oldest thread running the loop, it performs a validation of each of its buffered reads against the versions in shared memory and, if successful, commits the writes sequentially.

Janus' STM could be used to support thread-level speculation. However, excessive use of speculative rewrites normally incurs a high overhead from tracking and buffering heap reads and writes, so in Janus we use it sparingly and only to ensure correctness when we encounter code discovered at runtime, such as that in shared libraries. If the code has too many speculative accesses at runtime, we abort and execute again non-speculatively.

3) Shared-Library-Call Check: Janus has the ability to parallelise loops with shared-library calls, which is not possible with static parallelisers because the code is not discovered until runtime. Janus is able to parallelise these dynamic DoAll loops with judicious use of speculation, relying on profiling to filter out dynamic code that is likely to have a large number of memory accesses. Figure 5 shows an example. A TX_START rewrite rule is generated before the shared-library call, which takes a checkpoint of the register state. During the call, all

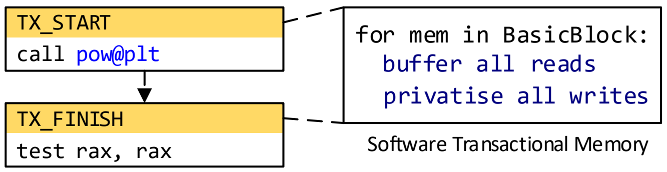


Fig. 5. Handling dynamically discovered code using speculation.

heap accesses are modified to keep track of addresses read and written and to buffer stored data. After the call, a `TX_FINISH` rule is inserted to ensure that no memory dependencies have been violated and to write back the buffered stores to shared memory in thread order. If the check fails, execution rolls back to the checkpoint and the code is re-executed, which will succeed because the thread is now the oldest and so non-speculative. This speculation scheme incurs a high overhead from tracking and buffering heap reads and writes, so we use it sparingly and only to ensure correctness.

F. Strengths of Binary Parallelisation with Janus

Janus keeps the source binary unchanged, augmenting it with an intermediate domain-specific rewrite schedule that is generated through static analysis from one or more tools. Under the control of this rewrite schedule, the dynamic binary modifier optimises the application immediately before execution. This allows it to specialise the actual implementation of the rewrite rules according to the thread that is running the code, by taking advantage of the thread-private code caches within the underlying dynamic binary modifier. In addition, it can deal with static ambiguity through the use of checks for testing runtime conditions and software transactional memory for dynamically discovered code.

G. Implementation

Dynamic Binary Translator: We implemented our dynamic binary modification tool as a client within DynamoRIO [8]. Although there were other candidates, such as Pin [10], DynInst [11], QEMU [12] and Valgrind [13], we selected DynamoRIO because of its performance and transparency to the executing application. First, DynamoRIO achieves high decoding and encoding efficiency [14] since its IR is close to the machine instructions and it employs a lazy decoding scheme that only decodes an instruction when it needs to examine it for modification. Second, it maintains a correspondence between registers used in the executable's instructions and those used when they are placed in the code cache, which is fundamental for enabling the static analyser to correctly transform instructions. (In fact, DynamoRIO does not redo register allocation at all, currently even in its trace optimisation.) Third, it has a rich API for transforming instructions within each basic block, and, fourth, although not essential, it supports both x86 and ARM instruction sets. These strengths combined meant that writing a client to interpret the rewrite schedule cleanly integrated into the dynamic modifier and our optimisations did not have to overcome significant performance overheads incurred by the framework.

Static Analyser: The prime consideration for the static analyser to produce effective rewrite schedules was to be aware of the nature and constraints of the dynamic binary modifier. For example, it must have the same definition of data structures, basic blocks, control flow, and heuristics as DynamoRIO. Existing static analysis tools, such as BAP [15], BitBlaze [16], and SecondWrite [2], lift machine code to a higher IR than we require. From the rich context of a high-level IR, they can utilise existing analysis passes from other compilers. However, for generating rewrite rules, decompiling binaries to a high-level IR may lose the mapping to the original hardware instructions. For example, an x86 instruction might be translated to multiple IR statements that deal with loading data from memory, performing the operation and subsequent flag manipulation. This complicates the mapping back down to machine instructions when we generate rewrite rules for this IR. We therefore wrote a custom tool to perform static analysis based on the Capstone disassembler library [17]. While all indirect jumps are marked as having undetermined targets, this does not limit the tool in the use cases we study as we can insert runtime checks to maintain correctness.

Limitations: C++ binaries that use STL calls to control loop flow prevent Janus from recognising loop iterators and therefore from parallelising these loops, although this could be addressed with engineering effort. In addition, Janus requires profiling to fine tune its performance, which is not feasible for some applications, or may take a prohibitive amount of time to complete.

H. Summary

Automatic parallelisation in Janus uses static analysis combined with profile information to extract DoAll parallelism from loops. Each thread executes a subset of the iterations, with runtime checks to catch data dependencies that cannot be disproved with static analysis.

III. Evaluation

We evaluated Janus on an Intel Sandy Bridge Xeon E5-2667 v4 CPU on Ubuntu 16.04 that contains eight cores (16 threads), a 25MB L3 cache and runs at a frequency of 3.3GHz with frequency scaling (turbo boost) disabled. Instead of choosing a benchmark suite with a high amount of parallelism, we selected the SPEC CPU2006 benchmarks [18] as our workloads because they represent a generic suite of applications which are considered difficult to parallelise. All benchmarks were compiled by gcc 5.4 using optimisation level `-O3` for single-threaded performance. We used all applications, apart from omnetpp, tonto, and wrf which either would not run correctly natively or have target execution times, making them unsuitable for our environment. We report the median, maximum and minimum execution times from ten runs using the reference inputs. Profiling results were obtained using the training inputs.

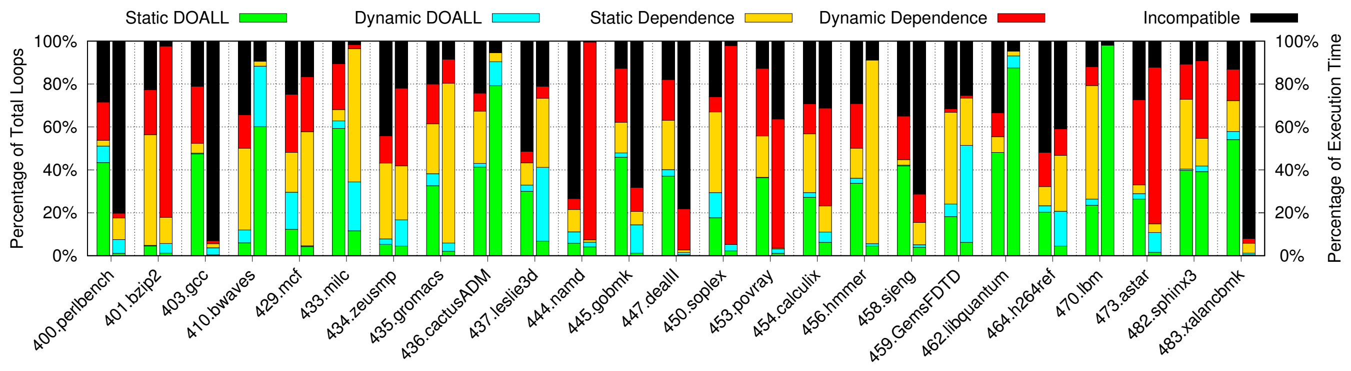


Fig. 6. Coverage fraction from binary instrumentation using training inputs

A. *Profile-Guided Static Analysis*

The focus of the static analysis phase in Janus is to identify loops that are suitable for parallelisation. Figure 6 shows the results of this analysis, aided by profile information, for all of our benchmarks. It breaks down loops into the five categories described in section II-D. Each application is shown with two bars: the first, on the left, presents the static fraction of total loops falling into each category; the second bar, on the right, presents the dynamic fraction of loops within each category (i.e., how much of the application's execution is spent in loops of each category), which is gathered from profiling.

From static analysis, for the majority of the benchmarks, Janus can analyse over half the loops. The exceptions are leslie3d, namd, and h264ref. The majority of these incompatible loops are small, with no fixed pattern for their induction variables, or contain ambiguous control flow that has typically been generated by compiler optimisation (e.g., vectorisation). Janus can parallelise the fraction of loops that are green and cyan. From profiling it is clear that these loops represent a significant fraction of application runtime for some workloads (e.g., 98% of lbm), but correspond to only a tiny fraction for other benchmarks (e.g., 1% for xalancbmk).

Although Janus can analyse some loops with cross-iteration data dependencies (yellow and red bars), these loops are the focus of future work. Those with static dependencies require synchronisation to ensure that the application semantics are maintained, and code scheduling to overcome the synchronisation overheads. Others require speculation because there may be cross-iteration dependencies that did not occur during execution with training inputs.

O3 integer and C++ benchmark binaries often contain irregular optimised loops where the loop iterators cannot be deterministically identified by our current implementation (black bars). However, this could be addressed with further engineering effort. In all, only nine of the 25 benchmarks spend at least 20% of their execution time in loops displaying DoALL parallelism. We therefore focus solely on these.

B. *Whole-Program Performance*

Figure 7 shows the performance of the SPEC binaries that are amenable to parallelisation using eight threads, normalised to native single-threaded execution for the whole application.

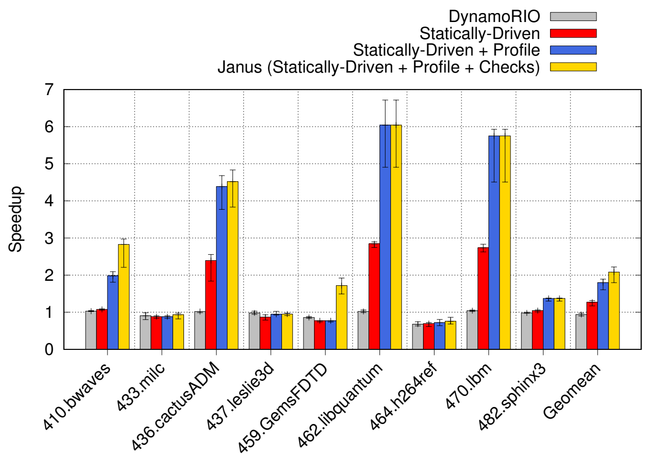


Fig. 7. Performance with parallelisation for eight threads on eight cores with each thread bound to its own core.

We show four bars per benchmark. The first shows the performance of the application when executed under DynamoRIO without performing any modification, reflecting the overhead of the dynamic binary translator. This has a significant impact on some applications, such as milc, GemsFDTD, and h264ref, which experience slowdowns of 10%, 14%, and 32% respectively. Other applications gain negligible performance, thanks to the trace optimisation that DynamoRIO performs, the highest being lbm with a speedup of 4%. On average, performance drops by 6% simply through use of the dynamic binary translator.

The second bar, labelled "Statically-Driven" represents parallelisation based on static analysis only. In this case we parallelise all loops categorised as static DoALL, and do not use any profile information or runtime checks. Although this realises performance improvements for libquantum and lbm, which achieve speedups of 2.8× and 2.7× respectively, most applications see little change. In fact, leslie3d and GemsFDTD lose performance with this approach—13% and 23% respectively compared to native execution, or 11% and 10% compared to DynamoRIO alone. This is due to the selection of loops with low coverage or those with a high invocation count where overheads of parallelisation out-weigh the benefits of shorter runtimes.

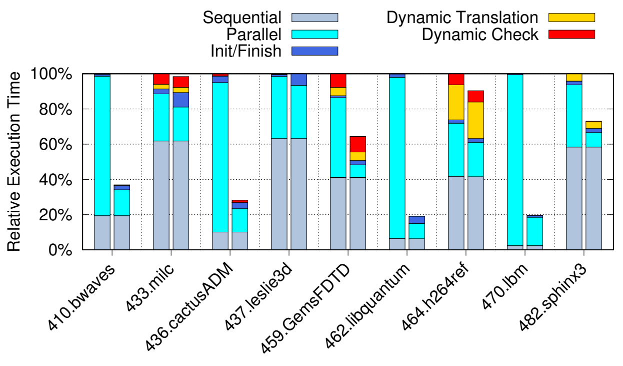


Fig. 8. Breakdown of execution time in Janus for one thread (left) and eight threads (right).

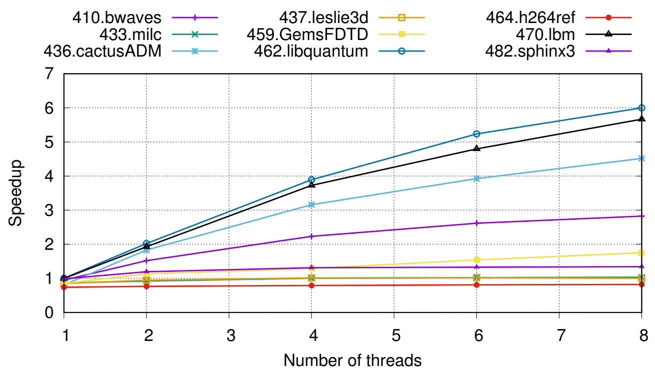


Fig. 9. Performance for different numbers of threads.

TABLE I
NUMBER OF ARRAY BOUNDS CHECKS PER LOOP THAT REQUIRES THEM, AVERAGED FOR EACH BENCHMARK.

410.bwaves	1	436.cactusADM	3	433.milc	12
459.GemsFDTD	19.5	464.h264ref	12		

The next bar, labelled "Statically-Driven + Profile" uses profile information to perform loop selection, removing these loops that are statically proven to be parallel, but not profitable to parallelise. Adding this profile significantly increases performance in the two benchmarks (libquantum and lbm) where the overwhelming majority of execution time is spent in static DOALL loops. These applications achieve speedups of 6.0× and 5.8× respectively. Other benchmarks, such as bwaves, experience more modest performance increases.

Parallelisation through Janus is shown in the final bar, which builds on the previous results by adding runtime checks to enable safe parallelisation of dynamic DOALL loops (section II-E1). This extends the coverage of parallel loops and is essential to obtain a larger speedup for bwaves (2.8×) and to gain a speedup in GemsFDTD (1.7×). For other applications, such as milc and leslie3d this optimisation does not result in higher performance than native execution due to loop candidates having low iteration counts. Unfortunately, for h264ref the overheads incurred by DynamoRIO cannot be clawed back and this application still experiences a slowdown of 24%. The bwaves benchmark contains a shared-library call in its hot loop, which requires speculation to safely parallelise, gaining a 2.9× speedup. Within this shared library call, we observed on average 49 instructions with 11 heap reads and 0 writes. These are all translated using Janus' STM. Since there are no writes, the shared library call incurs no conflicts and reasonable overhead.

C. Analysis

The overheads of parallelisation are explored further in figure 8 where we break down the execution time for each application using a sampling interval of 0.1s. The first (left) histogram for each benchmark is the breakdown using Janus with one thread, and the second (right) is the breakdown for eight threads. The breakdown for each benchmark is normalised to single-threaded Janus performance for that

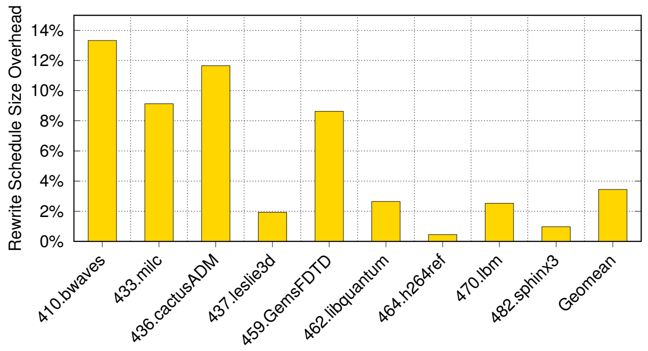


Fig. 10. Size overhead of the rewrite schedule as a percentage of the size of each executable.

benchmark. As can be seen, for the applications that obtain low speedups, there is a large fraction of time spent executing sequential code, such as milc, leslie3d, and sphinx3. Hence we are limited by Amdahl's law.

The "Init/Finish" bar, representing time taken to initialise a loop (start all threads) and finish it (wait for all threads to end) can be significant for some applications too, such as milc, and this added overhead limits parallel performance further. As shown in figure 6, time taken to perform translation, which is essentially DynamoRIO's overhead, is high for h264ref and GemsFDTD but negligible for all others. The dynamic checks added to ensure static analysis is safe (see section II-E1) also add significant overheads for half the benchmarks. Table I shows the average number of array bounds checks for each loop that requires these runtime checks, where missing benchmarks do not contain array bounds checks. For some applications, the number of checks can be high, such as in GemsFDTD where there are 13 loops, each averaging 5 different array bases.

Figure 9 shows how the performance changes with the number of threads. Both libquantum and lbm have almost ideal scaling with four threads, at 3.9× and 3.7× respectively. This tapers off with larger thread counts as the sequential parts of the application become relatively more important.

D. Rewrite Schedule Size

The size of the rewrite schedules for each application to encode the rewrite rules for parallelisation are shown in

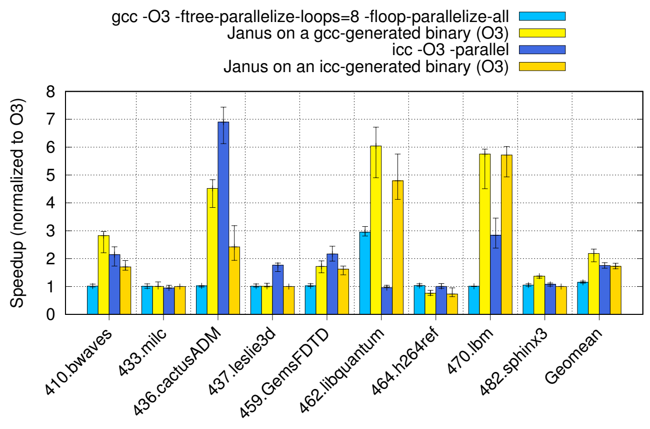


Fig. 11. Janus' speedup for sequential binaries generated by gcc and icc, compared with compiler parallelisation.

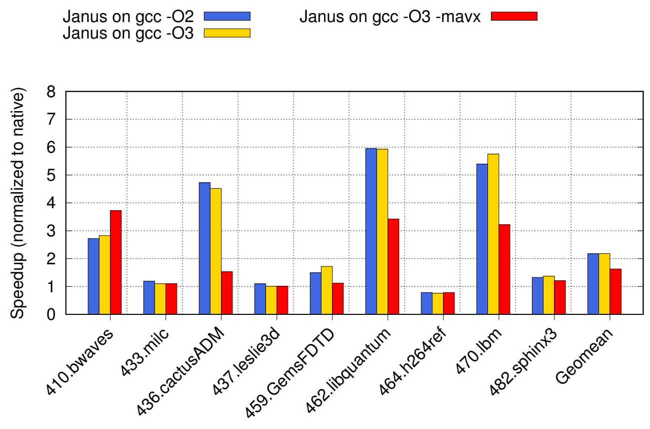


Fig. 12. Janus speedup on respective O2, O3 and vectorised O3 sequential binaries generated by gcc.

figure 10, normalised to the size of the corresponding binary. It is clear that the rewrite schedules are generally small, although they can reach over 10% if there are many transformations to apply. On average though, they are just 3.7% the size of the original executable. They can be further reduced by creating specialised rewrite rules that perform multiple common transformations at the same time.

E. Comparison with Compiler Parallelisation

Although Janus targets stripped binaries without source code, it is interesting to see whether a lack of information inhibits automatic parallelisation. To assess this, we compare the performance of Janus against binaries parallelised automatically using gcc and Intel's compiler, icc version 18.0 (figure 11). Each result is normalised against the performance of a native executable compiled with the same compiler using optimisation level O3. Neither gcc nor icc used feedback-directed optimisation.

Janus can achieve speedup on binaries from both compilers, which demonstrates that it is compiler agnostic. However, Janus achieves less speedup (1.3×) on icc binaries compared to gcc (2.2×), which mostly due to the baseline for icc being faster. Icc tends to unroll more loops and use SIMD instructions, so the average number of iterations each thread executes is actually less, which magnifies the threading overhead. In addition, icc alters the code in a way that is less amenable to our static analysis, meaning Janus executes additional runtime checks, slowing down parallel performance.

However, it is also clear that loss of symbolic information is not a barrier to automatic parallelisation. For the benchmarks when Janus performs the best (libquantum and lbm), neither gcc nor icc can achieve the same level of parallel performance, although they could improve with their own feedback-directed optimisations. Further comparisons are not helpful since Janus is targeting a different application domain without source code. Gcc, in general, does not manage to achieve significant speedup from these applications from our experiments. Icc performs better, especially for cactusADM where it carries out heavy vectorisation in addition to parallelisation. On average, however, Janus achieves its best speedup with gcc's binaries (2.2× compared to 1.1× for gcc) and almost the same speedup as icc on icc's binaries (1.7× compared with 1.8×).

F. Impact of Compiler Optimisation

As described in section III-E, different compiler optimisations can affect Janus' ability to parallelise loops. We explore this further in figure 12 where we show Janus' speedup on binaries generated by gcc with different optimisation levels. The performance on O2 binaries, compared to O3, is negligible, and mainly due to differences in where data is placed (e.g., in registers or on the stack). Generic vectorisation is already performed at O3, using SSE instructions. Adding more aggressive vectorisation by using -mavx generally limits the amount of performance Janus can obtain. One reason is that the vectorised code is difficult to analyse (i.e., alias analysis) due to peeling for alignment. However, this can be resolved through further engineering effort in the alias analysis pass. Some loop iterators are difficult to identify because compilers tend to keep two different copies of unrolled loops in the same outer loop, complicating analysis. For some loops, there are not enough iterations to be profitable after unrolling and vectorisation, making Janus reject those loops during profiling. The exception is bwaves, which obtains a 3.7× speedup on the vectorised binary, compared with just 2.8× on the O3 binary. The reason is that false sharing within the cache hierarchy limits speedups when compiled with O3; vectorisation alleviates this bottleneck by bringing more consecutive data into the cache per iteration of the loop.

G. Summary

Janus demonstrates parallel performance improvement despite working in a dynamic binary modifier thanks to the combination of static analysis, profiling, and runtime checks.

IV. Related Work

Binary Parallelisation: A number of binary parallelisers have been developed for a variety of architectures, but, to the best of our knowledge, none are publicly available, which makes a quantitative comparison impossible. We provide a qualitative comparison and summarise in table II.

TABLE II
SUMMARY OF KEY FEATURES OF REAL-SYSTEM BINARY PARALLELISATION TOOLS (* MANUAL PROFILING REQUIRED).

Tool	Platform	Open source	Automatic	Runtime checks	Shared-libraries	Parallelisation	Spectrum
Yardımcı and Franz [19]	PowerPC		✓			Static DOALL	Generic binaries
SecondWrite [3], [5]	x86-64		✓*	✓		Affine loops	Affine binaries
Pradelle et al [4]	x86-64		✓*			Decompile Src2Src	Affine binaries
Janus	x86-64, AArch64	✓	✓	✓	✓	Dynamic DOALL	Generic binaries

Yardımcı and Franz [19] proposed a binary parallelisation scheme for PowerPC binaries, which combines static analysis and dynamic binary parallelisation by transforming executables into an intermediate form, with just-in-time compilation in their dynamic software layer. Their approach requires profiling at runtime to determine whether to perform parallisation, whereas Janus performs loop selection statically based on profile information.

Kotha et al. [3] proposed a binary paralleliser built in the LLVM-based SecondWrite [2] static binary rewriter, which focuses on affine DOALL loops with statically-known iteration counts. They rely on static dependence analysis and polyhedral analysis to disprove cross-iteration dependences. As it is a purely static approach, they fail to parallelise loops when the memory accesses cannot be determined. Their following work [5] alleviates these weaknesses by including runtime checks to verify their static analysis for loop bounds. Janus takes advantage of a similar static alias analysis but is able to parallelise more loops, including those with irregular control flow and shared-library function calls, where correctness is guaranteed through statically-controlled runtime checks and software speculation.

Neither Yardımcı's nor Kotha's works are fully automatic as they require user intervention to perform parallelisation or fine tune performance. Janus, in contrast, can perform parallelisation of a binary completely autonomously.

There have been a number of articles investigating the feasibility of dynamic binary parallelisation through simulation and limit studies [7], [6], [20], [21]. Edler von Koch and Franke, for example, model dynamic binary parallelisation, finding realistic upper bound speedups of only $1.09\times$ because everything is performed at runtime [7] whereas some dependencies could have been removed using static analysis. Others model thread-level speculation, which is not supported within existing commercial hardware and requires very large overhead to implement in software.

Automated Program Optimisation: The most similar tool to Janus is a combination of Calpa [22] and DyC [23] (Calpa-DyC). DyC is a JIT compiler driven through user annotations in C source code. The annotations specify an intermediate structure for variables and code that is lazily compiled to the underlying hardware at runtime. Performance can be achieved through a variety of dynamic peephole optimisations. Calpa automates the process of adding annotations to guide DyC through profiling. Compared to Janus, Calpa-DyC is limited to annotating C language programs, whereas Janus is able to modify binaries compiled from any language without need of the source.

Binary Optimisation: DynamoRIO [8] is a robust and well-supported open-source runtime code manipulation system which originates from the well-known high-performance binary translator, Dynamo [24]. Other dynamic modification tools, such as Pin [10], are closed source, and, like DynInst [25], are more focused on binary instrumentation. Some static binary translators, such as peephole superoptimisers [26], DIABLO [27], and ATOM [28] use extra profiling or debugging information to compensate for the loss of information and ambiguities at the binary level. The Sun Studio Binary Code Optimizer [29] and Microsoft Vulcan [30] are well-known commercial tools for rewriting binaries for better single-threaded performance, but both rely on instrumentation to collect profiling information.

Automatic Parallelisation: Compiler-based automatic parallelisation relies on a program's source code to compile into parallel binaries. Conventional automatic parallelising compilers, such as Polaris [31], SUIF [32], PLUTO [33], and LLVM Polly [34] reject ambiguous irregular loops for DOALL or polyhedral parallelisation. HELIX [35] and DSWP [36] handle DOACROSS and DOPIPE parallelism respectively by employing more aggressive and expensive data dependence analysis and code restructuring transformations. Janus does not propose new approaches to automatic parallelisation but instead adopts existing techniques efficiently, applying them to binaries.

V. CONCLUSION

We have presented Janus, a framework for dynamic binary parallelisation that incorporates static analysis, profile information, and runtime checks. Using a custom static analyser, Janus determines the transformations required to parallelise an application, recording them in a rewrite schedule specific to the binary. The dynamic binary modifier reads the rewrite rules it contains and transforms the application at runtime as instructed. We use Janus to parallelise a range of SPEC CPU 2006 applications gaining average speedups of $2.1\times$.

ACKNOWLEDGEMENTS

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through grant references EP/K026399/1 and EP/P020011/1, and Arm Ltd. Additional data related to this publication is available in the data repository at https://doi.org/10.17863/CAM.33893.

REFERENCES

[1] J. T. Richards, J. Brezin, C. B. Swart, and C. A. Halverson, "A decade of progress in parallel programming productivity," *Commun. ACM*, vol. 57, 2014.

[2] K. Anand, M. Smithson, A. Kotha, K. Elwazeer, and R. Barua, “Decompilation to compiler high IR in a binary rewriter,” University of Maryland, Tech. Rep., 2010.

[3] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua, “Automatic parallelization in a binary rewriter,” in *MICRO*, 2010.

[4] B. Pradelle, A. Ketterlin, and P. Clauss, “Transparent parallelization of binary code,” in *First International Workshop on Polyhedral Compilation Techniques, IMPACT 2011, in Conjunction with CGO 2011*, 2011.

[5] A. Kotha, K. Anand, T. Creech, K. ElWazeer, M. Smithson, and R. Barua, “Affine parallelization of loops with run-time dependent bounds from binaries,” in *ESOP*, 2014.

[6] B. Hertzberg, “Runtime automatic speculative parallelization of sequential programs,” Ph.D. dissertation, Stanford University, 2009.

[7] T. J. K. Edler von Koch and B. Franke, “Limits of region-based dynamic binary parallelization,” in *VEE*, 2013.

[8] D. Bruening, T. Garnett, and S. Amarasinghe, “An infrastructure for adaptive dynamic optimization,” in *CGO*, 2003.

[9] M. Olszewski, J. Cutler, and J. G. Steffan, “Judostm: A dynamic binary-rewriting approach to software transactional memory,” in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, 2007.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *PLDI*, 2005.

[11] C. C. Williams and J. K. Hollingsworth, “Interactive binary instrumentation,” in *RAMSS*, 2004.

[12] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX Annual Technical Conference*, 2005.

[13] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *PLDI*, 2007.

[14] D. L. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[15] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “Bap: A binary analysis platform,” in *CAV*, 2011.

[16] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “Bitblaze: A new approach to computer security via binary analysis,” in *ICISS*, 2008.

[17] N. A. Quynh, “Capstone: Next-gen disassembly framework,” in *Blackhat USA*, 2014.

[18] S. P. E. C. (SPEC), “SPEC CPU 2006,” https://www.spec.org/cpu2006/, 2006.

[19] E. Yardımcı and M. Franz, “Dynamic parallelization and mapping of binary executables on hierarchical platforms,” in *CF*, 2006.

[20] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T.-f. Ngai, and J. Fang, “Dynamic parallelization of single-threaded binary programs using speculative slicing,” in *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[21] J. Yang, K. Skadron, M. L. Soffa, and K. Whitehouse, “Potential of dynamic binary parallelization,” in *Workshop on Unique Chips and Systems UCAS-7*, 2012.

[22] M. Mock, C. Chambers, and S. J. Eggers, “Calpa: A tool for automating selective dynamic compilation,” in *MICRO*, 2000.

[23] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, “Dyc: an expressive annotation-directed dynamic compiler for c,” *Theoretical Computer Science*, vol. 248, 2000.

[24] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: a transparent dynamic optimization system,” in *PLDI*, 2000.

[25] J. K. Hollingsworth, B. P. Miller, and J. Cargille, “Dynamic program instrumentation for scalable performance tools,” in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, 1994.

[26] S. Bansal and A. Aiken, “Binary translation using peephole superoptimizers,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[27] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, “Diablo: a reliable, retargetable and extensible link-time rewriting framework,” in *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, 2005.

[28] A. Eustace and A. Srivastava, “Atom: A flexible interface for building high performance program analysis tools,” in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON’95, 1995.

[29] S. Lobo, “The sun studio binary code optimizer,” http://www.oracle.com/technetwork/server-storage/solaris/binopt-136601.html, 1999.

[30] A. Srivastava, A. Edwards, and H. Vo, “Vulcan: Binary transformation in a distributed environment,” Microsoft Research, Tech. Rep. MSR-TR-2001-50, 2001.

[31] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, “Polaris: The next generation in parallelizing compilers,” in *Workshop on Languages and Compilers for Parallel Computing*, 1994.

[32] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam *et al.*, “Suif: An infrastructure for research on parallelizing and optimizing compilers,” *ACM Sigplan Notices*, 1994.

[33] U. Bondhugula, J. Ramanujam, and P. Sadayappan, “Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer,” The Ohio State University, Tech. Rep., 2007.

[34] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, “Polly-polyhedral optimization in llvm,” in *IMPACT*, 2011.

[35] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, “Helix: automatic parallelization of irregular programs for chip multiprocessing,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[36] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, “Automatic thread extraction with decoupled software pipelining,” in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005.