A Generalized Framework for Automatic Scripting Language Parallelization

Taewook Oh[*] Stephen R. Beard Nick P. Johnson[†] Sergiy Popovych David I. August
Princeton University
Princeton, NJ, USA
{twoh,sbeard,npjohnso,popovych,august}@princeton.edu

***Abstract*—**
Computational scientists are typically not expert programmers, and thus work in easy to use dynamic languages. However, they have very high performance requirements, due to their large datasets and experimental setups. Thus, the performance required for computational science must be extracted from dynamic languages in a manner that is transparent to the programmer. Current approaches to optimize and parallelize dynamic languages, such as just-in-time compilation and highly optimized interpreters, require a huge amount of implementation effort and are typically only effective for a single language. However, scientists in different fields use different languages, depending upon their needs.

This paper presents techniques to enable automatic extraction of parallelism within scripts that are universally applicable across multiple different dynamic scripting languages. The key insight is that combining a script with its interpreter, through program specialization techniques, will embed any parallelism within the script into the combined program that can then be extracted via automatic parallelization techniques. Additionally, this paper presents several enhancements to existing speculative automatic parallelization techniques to handle the dependence patterns created by the specialization process. A prototype of the proposed technique, called Partial Evaluation with Parallelization (PEP), is evaluated against two open-source script interpreters with 6 input linear algebra kernel scripts each. The resulting geomean speedup of 5.10× on a 24-core machine shows the potential of the generalized approach in automatic extraction of parallelism in dynamic scripting languages.

***Keywords*-Multicore Processing; Parallel processing;**

I. Introduction

The rise of computational science has put computational efficiency on the critical path for many scientific fields. This is true even for scientists who are not strong programmers and who would rather focus on theory and experiments in their field of study than on programming [40]. This has lead to the widespread use of dynamic scripting languages, including domain specific languages (DSL), in the scientific community, as such languages are typically easier for non-expert programmers to use. Unfortunately, these languages have typically sacrificed performance in exchange for their productivity. Thus, there is a conflict between the computational scientists need for high performance and their language choice.

[*]Work performed at Princeton University. Now at Facebook.
[†]Work performed at Princeton University. Now at D. E. Shaw Research.

Given the parallel nature of today's computational resources (from multicore architectures, to GPUs, to clusters), parallelism is essentially a requirement for creating high performance programs. This is even more true for scientific programs, as many scientific computations are amenable to parallelization. However, manually parallelizing programs is known to be a time consuming and error-prone task, even for skilled programmers. Considering that ease of use is the primary reason behind the use of scripting languages in scientific computing, requiring manual parallelization in these languages does not make sense.

Automatic parallelization techniques have had success in parallelizing both scientific [6], [48], [12] and general purpose applications [8], [19], [47], [63]. However, most efforts to improve scripting language performance have not been focused on extracting parallelism. Many performance optimizations on script interpreters or just-in-time (JIT) compilers have been proposed [16], [1], [3], [2], [43], but there has been little work done on automatically extracting parallelism via JIT compilers [27]. Some have investigated reimplementing the interpreter to take advantage of the parallelism within the input script [25], [31], [55]. However, the implementation of JIT and interpreter optimizations are typically only effective for a single language. There currently does not exist a single dominant scripting language. Different domains rely on different languages, with new DSLs constantly appearing to meet the needs of scientists and researchers in specialized fields. Advances in modern compilers proved that a significant amount of engineering work can be amortized by having one tool common to many languages (e.g. C, C++, Swift, etc. in LLVM) rather than many tools for many languages.

This paper presents a novel solution to extract the parallelism within scientific programs written in dynamic scripting languages, in working towards restoring the one-tool-many-languages paradigm. Script interpreters are, in some sense, incomplete and unpredictable programs. This nature causes standard compiler analysis and speculation techniques to fail when applied to the interpreter alone. The key insight of this work is that combining a script and its interpreter using program specialization techniques [33] causes the resulting program to become predictable and amenable to compiler optimization; what's more, any parallelism within the script is embedded into the combined program and is thus exposed to the compiler's automatic parallelization techniques. With this approach, multiple scripting languages can be handled with a single system.

The structure of interpreters that have been specialized against a script are too complex to be handled by existing automatic parallelization techniques. This paper additionally presents two enhancements to existing automatic parallelization techniques to deal with the complex structure of specialized interpreters. The first technique is *context-sensitive* speculation. Although the specialization process folds parallelism into the specialized program, it generates aggressively unrolled loops that are difficult for analysis and speculation techniques to reason about. Context-sensitive speculation overcomes this limitation by improving the precision of speculative dependence analysis. Second, this paper presents performance optimizations to the speculative run-time system. Extensive use of speculation is unavoidable to automatically parallelize complex programs. However, existing run-time systems add significant overhead for each speculated memory operation and can only tolerate a few speculated operations per loop before the overhead completely eclipses the benefit of parallelization. By resolving this performance issue, the optimized system proposed in this paper is able to handle several thousand speculated instructions, whereas previous systems could handle only a few.

This paper additionally presents a prototype of the proposed technique called Partial Evaluation with Parallelization (PEP), the first fully automatic system that extracts parallelism within scripts and is applicable to multiple dynamic scripting languages. PEP was evaluated using open-source Lua [24] and Perl [37] script interpreters. PEP specialized both interpreters against 6 sequential scripts that describe linear algebra kernels containing latent parallelism, automatically parallelized the resulting specialized programs, and achieved a geomean speedup of 5.10× on a 24-core machine. Results from the first prototype implementation demonstrate the promising potential of the proposed technique.

The primary contributions of this work are:

- The first fully-automatic technique to harness parallelism within programs written in dynamic scripting languages by combining program specialization with automatic parallelization;
- A set of context-sensitive speculation techniques to extract parallelism out of complex programs (such as specialized script interpreters);
- Performance optimizations to the run-time system supporting speculative parallelization, which enables scalable speedup of extensively speculated parallel programs; and,
- Implementation and evaluation on two real-world interpreters.

II. Background and Motivation

A. Program Specialization

Program specialization, sometimes referred to as partial evaluation, optimizes a program for the values that remain invariant across program execution. These invariant values include known, fixed program inputs. Constant propagation can be considered a simple form of program specialization. Specialized programs can be generated either at compile-time [11], [26] or at run-time [5], [17], [52]. Many prior specialization techniques rely on user annotations to find invariants or generate efficient code [4], [11], [17], [26]. Recent work proposes fully-automatic run-time [5], [52] and compile-time [33] program specialization techniques that do not require user annotations.

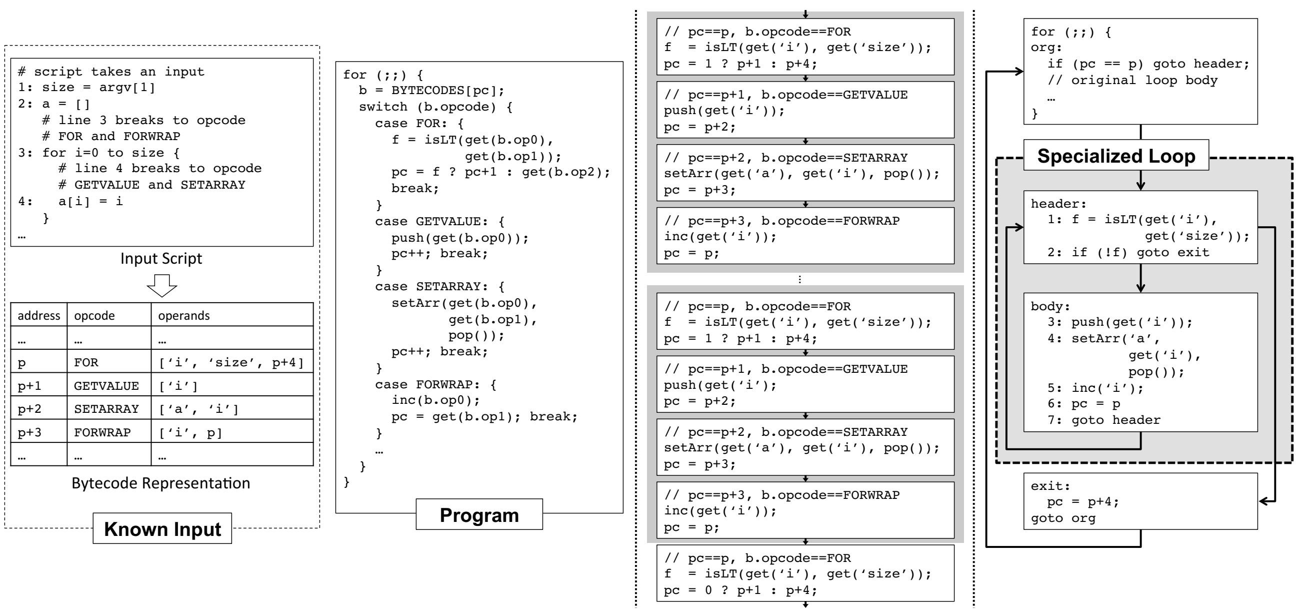
Invariant-induced Pattern based Loop Specialization (IPLS) [33] is a fully-automatic, compile-time program specialization technique. A key feature of IPLS is that it generates a program specialized for the predictable patterns of values induced by program invariants across loop iterations. IPLS traces the values of instructions in hot loops that depend solely on program invariants and discovers repeating patterns from the traces. IPLS generates a specialized loop by unrolling the original loop for the length of the detected pattern and specializes each unrolled iteration with respect to the value in the pattern.

B. Automatic Speculative Parallelization

The ideal method of harnessing loop-level parallelism is DOALL parallelization. Iterations of DOALL loops run completely independent of other iterations, thus resulting in scalable parallel speedup. However, DOALL parallelization can only be applied to loops with no loop-carried dependences, often referred to as embarrassingly parallel loops.

Decoupled Software Pipelining (DSWP) [35] overcomes the limited applicability of DOALL parallelization and is applicable even when loop-carried dependences exist. DSWP divides the loop body into multiple stages and assigns each stage to different threads to create a pipeline, where data does not flow from later stages to earlier stages. DSWP first builds a DAG_{SCC} of the Program Dependence Graph (PDG) [15] of the target loop by coalescing each strongly connected component (SCCs) in the PDG into a single node, then assigns those nodes to stages to identify a pipeline schedule. If there is a *parallel stage*, a stage with no loop-carried dependences, the stage can run independently of different iterations of the same stage to achieve better scalability [47]. DSWP is more tolerant than DOACROSS, another parallelization scheme that is applicable to loops with loop-carried dependences, to high communication latency between threads [61].

The performance improvement of parallel transformations is limited by the amount of parallelism extracted from the program. Extracting parallelism is often prevented by



```
# script takes an input
1: size = argv[1]
2: a = []
   # line 3 breaks to opcode
   # FOR and FORWRAP
3: for i=0 to size {
     # line 4 breaks to opcode
     # GETVALUE and SETARRAY
4:   a[i] = i
   }
…
```

address	opcode	operands
…	…	…
p	FOR	[‘i’, ‘size’, p+4]
p+1	GETVALUE	[‘i’]
p+2	SETARRAY	[‘a’, ‘i’]
p+3	FORWRAP	[‘i’, p]
…	…	…

```
for (;;) {
  b = BYTECODES[pc];
  switch (b.opcode) {
    case FOR: {
      f = isLT(get(b.op0),
               get(b.op1));
      pc = f ? pc+1 : get(b.op2);
      break;
    }
    case GETVALUE: {
      push(get(b.op0));
      pc++; break;
    }
    case SETARRAY: {
      setArr(get(b.op0),
             get(b.op1),
             pop());
      pc++; break;
    }
    case FORWRAP: {
      inc(b.op0);
      pc = get(b.op1); break;
    }
    …
  }
}
```

```
// pc==p, b.opcode==FOR
f  = isLT(get(‘i’), get(‘size’));
pc = 1 ? p+1 : p+4;

// pc==p+1, b.opcode==GETVALUE
push(get(‘i’));
pc = p+2;

// pc==p+2, b.opcode==SETARRAY
setArr(get(‘a’), get(‘i’), pop());
pc = p+3;

// pc==p+3, b.opcode==FORWRAP
inc(get(‘i’));
pc = p;

⋮

// pc==p, b.opcode==FOR
f  = isLT(get(‘i’), get(‘size’));
pc = 1 ? p+1 : p+4;

// pc==p+1, b.opcode==GETVALUE
push(get(‘i’);
pc = p+2;

// pc==p+2, b.opcode==SETARRAY
setArr(get(‘a’), get(‘i’), pop());
pc = p+3;

// pc==p+3, b.opcode==FORWRAP
inc(get(‘i’));
pc = p;

// pc==p, b.opcode==FOR
f  = isLT(get(‘i’), get(‘size’));
pc = 0 ? p+1 : p+4;
```

```
for (;;) {
org:
  if (pc == p) goto header;
  // original loop body
  …
}

header:
  1: f = isLT(get(‘i’),
              get(‘size’));
  2: if (!f) goto exit

body:
  3: push(get(‘i’));
  4: setArr(‘a’,
            get(‘i’),
            pop());
  5: inc(‘i’);
  6: pc = p
  7: goto header

exit:
  pc = p+4;
goto org
```

(a) (b) (c)

Figure 1: Example of interpreter specialization. (a) An input script, its bytecode representation, and a snippet of the main interpreter loop. (b) Execution trace of the interpreter running the script. The grey boxes represent four iterations of the interpreter loop, which is one iteration of the loop in the input script. Note that `isLT` evaluates to 1 in the first two iterations and 0 in the last. (c) Resulting specialized loop from IPLS.

imprecise analysis, especially for general purpose programs with irregular data structures and complex control-flows. However, these programs have many statically unresolvable dependences that may not manifest at runtime.

Speculative parallelization [22], [28], [45], [54], [65] can overcome the limits of static analysis by speculatively removing dependences that are unlikely to occur in practice, thereby giving the compiler increased freedom to apply parallelizing transforms. To ensure that the dependences that were speculatively removed do not actually manifest during the program's execution, the compiler inserts validation checks. If the speculative assumptions are violated, the speculative run-time system signals *misspeculation* and then *recovers* the program to non-speculative state. Recovery is typically handled through a checkpoint and rollback mechanism. To support speculation in conjunction with pipelined parallelization, a run-time system called Multi-Threaded Transactions (MTXs) [60] has been proposed. As the original MTX proposal requires extensive hardware modifications, a software-only implementation of MTX (SMTX [46]) was developed as well.

C. Motivating Example

Figure 1(a) describes the high level algorithm of an interpreter and input script. The script takes an input from the command line to set the value of a variable `size`, and runs a loop that iterates with induction variable `i` from zero to `size` to initialize the index `i` of array `a` with `i`. The loop in the input script is described with four different instructions at the bytecode level: `FOR`, `GETVALUE`, `SETARRAY`, and `FORWRAP`. There are no loop carried dependences, thus the loop contains DOALL parallelism.

Due to the loop in the script, the main loop of the interpreter program experiences a recurring control- and data-flow pattern during execution. As shown in Figure 1(b), the same code blocks and values are repeated across iterations of the interpreter's main loop. IPLS captures these repeating patterns and generates a specialized loop, as depicted in Figure 1(c). Some bookkeeping instructions, like an instruction that increases `pc`, are optimized out during the specialization process.

IPLS was invented to optimize the performance by specializing target loops in this way. However, we found that there is another important consequence of IPLS; with IPLS, parallelism expressed by the fixed input (the input script in this example) is expressed by the specialized loop. The variability in the potential dynamic behaviors of the interpreter in Figure 1(a) makes it a poor choice for automatic parallelization. In contrast, the specialized loop in Figure 1(c) reflects the structure of the loop in the input script, as well as the parallelism within the loop.

Figure 2(a) shows the PDG of the specialized loop. Dashed boxes in Figure 2(a) represent basic blocks. Static analysis may not be able to prove that there is no data

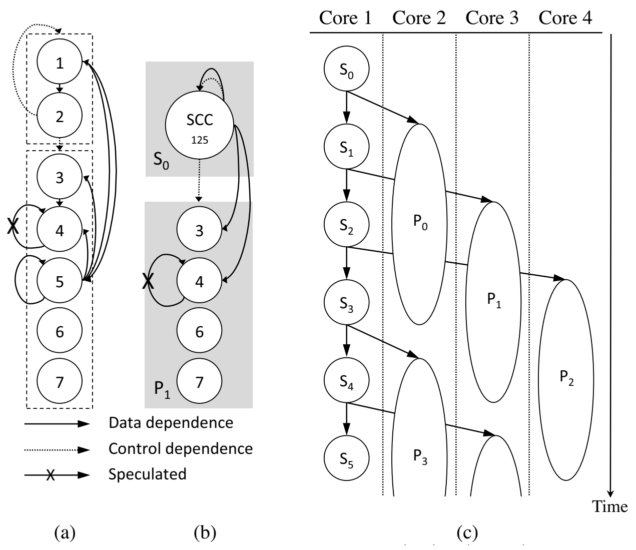


Figure 2: (a) PDG of the specialized loop from Figure 1(c). Note that the control dependence from node 2 applies to both sets of nodes 1-2 and 3-7. (b) $\mathrm{DAG}_{\mathrm{SCC}}$ of PDG from (a). (c) Parallel execution plan using DSWP.

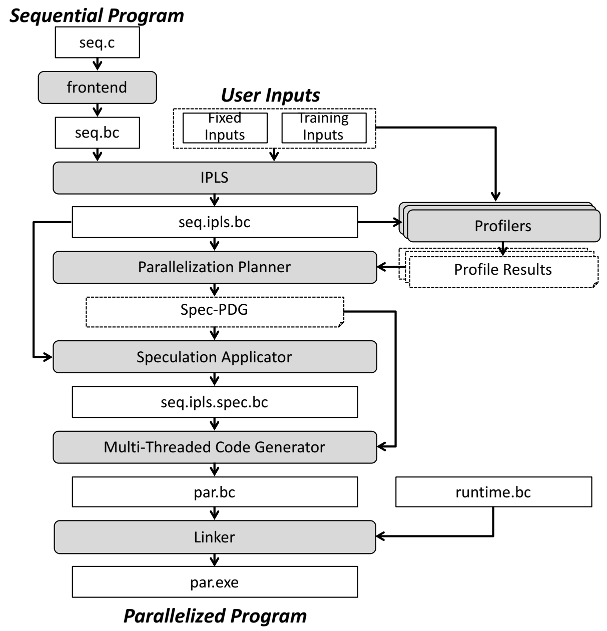


Figure 3: The PEP workflow. A sequential program, fixed inputs, and training inputs are inputs to the system. After undergoing specialization, PEP profiles the program and then performs speculative parallelization. Note that `.bc` files are intermediate files containing LLVM bitcode.

dependence between the calls of `setArr` across multiple iterations; however, *data-dependence profiling* can determine that the dependence is unlikely to manifest and can be speculatively removed. Figure 2(b) is a $\mathrm{DAG}_{\mathrm{SCC}}$ of Figure 2(a). The strongly-connected component formulated around nodes 1, 2, and 5 is merged into a single node in the graph.

The loop-carried dependences in SCC_{125} prevent the DOALL parallelism in the input script from appearing in the specialized loop. Conceptually, this is because simple values from the original script become complex, and thus hard to analyze, data structures stored in memory in the specialized interpreter; `i` and `size` change from being simple integers to data structures containing values and meta-data. Thus, standard techniques for dealing with common dependence patterns, such as loop induction variables, are no longer applicable. However, instructions 3, 4, 6, and 7 do not contain loop-carried dependences, so each invocation of these instructions for different loop iterations can run in parallel. Applying DSWP results in a two stage pipeline where SCC_{125} is in a sequential stage, to respect its loop-carried dependence, and instructions 3, 4, 6, and 7 are in a parallel stage. Figure 2(c) shows that exploiting the parallelism within the specialized loop can deliver speedup on multi-core machines, assuming that the parallel stage dominates execution time of the specialized loop.

Existing automatic parallelization techniques work nicely for such simple examples. Unfortunately, the loops that result from specializing real interpreters and scripts are much more complex and beyond the capabilities of existing parallelization techniques. First, previous speculative analyses have limited precision and thus do not identify many dependences that do not manifest at run-time. As a result, existing techniques fail to find parallelism obfuscated by these dependences. Second, the existing design of SMTX incurs significant performance overheads, and is therefore applicable only when a small number of dependences are speculatively removed. However, aggressive speculation is required to discover parallelism in complex programs. Section IV and V describe how the techniques proposed in this paper overcome these problems.

III. PEP Workflow

Figure 3 depicts the PEP workflow. The workflow is fully automatic; once the user provides a C or C++ program (e.g. a script interpreter), a fixed input (e.g. an input script), and a representative training input (e.g. an input to the script), no more user intervention is required. Currently, PEP is evaluated on harnessing DOALL parallelism within the fixed input, which becomes pipeline parallelism in the specialized code.

IPLS: PEP compiles a sequential version of the input program to an Intermediate Representation (IR) and specializes it with respect to *fixed inputs* using IPLS [33]. IPLS employes a profiler to record traces of the values computed by expressions that depend solely on program invariants, including fixed inputs. After profiling, IPLS codifies long and frequent traces of execution by recognizing recurring patterns in those traces. IPLS unrolls and specializes loops against these patterns to optimize for the expected case.

Profiling: PEP further enables parallelization using high-confidence, profile-guided speculation of biased conditional branches, predictable values, and memory dependences. PEP profiles the specialized program on a representative input to estimate its expected-case behavior. Whereas a Program Dependence Graph (PDG) [15] represents a program's worst-case behavior, the speculation phase builds a *speculative PDG* representing the program's expected-case behavior. Only loop-carried dependences are speculatively removed, as intra-iteration dependences have minimal effect on the applicability of parallel transformations.

A **control-flow edge profiler** (LLVM's `-insert-edgeprofiling` [21]) identifies heavily biased branches. By speculating that heavily biased branches are unconditional branches, unlikely instructions become dead code and some control dependences are removed from the speculative PDG. A **linear value prediction profiler** discovers load instructions whose value can be predicted as a linear function of the loop's canonical induction variable. The profiler drives value prediction speculation to eliminate loop-carried data dependences from the speculative PDG. Finally, a **memory dependence profiler** records the memory dependences observed during a training run. PEP speculates that memory dependences not observed during profiling do not exist and removes them from the speculative PDG.

Profiling results are *context-sensitive* for linear value prediction and memory dependence profiles. Section IV describes how PEP leverages context-sensitivity.

Parallelization Planner: The parallelization planner formulates a concrete plan to achieve parallel execution based upon the speculative PDG. The planner considers the parallelizability of the speculative PDG of each hot loop [47].

The planner rejects any parallelization that it does not expect to yield a performance improvement beyond a certain threshold. Expected speedup is computed based on the weight of the target loop, weight of the instructions in each pipeline stage, and the number of cores in the target system. The weight of each loop/instruction is provided by the control-flow edge profiler, while the number of cores is provided as a compiler option.

Speculation Applicator and Multi-Threaded Code Generator: Once the parallelization planner has selected a parallelization plan, PEP proceeds to the transformation phase. First, it transforms the sequential IR into a speculative-sequential IR. The speculation applicator inserts new instructions for speculation and to validate that all speculative assumptions hold true at run-time. If any speculative assumption fails, these validation instructions signal *misspeculation* to initiate the recovery mechanism.

After applying speculation, PEP parallelizes the speculative-sequential IR. The Multi-Threaded Code Generation (MTCG) algorithm automatically transforms the speculative-sequential IR into a DSWP execution schedule [34]. Instructions assigned to each stage of the pipeline are copied into new functions representing each pipeline stage. MTCG additionally inserts produce/consume communication primitives to preserve data dependences that span pipeline stages and duplicates control flow instructions to preserve control dependences that span pipeline stages.

```
L:
  Object* o_i = getObj("i");                 (1)
  ArrayObject* o_arr = getArrObj("arr");     (2)
  Object* o_elem = getObj(o_arr, o_i);       (3)
  add1(o_i, o_i);                            (4)
  add1(o_elem, o_elem);                      (5)
  if (isLT(o_i, getObj("size"))) goto L;     (6)
…
```

(a)

```
void add1(Object* dst, Object* src) {
  FLAG flag = src->flag;       (7)
  if (flag == INT) {           (8)
    dst->flag = INT;           (9)
    dst->value.i += 1;        (10)
  } else {
    dst->flag = FLOAT;        (11)
    dst->value.f += 1.0;      (12)
  }
}
```

(b)

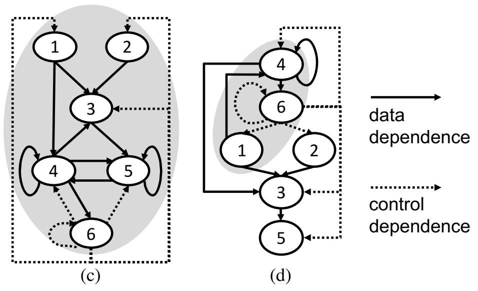


Figure 4: Benefits of context sensitivity. (a) Specialized interpreter code example. (b) Source code for the `add1` function. (c) PDG of profiling results with no context sensitivity resulting in no opportunity for parallelism where the grey region represents an SCC. (d) PDG of profiling results with context sensitivity. The self edge on node 5, and the mutual edge between nodes 4 and 5 has been removed, which makes the SCC smaller and enables parallelism.

IV. Context-Sensitive Speculation

PEP's value and memory dependence profilers provide context-sensitive information. Figure 4 illustrates the importance of context-sensitivity in profiling results.

Figure 4(a) is a code snippet of an interpreter program specialized for a script `arr[i]+=1`, where `i` is the loop induction variable. Lines `(1)` to `(3)` load the memory object for variables `i`, `arr`, and `arr[i]`, respectively. The function `add1` in Figure 4(b) is called twice in the loop. Once at line `(4)` to increment the induction variable `i`, which does have a loop carried data dependence across iterations of the loop `L`, and once at line `(5)` to increment the value of `arr[i]`, which has no loop carried data dependence across iterations of the loop `L`. Note that these were the same callsite in the non-specialized version of the interpreter.

If profiling results ignore the calling context and only report that there are loop-carried dependences 9→7 and 10→10 in the `add1` function, the compiler will conservatively assume that those dependences exist across *every* callsite of `add1`. Figure 4(c) is the speculative PDG of loop `L` for such a case.[1] Ignoring context information adds dependence edges 4→5 and 5→4, as well as self-dependence edges 4→4 and 5→5. These edges create a single SCC in the PDG, which means that there is no chance for parallelization. However, if context information is provided by the profiling results, the compiler can apply speculation further to remove edges 4→5, 5→4, and 5→5. The resulting PDG (Figure 4(d)) shows that only nodes 1, 4 and 6 form an SCC, thus parallel transformation can be applied.

The following subsections describe how PEP implements, in a context sensitive environment, linear value-prediction/memory dependence speculation, which are critical to unlock the parallelism out of complex programs.

A. Linear Value Prediction Speculation

Linear value prediction speculation enables optimization by removing dependences incident on load instructions that meet the following criterion:

Linear Value Prediction Criterion: *Let LD be a load instruction in loop L, and let $V(I)$ be the value which LD reads from memory during iteration I of loop L. LD is* speculatively linear *in L under context C iff throughout the profile run, $V(I) = m \times I + b$ for some fixed constants m, b.*

The context C specifies a position within loop L. If instruction LD is syntactically contained in loop L, then $LD = C$. Alternatively, if instruction LD is buried within a procedure call, then context C references the call site in loop L. In the example of Figure 4(a), where the target loop is `L`, the execution context allows the system to distinguish between the execution of the `add1` function for `o_i` from the execution for `o_elem`. In the former case the execution context is `(4)`, while in the latter case it is `(5)`.

The linear value prediction profiler determines coefficients m, b to characterize each speculatively linear load. The profiler finds m and b by computing the linear interpolant from the sample points computed by first two iterations, then checks the consistency of the linear interpolant with additional samples from following iterations. $m = 0, b = C$ is for the case when an instruction loads the same value C throughout the entire loop execution.

For the speculative run-time system to support linear value prediction speculation, the loaded address A must be known. In many cases, A is computed by a complicated expression that cannot be evaluated at compile time. PEP overcomes this problem through loop peeling. The first *peeled* loop iteration captures the effective address A for each speculatively linear load instruction LD and saves it into the data structure called *linear-prediction table* alongside coefficients m, b. Before each iteration I, a worker process runs the iteration and enforces the prediction by initializing its private memory to concur with the linear-prediction table: for each (A, m, b) in the table, it stores $m \times I + b$ to address A. After each iteration I, a worker validates the next iteration's prediction: for each (A, m, b) in the table, it loads V' from A and triggers misspeculation if $V' \neq m \times (I + 1) + b$.

Figure 5 shows how PEP transforms a sequential program (5(a)) into a parallel version (5(b)) using context sensitive speculation. Assuming that the profiler detects that the load instruction `LD1` is speculatively linear in `L` under context `C1`. The profiler can compute m and b of `LD1` under `C1` as well, but the loaded address can only be found at run-time. In the parallel version, the loaded address is updated in the row corresponding to `C1` and `LD1` in the linear prediction table when `foo_peeled` is called from the peeled loop under context `C1`.

When parallel execution of loop `L` begins, the worker process for each iteration updates its private memory according to the linear prediction table. When `LD1` is executed under context `C1`, the effective address of `LD1` is checked with the one in the linear prediction table to ensure that the address remains invariant across the loop execution. At the end of each iteration, the worker process checks if the values stored in the effective addresses for the speculated load instructions match the predicted values for the next iteration, $m \times (I + 1) + b$.

B. Memory Dependence Speculation

PEP's memory dependence profiler employs a *shadow-memory* based profiler to trace memory accesses within the profiler. For each dynamic instruction that writes to memory, the profiler records the execution context and the iteration count of the target loop as metadata. Note that only the execution context is recorded into metadata, the write instruction itself is not recorded. Since the client of the profiling result is automatic parallelization, the dependence relationship between instructions syntactically included in the target loop is the only interest. In the example of Figure 4, the dependence from instruction 9 to 7 is formulated by the execution of instruction 4 in subsequent iterations of loop L. However, as both 9 and 7 are not syntactically included in loop L, a dependence between them is not of concern to the compiler when it is attempting to parallelize loop L. What the compiler is concerned with is a loop-carried self-dependence formulated across instruction 4, which results from the 9 to 7 dependence. This information

[1]MTX memory versioning removes all false dependences [46] and thus they are not displayed.

```
L:
  …
  // context C1
  foo();
  …
```

```
foo() {
  …
  // load LD1
  x = *p;
  …
}
```

(a)

```
  // Peeled loop
  …
  context = C1;
  foo_peeled();
  …
  verify();
L:
  // Begin iteration I
  for (Row: LinearPredTable)
    Row.addr = Row.m*I+Row.b;
  …
  context = C1;
  foo();
  …
  // End itertion I
  verify();
```

```
foo_peeled() {
  …
  if (context == C1)
    GETROW(C1, LD1).addr = p;
  x = *p;
  …
}

foo() {
  …
  if (context == C1 &&
      GETROW(C1, LD1).addr != p)
    misspeculation();
  x = *p;
  …
}

verify() {
  for (Row: LinearPredTable)
    if (*Row.ptr !=
         Row.m*(I+1)+Row.b)
      misspeculation();
}
```

(b)

Figure 5: Example showing the original code (a) and transformed code (b) for linear value speculated code. Bold lines represent instrumentation added to support speculation.

tells the compiler that each invocation of instruction 4 cannot run in parallel.

Memory dependences are logged at each dynamic execution of an instruction that reads from memory. If a load instruction reads a value from the memory address A, the profiler reads the metadata from the shadow memory corresponding to A and computes the dependence information. Dependences are characterized by the execution context of the write event, the execution context of the read event, the instruction that reads the memory, and whether the dependence is loop-carried or not. Unlike memory write instructions, memory read instructions are traced along with execution contexts. The motivation behind this is that if linear value prediction speculation is applicable to the memory read instruction, the dependence formulated around the read instruction can be speculatively removed.

Memory dependence speculation inserts expensive validation instructions into the parallel region. Context-sensitivity enables more aggressive speculation and therefore increases run-time overhead. Recall the example of Figure 4. If context-sensitive speculation is not supported (Figure 4(c)), no validation instructions are required simply because parallelization is not applicable. If context-sensitive speculation is supported (Figure 4(d)), validation instructions need to be added to all memory operations within the `add1` function, as the dependence between callsites of the function is speculatively removed. In other words, context-sensitive memory dependence speculation improves the applicability of speculative parallelization at the cost of additional run-time overhead. This motivates the performance optimizations to the run-time system, which are described in the next section.

V. Optimizing the Run-time System Supporting Speculative Parallelization

As mentioned in Section II, Multi-threaded transactions (MTX) have been proposed to enable speculation in conjunction with pipeline parallelization. MTX is based on transactional memory systems that observe the order of memory operations to ensure that they were executed in an order consistent with sequential execution and can be used to support memory dependence speculation [28], [65].

A. *Multi-threaded Transactions*

Typical transactional systems are designed for transactions that occur within a single thread. This leaves them incompatible with pipeline parallelization, which distributes work from a single loop iteration, and thus a single traditional transaction, into multiple pipeline stages that are each executed in their own thread. A multi-threaded transaction has one sub-transaction (subTX) per stage, which exists in it's own private memory space. Each thread participating in a multi-threaded transaction opens its own subTX to maintain speculative state.

An MTX system requires two features to enable pipeline execution and multi-threaded atomicity. The first is called *uncommitted value forwarding*. Uncommitted value forwarding ensures that the results of all the stores executed in a subTX are visible to later subTXs, even if the stores are executed speculatively. The other feature is *group transaction commit*. This ensures that speculative work done in different subTXs inside a single transaction are either committed or discarded altogether.

To implement uncommitted value forwarding, the existing software-only implementation of MTX (SMTX [46]) communicates the address-value pair of every store operation executed in a subTX to later subTXs. Additionally, to enable group transaction commit, SMTX forwards the address-value pair of every store and speculative load operation executed in a worker process to a separate *commit* process, a process that maintains the committed, non-speculative state. Once a transaction is closed, the commit process determines if a conflict has occurred by sequentially re-executing memory operations within the transaction and comparing results with the value reported from the worker process. If there is no conflict, the transaction is committed to non-speculative memory. Otherwise, the commit unit flushes any transaction after the transaction with the conflict and then restarts the flushed transactions with a copy of the correct, non-speculative memory.

B. *PEP's Optimizations to SMTX*

The main problem with the existing SMTX implementation is that it performs inter-process communication for every store and every speculative load in the parallelized loop, which adds substantial run-time overhead that may outweigh the gains from parallelization. PEP optimizes the SMTX implementation to reduce the run-time overhead and enable scalable performance improvements, even with aggressively speculated programs.

The optimizations are based on two key insights. First, for each subTX, it is sufficient to forward its final memory state to later subTXs to support uncommitted value forwarding. The same idea applies when subTXs communicate the values to be committed to the commit process. Second, checking that the pipeline scheduling, i.e. data does not flow from later subTXs to earlier subTXs, and parallel stage, i.e. there is no loop carried dependence in a parallel subTX, assumptions are not violated is sufficient to validate memory dependence speculation.

Utilizing these insights, the optimized run-time system reduces the total number of communications and the total number of bytes communicated. For each dynamic memory operation, the optimized run-time simply updates shadow metadata instead of issuing inter-process communication. Shadow memory is private to each subTX and cleared at the beginning of the subTX for each iteration. Each byte of the program's memory is associated with 2-bits of metadata in the shadow memory. These bits, *read* and *write*, indicate the state of the byte during the subTX execution and indicate to the commit process what actions it must perform. If *read* is set, the commit process must perform speculation checks. If *write* is set, the new value must be forwarded to later subTXs and the commit process. Note that if a subTX writes the byte before reading, it will never set the *read* bit as the subTX is then simply reading the value it wrote, which does not need to be checked.

Supporting Uncommitted Value Forwarding: With the optimized run-time system, inter-process communication happens only at subTX boundaries. At the end of the subTX, all pages that have been accessed during the subTX execution are forwarded to later subTXs, along with their shadow pages. When the later subTX receives the pages, it scans the shadow pages to find the values in the original pages that have been written by the earlier subTX and updates its private memory accordingly. Uncommitted value forwarding is accomplished via this update. Note that if a page is accessed but not written to, it is not forwarded to later subTXs as there are no updated values that need to be updateed.

Supporting Group Transaction Commit: Accessed pages and their corresponding shadow pages are forwarded to the commit process from each subTX. The commit process verifies pages communicated from each subTX in order and maintains shadow memory that holds the ID of the subTX that last wrote to each byte of memory. If the read bit is set in the metadata for a byte, the commit process checks if misspeculation occurred. Misspeculation occurs

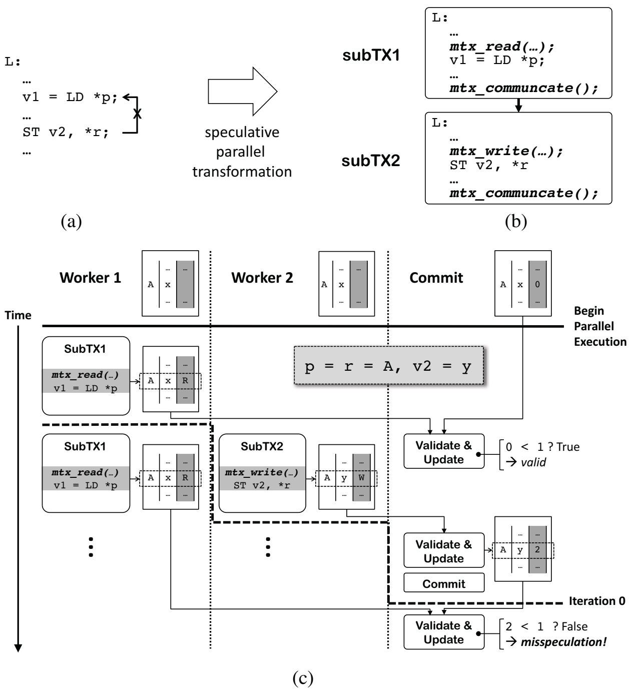


Figure 6: Example demonstrating how the commit process detects misspeculation. (a) Parallelization target loop. (b) Multi-threaded loop using the optimized run-time system. `mtx_read` takes arguments (address, size) of the value of the corresponding load instruction, while `mtx_write` takes arguments (address, size, value) of the corresponding store instruction. (c) A time line of parallel execution. Rectangular boxes next to subTXs represents the memory state of each process at a given time. The shaded column indicates shadow memory for each byte.

if the last write ID is greater than the reader ID, which represents a violation of the pipeline scheduling assumption. Additionally, in the case of a parallel stage, misspeculation can also occur if the last write ID is equal to the reader ID, which indicates an illegal loop carried dependence within the parallel stage. If the write bit is set in the metadata for a byte, the commit process updates its own memory with the written value and writes the subTX ID to the shadow memory. If no misspeculation occurred in the transaction, the commit process writes its memory state to the global state. This write represents group transaction commit. Note that if a page is read but not written to, subTXs send only the address of the page to the commit process, as an optimization, since the actual values are not needed.

C. An example of Misspeculation Detection

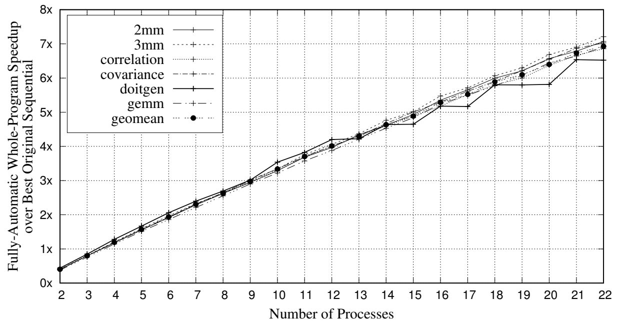
Figure 6 demonstrates how the optimized run-time system discovers memory dependence misspeculation. Figure 6(a) shows a parallelization target loop. With the assumption that the loop-carried dependence between the store and the load in Figure 6(a) can be speculatively removed, pipeline parallelization is applicable to the loop. Figure 6(b) is a multi-threaded loop using the optimized run-time system. As the dependence between the store and the load is speculatively removed, `mtx_read` and `mtx_write` calls are inserted before the load and the store. `mtx_communicate` is called at the end of iteration to forward accessed and shadow pages to the later subTXs and the commit process.

Figure 6(c) is a schematic time line of parallel execution where misspeculation occurs. Pointers `p` and `r` both point to address `A` throughout the loop execution. For each execution of SubTX1 from Worker1, `mtx_read` sets a *read* bit in the shadow byte of `A`. This information is communicated to the commit process, and the commit process checks if the byte is not written by the later subTX. For iteration 0, the commit process verifies that the load in subTX1 is valid; the shadow memory maintained by the commit process shows that metadata for address `A` is `0`, indicating the value stored in `A` is a live-in. As the commit process handles each subTX in sequential order, pages forwarded from subTX2 of iteration 0 are processed next. The forwarded information indicates that address `A` is written by subTX2, so the commit process updates its private memory according to the value written by subTX2. In addition, the commit process sets its metadata for address `A` to `2`, indicating that subTX2 writes the address. Next, the commit process evaluates pages from subTX1 of iteration 1 and detects misspeculation. Address `A` is read by subTX1, but metadata indicates that the address is written by subTX2 if the program follows the sequential order. This violates the pipeline partitioning assumption, and accordingly, the commit process flags misspeculation.

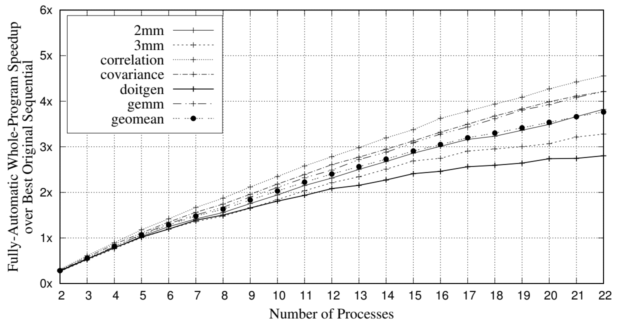
VI. Evaluation

PEP is evaluated on a shared-memory machine with four 6-core Intel Xeon X7460 processors (24 cores total) running at 2.66 GHz with 24 GB of memory. It runs 64-bit Ubuntu 9.10. The PEP compiler is implemented in the LLVM compiler framework [21].

PEP is evaluated with two open-source C programs: the Lua script interpreter version 5.2.3 and the Perl script interpreter 5.20.1. Although PEP is applicable to any kind of program, we focused on script interpreters because this domain demonstrates the usefulness of PEP most clearly. The PEP compiler specialized each program against 6 input scripts, then parallelized the specialized interpreter. Neither interpreter was modified, though Perl was compiled with non-default configuration options (see Section VI-B). As we wanted to show that PEP can achieve parallel speedup with script interpreters if the input script has scalable parallelism, we chose 6 programs from Polybench [38] that are known to be amenable to DOALL parallelism [19] and reimplemented them as Lua and Perl scripts. Kim et al. [19] applied speculative automatic parallelization to the C version of the scripts used in the evaluation, and achieved about $12\times$ geomean speedup across 6 programs on a commodity machine using 24 threads. The C versions result in better



(a) Lua-5.2.3



(b) Perl-5.20.1

Figure 7: Whole-program speedup of the automatically specialized and parallelized code, compared to the sequential, unspecialized version compiled with `-O3`. `Number of Processes` counts the number of parallel workers excluding the commit process.

speedup than the specialized interpreter versions as they are much simpler and easier for a compiler to analyze: no more than 1 dependence needs to be speculatively removed to parallelize the C versions of the programs. This number is much smaller than the number of speculated dependences presented in Table I. Even with these disadvantages, PEP manages to be comparable to Kim et al.

Table I describes the six input scripts for the Lua and Perl interpreters that were used to test PEP. In each case, the main interpreter loop has been specialized and parallelized. Note that the main interpreter loop of Perl contains only 8 LLVM IR instructions due to the use of indirect function calls indexed by the instruction OP-code. Without specialization, this loop would be exceedingly difficult to parallelize. By exploiting fixed inputs, PEP is able to build and then parallelize a specialized loop for each input.

A. Performance Results

Figure 7 presents whole-program speedup. These speedups are normalized against the sequential, unspecialized version of the Lua and Perl interpreters compiled with *clang -O3*. The majority of the performance improvement is due to parallelization. Specialization provides up to 24% and 19% sequential improvement over Lua and Perl per-

Input Script	P'loops	Coverage(%)	Size (LLVMIRs)	Context-Insensitive		Context-Sensitive		Communicated Bytes	
				Spec. Deps	Expected Speedup	Spec. Deps	Expected Speedup	w.o. Opti (MB)	w. Opti (MB)
Lua-5.2.3 (20,258 LOC, Interpreter main loop: 2,499 LLVMIRs)									
2mm	2	96.00%	1,028 / 973	21,884	1.00×	52,805	16.94×	2,772,879.8	36.3
3mm	3	96.61%	973 / 973 / 973	31,442	1.00×	74,769	17.49×	2,773,429.2	38.3
correlation	1	95.17%	1,224	16,650	1.00×	40,512	11.14×	4,978,648.1	172.3
covariance	1	93.14%	1,231	16,650	1.00×	40,220	14.12×	4,986,986.7	144.1
doitgen	1	95.58%	1,337	52,006	1.00×	134,561	13.78×	3,222,117.3	73.4
gemm	1	94.43%	2,229	19,176	1.00×	48,302	16.56×	2,856,902.9	38.2
Perl-5.20.1 (296,166 LOC, Interpreter main loop: 8 LLVMIRs)									
2mm	2	96.22%	927 / 1,087	12,507	1.00×	67,216	13.84×	6,503,153.9	288.9
3mm	3	95.17%	1,140 / 1,063 / 927	18,627	1.00×	121,505	12.18×	6,573,151.9	288.5
correlation	1	94.90%	1,427	12,477	1.00×	95,273	5.54×	13,163,604.7	278.1
covariance	1	90.73%	1,562	12,168	1.00×	62,955	10.73×	13,186,003.5	5256.9
doitgen	1	95.28%	1,332	24,464	1.00×	193,938	6.29×	6,439,570.3	426.1
gemm	1	89.44%	2,209	8,450	1.00×	75,646	12.26×	7,668,543.6	258.7

Table I: Execution characteristics of each interpreter and static input: *P'loops* denotes the number of loops that have been parallelized after specialization. *Coverage* denotes the fraction of the runtime spent in the parallelized loops compared to total program execution time. *Size* denotes the size of the parallelized loops in the sequential version (i.e. size of IPLS specialized loops that are parallelized), in units of LLVM IR instructions. *Context-Insensitive* and *Context-Sensitive* show the results without/with context-sensitive speculation support, respectively. *Spec. Deps* denotes the number of edges that are speculatively removed from the program dependence graph. *Expected Speedup* denotes the speedup estimated by the parallelization planner, assuming 24 cores in the target system. *Communicated Bytes* denotes the total number of bytes communicated between processes during parallel execution. *w.o. Opti* and *w. Opti* show the results without and with the run-time optimization presented in the paper, respectively.

formance, respectively. These improvements are consistent with IPLS [33].

The DOALL parallelism in the input scripts becomes pipeline parallelism in all 12 specialized programs (6 for each Lua and Perl). DSWP extracts a two-stage pipeline featuring a leading sequential stage and a trailing parallel stage. The leading sequential stage contains code to control the loop execution; the specialized loop is no longer a simple counted loop, its control predicate is computed from two complex data structures, the loop control variable and bound value from the input script, loaded from memory. The parallel stage handles the main workload of the input script. The presence of a parallel stage is consistent with the DOALL parallelism in the input script. As the parallel stage contains the majority of each iteration's execution time, this parallelization scales well. This confirms that IPLS folds the parallelism within the input script into the specialized program.

Harnessing DSWP parallelism with a parallel stage is only achievable with the support of context-sensitive speculation. Comparing the *Context-Insensitive* and *Context-Sensitive* columns in Table I shows the effect of context-sensitive speculation. Each column represents the number of speculatively removed dependences and expected speedup computed by the parallelization planner described in Section III, without and with context-sensitive speculation support, respectively. As shown in the table, context-sensitive speculation removes more unlikely-manifesting dependences from the program dependence graph. Without context-sensitive speculation the expected speedup is 1.00× across all benchmarks, which means that no speedup is expected from the given parallelization plan. Even though context-insensitive speculation removes many dependences, the loop is not parallelizable at all until the last dependence that prevents parallelization is removed.

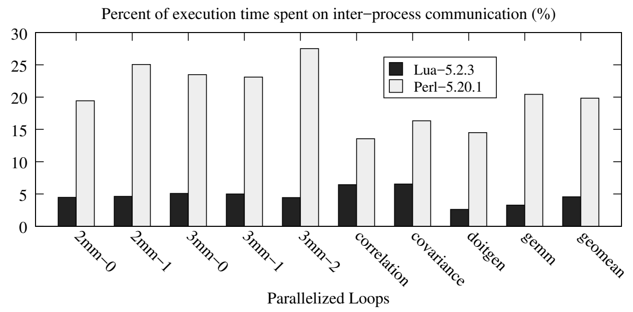


Figure 8: Percentage of the parallel execution capacity that parallel workers spend on inter-process communication. The number is averaged across all parallel worker processes.

Table I also shows the effect of the optimized run-time. The table shows that the total number of bytes communicated between processes is reduced by several orders of magnitude with the proposed optimized run-time system. This optimization is critical in achieving speedup with speculatively parallelized programs. Without the optimized run-time system, the benchmarks listed in Table I incur *at least* 50× slowdown rather than the speedup shown in Figure 7.

Across all scripts, parallelization provides greater speedup on Lua than on Perl. This is due to the higher communication overhead in Perl. Figure 8 presents communication overhead as a percentage of the execution capacity that parallel workers spend at subTX boundaries when using 24 processes. These percentages are normalized to the total computation capacity of the parallel invocation (in core-seconds), i.e., the number of cores times the duration of the invocation. The figure shows that Lua spent a geomean of 4.55% of its execution at subTX boundaries while Perl spent 19.84%. As Perl consumes more memory than Lua to run the same algorithm (across 6 programs, the sequential version of Perl consumes 2.23× more memory by geomean than Lua), Perl suffers from higher communication overhead.

B. Current Limitations and Future Work

While speculation does greatly reduce the reliance on analysis, both the analysis and speculative logics must be able to handle complex dependencies. Adding context sensitivity solved a large class of dependencies found in interpreters but more work remains to be done.

While no modification has been made to the interpreter source code, we configured Perl with the `PURIFY` and `NO_PERL_PRESERVE_IVUV` compile time options. The `PURIFY` flag compiles Perl with the C's default implementation of `malloc` and `free`, rather than a specially crafted implementation. Custom memory allocators introduce many complex dependences that are hard to analyze; whereas PEP's analysis understands common library functions, such as `malloc` and `free`, and is thus able to disprove dependences involving them. The `NO_PERL_PRESERVE_IVUV` flag disables an interpreter optimization for script values that are assumed to be integers. Without these flags, the specialized code includes additional features that are difficult to analyze.

As part of future work, we wish to parallelize the Python [44] interpreter but expect to run into difficulties with the way Python handles integer overflow and asynchronous events. Adding analysis and speculative logic to systematically handle such dependencies is part of future work to turn PEP into a complete system.

Recall that the specialization process introduces dependences that change DOALL parallelism in input scripts into pipeline parallelism in the specialized program, primarily due to simple values becoming complex data structures. If the input script contains complex forms of parallelism, e.g., pipeline parallelism, more non-trivial dependences manifest. Such dependences currently prevent PEP from achieving parallel speedup. Better analysis and logic to handle the way that interpreters store values as complex data structures and exploiting complex forms of input parallelism will also be investigated in future work.

VII. Related Work

Parallelizing Scripting Language Execution: There are a variety of parallel libraries or parallel language extensions [10], [23], [32], [41], [42], [49], [50] for scripting languages that allow the programmer to take advantage of multiple processors by manually parallelizing their code. Manual parallelization, however, is a toilsome and error-prone process, and is best avoided by leveraging automatic parallelization.

There are several techniques that were developed for automatically parallelizing sequential script programs. In [25], Ma et al. build a framework that uses dynamic dependence analysis to identify parallelizable tasks in R programs and parallelizes the execution accordingly. This technique is able to parallelize loops and function calls, but only when there are absolutely no dependences. Talbot et al. present Riposte [55], a runtime system that is able to dynamically discover and extract sequences of vector operations from arbitrary R code. These sequences can be fused to eliminate unnecessary memory traffic and compiled to exploit SIMD units as well as multiple cores. Mehrara et al. [27] enable dynamic speculative parallelization of Javascript programs by proposing ultra-lightweight software speculation mechanism. Though these techniques are able to successfully parallelize some scripts, they require manual changes or extensions to the interpreter or JIT. Thus porting them to other scripting languages requires large amounts of programming effort. In comparison, the technique proposed in this paper can be seamlessly applied across different script interpreters.

Program Specialization: C-Mix [4], [26] and Tempo [11] are program specializers for C programs. However, those specializers require two types of user annotations to work; one to find expressions that depend solely on program invariants, and the other to direct code generation policy to produce efficient programs. DyC [17] is a program specializer focused on minimizing code generation overhead. DyC partly automate the specialization process by using a tool called Calpa [30], which generates annotations for DyC. However, Calpa only provides annotations for code generation policy, thus DyC still resorts to user hints to find static expressions.

Some program specializers are fully automatic and do not require any user annotations [5], [52]. However, they cannot be used as an enabling transformation for automatic parallelizing compilers as they perform specialization at runtime.

Automatic Parallelization: There has been a large body of work on automatic parallelization. Some techniques [8], [36], [47] rely solely on static program analysis to identify the applicability of parallel transformations. Still, imprecision and fragility of static analyses limit the applicability of automatic parallelization. Several techniques have been proposed to overcome this limitation by resorting to the programmer's help [7], [14], [20], [39], [56], [59], [62], [64]. However, it is unsafe and error-prone to ask programmers to manually annotate or inspect complex programs. The problem is worse for specialized script interpreters, which are automatically generated by the compiler.

Speculation alleviates the limitations of static analysis without manual intervention. Although most automatic speculative parallelization systems require specialized hardware, there are techniques that do not require any hardware extensions and are applicable to commodity machines [18], [19], [57], [58]. In the CorD execution model [57], [58], each loop iteration is separated into a prologue and epilogue, which are executed sequentially by the main thread, and the body, which is executed speculatively. Privateer [18] is the first fully automatic system that supports speculative reduction/privatization and targets general purpose programs with complex control-flow and irregular data structures. Cluster Spec-DOALL [19] proposed an automatic parallelizing compiler and run-time system to enable speculative DOALL execution on commodity clusters. Unlike the system proposed in this paper, none of these techniques are capable of context-sensitive speculation.

Several works focus on proposing software run-time systems to support speculative parallelization. LRPD [48] and R-LRPD [12] supports speculative DOALL execution by validating the absence of loop-carried dependences at the end of parallel execution. Cintra and Llanos [9] proposed a run-time system that performs *eager* memory management where each speculative operation checks if misspeculation occurred and updates the non-speculative memory directly when there is no misspeculation. STMLite [28] is based on software transactional memory [13], [29], [51], [53]. It optimizes performance by providing enough functionality to support speculative loop parallelization without implementing the whole spectrum of transactional memory features. None of these techniques support multi-threaded atomicity, which is necessary to enable speculative pipeline parallelization. SMTX [46] resolved this problem, but has substantial performance overhead for heavily speculated programs, as described in Section V.

VIII. Conclusion

This paper proposes a fully automatic technique to exploit parallelism within scripts that is universally applicable to multiple scripting language interpreters. By coupling program specialization with advanced speculative parallelization techniques, parallelism within scripts can result in parallel speedup. PEP, the prototype implementation of the proposed technique, has been evaluated against two widely-used open-source script interpreters with 6 input scripts, which describe linear algebra kernel with latent parallelism, each yielding a geomean speedup of 5.10× over the best sequential version. This proves that the approach proposed in this paper is both feasible and worthy of further investigation.

IX. Acknowledgments

We thank the Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Science Foundation (NSF) through Grants OCI-1047879, CCF-1439085, and CNS-0964328. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the Liberty Research Group and do not necessarily reflect the views of the NSF. This work was carried out when the authors were working at Princeton University.

References

[1] Http://code.google.com/p/v8.

[2] Http://luajit.org.

[3] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The hiphop virtual machine," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. ACM, 2014.

[4] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, University of Copenhagen, 1994.

[5] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI*, 2008.

[7] M. J. Bridges, "The velocity compiler: Extracting efficient multicore execution from legacy sequential codes," Ph.D. dissertation, Department of Computer Science, Princeton University, November 2008.

[8] 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*, ser. CGO '12. ACM, 2012.

[9] M. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Transactions on Parallel Distributed Systems*, June 2005.

[10] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[11] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volansche, "A uniform approach for compile-time and run-time specialization," in *Selected Papers from the International Seminar on Partial Evaluation*, 1996.

[12] F. H. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD test: Speculative parallelization of partially parallel loops," in *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, 2002.

[13] D. Dice and N. Shavit, "Understanding tradeoffs in software transactional memory," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. IEEE Computer Society, 2007.

[14] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior oriented parallelization," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2007.

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, July 1987.

[16] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. ACM, 2009.

[17] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers, "Annotation-directed run-time specialization in C," in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1997.

[18] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August, "Speculative separation for privatization and reductions," *Programming Language Design and Implementation (PLDI)*, June 2012.

[19] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August, "Automatic speculative doall for clusters," *International Symposium on Code Generation and Optimization (CGO)*, March 2012.

[20] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2004.

[22] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: a TLS compiler that exploits program structure," in *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[23] L. Lu, W. Ji, and M. L. Scott, "Dynamic enforcement of determinism in a parallel scripting language," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014.

[24] Lua, http://www.lua.org/.

[25] X. Ma, J. Li, and N. Samatova, "Automatic parallelization of scripting languages: Toward transparent desktop parallel computing," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007.

[26] H. Makholm, "Specializing C — an introduction to the principles behind C-Mix/II," University of Copenhagen, Department of Computer Science, Tech. Rep., 1999.

[27] M. Mehrara, P. C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 87–98.

[28] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[29] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, Sept 2008.

[30] M. Mock, M. Berryman, C. Chambers, and S. Eggers, "Calpa: A tool for automating dynamic compilation," in *Proceedings of the Second Workshop on Feedback-Directed Optimization*, November 1999.

[31] S. C. Müller, G. Alonso, A. Amara, and A. Csillaghy, "Pydron: Semi-automatic parallelization for multi-core and the cloud," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '14. USENIX Association, 2014.

[32] Multicore, http://www.rforge.net/doc/packages/multicore/multicore.html.

[33] T. Oh, H. Kim, N. P. Johnson, J. W. Lee, and D. I. August, "Practical automatic loop specialization," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, 2013.

[34] G. Ottoni and D. I. August, "Communication optimizations for global multi-threaded instruction scheduling," in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

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

[36] G. Ottoni, R. Rangan, N. Vachharajani, and D. I. August, "Decoupled software pipelining: A promising technique to exploit thread level parallelism," in *Proceedings of the 4th Workshop on Explicitly Parallel Instruction Computing Techniques*, March 2005.

[37] Perl, http://www.perl.org/.

[38] L.-N. Pouchet, "PolyBench: the Polyhedral Benchmark suite," http://www-roc.inria.fr/˜pouchet/software/polybench/download.

[39] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August, "Commutative set: A language extension for implicit parallel programming," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[40] P. Prabhu, T. B. Jablin, A. Raman, Y. Zhang, J. Huang, H. Kim, N. P. Johnson, F. Liu, S. Ghosh, S. Beard, T. Oh, M. Zoufaly, D. Walker, and D. I. August, "A survey of the practice of computational science," *Proceedings of the 24th ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[41] PyMPI, http://pympi.sourceforge.net/.

[42] Pypar, https://code.google.com/p/pypar/.

[43] Pypy, http://pypy.org/.

[44] Python, http://www.python.org/.

[45] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2005.

[46] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative parallelization using software multi-threaded transactions," in *Proceedings of the Fifteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[47] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.

[48] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Transactions on Parallel Distributed Systems*, February 1999.

[49] RMPI, http://www.stats.uwo.ca/faculty/yu/Rmpi/.

[50] A. Rubinsteyn, E. Hielscher, N. Weinman, and D. Shasha, "Parakeet: A just-in-time parallel accelerator for python," in *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'12. USENIX Association, 2012.

[51] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural support for software transactional memory," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. IEEE Computer Society, 2006.

[52] A. Shankar, S. S. Sastry, R. Bodík, and J. E. Smith, "Runtime specialization with optimistic heap analysis," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005.

[53] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. ACM, 1995.

[54] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede approach to thread-level speculation," *ACM Transactions on Computer Systems*, February 2005.

[55] J. Talbot, Z. DeVito, and P. Hanrahan, "Riposte: A trace-driven compiler and parallel vm for vector code in r," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. ACM, 2012.

[56] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[57] C. Tian, M. Feng, and R. Gupta, "Supporting speculative parallelization in the presence of dynamic data structures," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. ACM, 2010.

[58] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, "Copy or discard execution model for speculative parallelization on multicores," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. IEEE Computer Society, 2008.

[59] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009.

[60] N. Vachharajani, "Intelligent speculation for pipelined multithreading," Ph.D. dissertation, Princeton, NJ, USA, 2008.

[61] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007.

[62] H. Vandierendonck, S. Rul, and K. De Bosschere, "The Paralax infrastructure: Automatic parallelization with a helping hand," in *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2010.

[63] 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*, ser. ICS '09. ACM, 2009.

[64] H. Yu, H.-J. Ko, and Z. Li, "General data structure expansion for multi-threading," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. ACM, 2013.

[65] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.