Automatic Exploitation of Input Parallelism

Taewook Oh

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

Advisor: Professor David I. August

September 2015

Abstract

Parallelism may reside in the input of a program rather than the program itself. A script interpreter, for example, is hard to parallelize because its dynamic behavior is unpredictable until an input script is given. Once the interpreter is combined with the script, the resulting program becomes predictable, and even parallelizable if the input script contains parallelism. Despite recent progress in automatic parallelization research, however, existing techniques cannot take advantage of the parallelism within program inputs, even when the inputs remain fixed across multiple executions of the program.

This dissertation shows that the automatic exploitation of parallelism within fixed program inputs can be achieved by coupling program specialization with automatic parallelization techniques. Program specialization marries a program with the values that remain invariant across the program execution, including fixed inputs, and creates a program that is highly optimized for the invariants. The proposed technique exploits program specialization as an enabling transformation for automatic parallelization; through specialization, the parallelism within the fixed program inputs can be materialized within the specialized program.

First, this dissertation presents Invariant-induced Pattern-based Loop Specialization (IPLS). IPLS folds the parallelism within the program invariants into the specialized program, thereby creating a more complete and predictable program that is easier to parallelize. Second, this dissertation applies automatic speculative parallelization techniques to specialized programs to exploit parallelism in inputs. As existing techniques fail to extract parallelism from complex programs such as IPLS specialized programs, *context-sensitive* speculation and optimized design of the speculation run-time system are proposed to improve the applicability and minimize the execution overhead of the parallelized program.

A prototype of the proposed technique is evaluated against two widely-used open-source script interpreters. Experimental results demonstrate the effectiveness of the proposed techniques.

Acknowledgments

First and foremost, I thank God, my Lord, for his abundant blessings to me. Despite my weaknesses and limitations, God has faithfully guided me through each and every day. I want my life to be used for His purpose and glory.

I was fortunate enough to be advised by Prof. David August during my Ph.D. at Princeton. I thank David for helping me find interesting and intriguing research topics, which made my times here full of excitement. I also thank David for believing in me and encouraging me when I was struggling through my research. David taught me invaluable lessons about how to address challenging problems, how to make constant progress through research, and how to talk persuasively with others about the research findings. It was a great privilege to work with David.

I sincerely appreciate Prof. Brian Kernighan and Prof. Scott Mahlke for reading my thesis and providing me their feedback. It was very lucky to have them as readers of my committee. Brian was an example of a great teacher and software engineer. Helping Brian teach his class as an Assistant in Instruction was both a pleasant and invaluable experience. Scott was the best external collaborator of my team when I was working for Samsung before I come to Princeton. His research inspired me to pursue a Ph.D. with an emphasis on computer architectures and compilers. I would also like to thank Prof. Andrew Appel and Prof. Sharad Malik for their invaluable time as nonreaders of my committee.

I must say that this dissertation would not have been possible without the support from everyone in the Liberty Research Group. I thank Thomas Jablin, Arun Raman, Yun Zhang, Jialu Huang, and Prakash Prabhu for their welcome greetings and guidance for graduate school when I first joined the group. Hanjun Kim gave me lots of help in doing research and settling down in Princeton, for which I am very grateful. I thank Nick Johnson for building a great compiler infrastructure which was an essential component of my research. Feng Liu was a big help to me in my final year when I was searching for a job and going through the graduation process. I thank Stephen Beard, Soumyadeep Ghosh, and Jordan

Fix for being both excellent collaborators and great friends. My days in Princeton were so much fun with you guys. I thank Heejin Ahn, Nayana Prasad Nagendra, Sergiy Popovych, and Hansen Zhang for making the group vibrant. Additionally, I thank Jae W. Lee, Ayal Zaks, and Matt Zoufaly for many interesting discussions and collaborations while they were in the group.

I would like to thank my wonderful friends I have met in Princeton. I thank my fellows in Computer Science and Electrical Engineering department, including Sunha Ahn, Wonho Kim, Kyong Ho Lee, Young-suk Lee, Srinivas Narayana, Chris Park, and Cole Schlesinger, for sharing the pains and joys of research. I thank Daeki Cho, Changhoon Ha, Taehee Han, Hyuncheol Jeong, Jonghun Kam, Hwanho Kim, Hyungwon Kim, Insong Kim, John Kim and Tae-Wook Koh for the coffee/lunch breaks that kept me sane through the years. I especially thank Taehee's family for always helping my family when we were in dire need. I am thankful to everyone in the Korean Graduate Student Association at Princeton and the All Nations Mission Church for their friendship and prayers.

Many thanks to the administrative staff of Princeton University and Department of Computer Science, in particular. I thank Melissa Lawson and Nicki Gotsis for their help as the Graduate Coordinator. I would like to thank the Sieble Scholars program for their recognition and generous support during my fifth year of graduate school. I thank Microsoft and Facebook for the opportunities to do summer internships, which provided me great experience. I had a wonderful time working with Chris Mckinsey, David Tarditi, Guilherme Ottoni, and Bert Maher among others.

My parents, Seongho Oh and Hyunjung Koo, have been great role models for me throughout my life. I have always felt that my success and happiness were their priority. I'm grateful for their unconditional love, support, and numerous sacrifices. I also thank my parents-in-law, Juick Kim and Yeonsim Yoo, for having supported me with their love and encouragement. I thank my two grandmoms, Ms. Oksook Jang and Ms. Jeongsoon Song, for their endless prayer for me to accomplish my Ph.D. I thank all my relatives for

their love and support.

Finally, I thank my wife, Kiyeon Kim, for her patience, understanding, and love. My Ph.D. journey was only possible because of her amazing support and sacrifice. Thank you, love, for being there through thick and thin, and being the best mom for our two sons, Chanjoo and Injoo. No amount of words could express my gratitude towards you!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multi-core processors have become ubiquitous across all levels of computing, from embedded systems to high performance servers. However, applications must expose enough parallelism to utilize multiple cores in order to benefit from these computational resources. Thus, extracting as much parallelism as possible out of programs is key to achieve high performance on today's architectures.

Figure 1.1 illustrates the historical performance scaling of SPEC benchmarks from 1992 to 2015 [86]. The y-axis represents performance normalized across generations of the SPEC benchmark suite, while the x-axis represents the time when the performance was reported. Each point on the graph represents the performance of the benchmarks on different machine configurations. Note that the graph is drawn in log scale.

As shown in the graph, the rate of performance improvement over time has slowed down since 2004. Up until 2004, each new generation of hardware improved the performance transparently to the programmers. This improvement mostly came from increasing clock speed and more instruction-level parallelism (ILP) enabled by microarchitectural enhancement.

Since 2004, however, processor vendors have been faced with the limitations of these traditional approaches. The limitations are two-fold. First, the processor's clock frequency

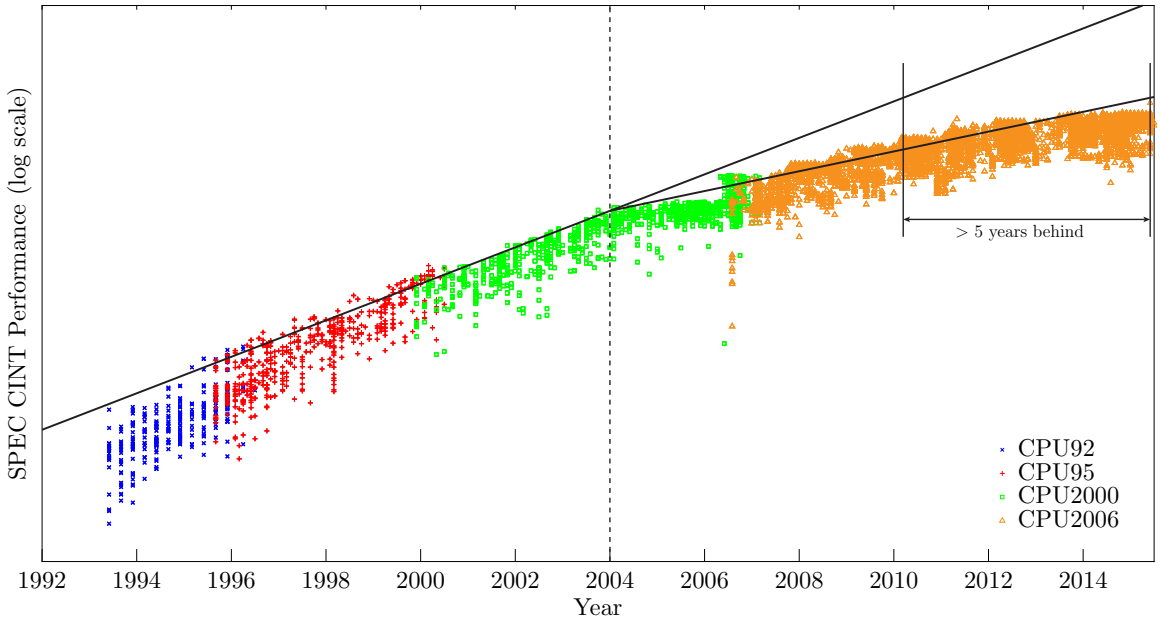


Figure 1.1: Normalized SPEC scores for all reported configuration of machines between 1992 and 2015.

cannot be increased indefinitely because of excessive power usage and thermal dissipation. Second, an ILP wall emerged. Allocating more hardware resources to exploit more ILP is providing diminishing returns. This has led to a fundamental change in processor design. Instead of improving single-core performance, processor manufacturers started to put multiple cores on the same die to utilize abundant transistors resulting from Moore's law [56].

Unfortunately, sequential programs do not benefit from additional cores. The consequence is a substantial decrease in the traditional rate of performance growth, shown in Figure 1.1. To harness multiple cores, programs need to be parallelized to exploit coarse-grained parallelism or thread-level parallelism. However, it is widely acknowledged that writing correct and efficient parallel programs is much harder than writing sequential ones, even for skilled computer programmers [68]. Parallel programs are vulnerable to concurrency bugs like deadlocks or data races, and debugging parallel programs is more tedious than debugging sequential ones. To make a parallel program perform well, programmers must understand not only the program itself but also the underlying hardware architecture.

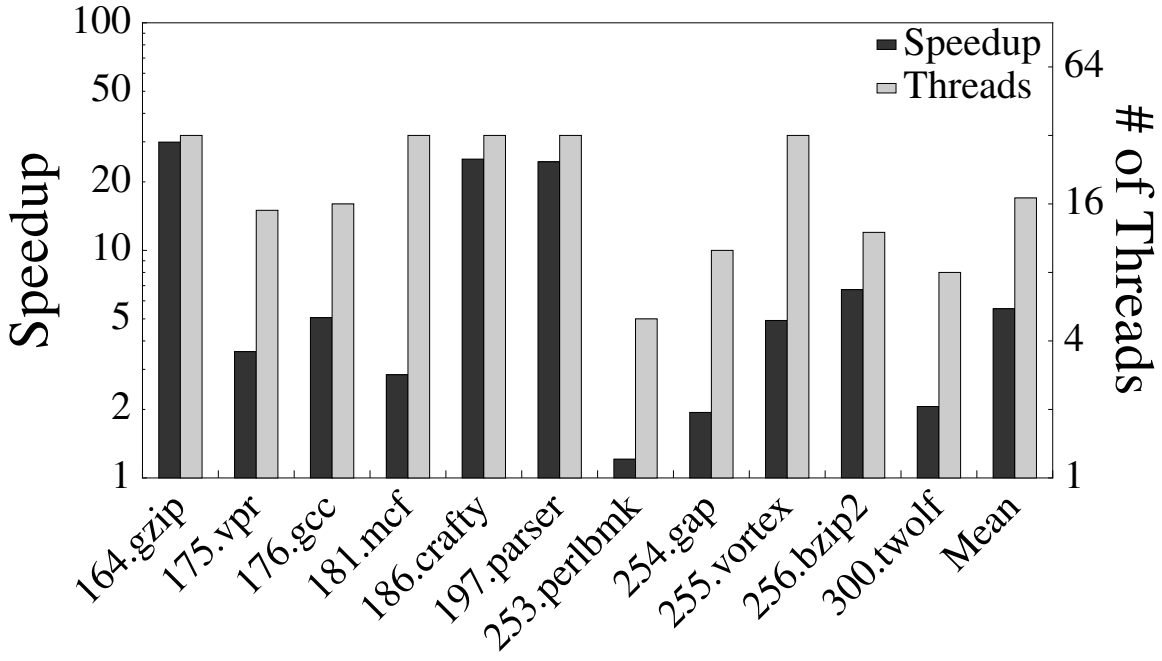


Figure 1.2: Speedup of multiple threads over single threaded execution for SPEC CINT2000 benchmark suite. The speedup numbers are measured on a simulation platform that assumes the existence of core-to-core communication queues and a versioned memory hardware subsystem [10]

Automatic parallelization is an attractive alternative approach to manually writing parallel code. Automatic parallelization allows programmers to focus on the algorithm itself and develop the sequential version of the program. Recent research in automatic parallelization systems [11, 34, 38, 53, 93] shows promising results that may reduce the expense of writing parallel programs. Figure 1.2 shows the speedup that can be obtained by applying an automatic parallelization system to each program in the SPEC CINT2000 benchmark suite, assuming architectural support of core-to-core queues and a versioned memory subsystem [10]. The figure presents the minimum number of threads for which the maximum speedup occurs as well. The system achieves decent speedups for several benchmarks. For example, it improves the performance of 164.gzip by 30$\times$ by exploiting slightly more than 30 threads.

However, existing automatic parallelizing systems are limited in the sense that they do not consider program inputs at all. In some cases, parallelism resides in the inputs of the

program rather than the program itself. Disregarding program inputs altogether limits the applicability of automatic parallelization systems to such programs.

For example, in Figure 1.2, the automatic parallelization system reported a maximum speedup of less than 2× for 253.perlbmk, a perl script interpreter. This is the worst speedup among all benchmarks. The speedup of 253.perlbmk does not scale beyond 5 threads, which is the worst scaling result among all the benchmarks as well. A script interpreter paired with an input script is an example of the case where parallelism exists in the inputs of the program, not in the program itself. Script interpreters are considered inherently sequential because they are, in some sense, incomplete programs with drastically different dynamic behavior depending upon the input script with which they are run. However, once paired with an input script, their behavior becomes much more predictable. If the input script describes a highly parallelizable algorithm and contains parallelism, then the combined interpreter and script contains parallelism. While the performance of script interpreters becomes more and more important as scripting languages are widely adopted among computational scientists [68], current automatic parallelization systems fail to exploit such parallelism and miss a great opportunity for massive performance improvements.

All prior work utilizing input parallelism requires manual modification of either the input or the program that takes the input. Most prior work focused on using parallel programming libraries or parallel language extensions to aid in the parallelization of input scripts [15, 57, 46, 69, 70, 79, 80] However, this approach shares the disadvantages of manual parallelization described above. Other approaches reimplement programs to harness parallelism within the inputs— for example, reimplementing a script interpreter to take advantage of parallelism within the input script [63, 48, 90]. The problem with these approaches is that each program needs to be modified separately, thus the effort put into modifying one program cannot be reused for other programs.

1.1 Dissertation Contributions

This dissertation demonstrates the feasibility of the automatic exploitation of coarse-grained parallelism within fixed program inputs, i.e., inputs that remain constant across multiple program executions. The key enabling insight is to combine program specialization with automatic parallelization. Program specialization optimizes a program with respect to program invariants, including fixed program inputs. The primary effect is performance improvement. Furthermore, program specialization can naturally fold the parallelism within fixed program inputs into the specialized program. Thus, program specialization can function as an enabling transformation for automatic parallelization.

Despite the large volume of continuing research in program specialization [1, 2, 3, 4, 18, 19, 20, 28, 29, 40, 49, 51, 58, 83, 84], prior techniques are ill-suited for adoption as enabling transformations for automatic parallelization. Most of them [1, 3, 4, 18, 19, 20, 28, 29, 40, 49, 51, 58, 83] are based on static analysis to identify specializable parts of the program that depend solely on program invariants. However, limited precision of analysis forces these techniques to resort to user annotations. Some techniques [2, 84] that perform specialization at runtime do not require user annotations. These techniques cannot be adopted because automatic parallelization requires compile-time analysis and code transformation. More importantly, all prior techniques specialize programs against specific values computed by the instructions in the program. To capture and reproduce parallelism within fixed program inputs, specialization needs to be performed on a larger granularity.

This dissertation proposes Invariant-induced Pattern-based Loop Specialization (IPLS). IPLS 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. It is possible that each occurrence of the repeating pattern can be computed independently of all other occurrences. In such cases, specialized loops generated from these patterns will contain loop-level parallelism,

which can then be realized by applying automatic parallelization techniques.

This dissertation presents enhanced automatic speculative parallelization techniques as well. The IPLS specialization process generates aggressively unrolled loops which are hard to reason about, either in compiler analysis or speculation techniques [34, 38, 45, 53, 72, 87, 94, 108]. To overcome this limitation, *context-sensitive* speculation is proposed. Context-sensitivity improves the applicability of automatic parallelization techniques by enhancing the precision of speculative analysis which leads to the extraction of more parallelism from the program.

Finally, this dissertation presents optimizations to the speculative run-time system to minimize performance overhead. Extensive use of speculation is unavoidable to automatically parallelize complex programs including IPLS specialized programs. However, existing run-time systems supporting speculative parallelization issue expensive inter-process communication for each speculated memory operation, offsetting the benefits of parallelization. The optimized run-time system proposed in this dissertation resolves this issue by reducing both the total number of communications and the total number of bytes communicated. This improvement provides parallel speedup even with aggressively speculated programs.

In summary, this dissertation has the following contributions:

- Invariant-induced Pattern-based Loop Specialization (IPLS), the first fully-automatic program specialization technique that is applicable to complex programs such as real-world script interpreters. IPLS not only improves program performance but also integrates input parallelism into the specialized program.

- Design and implementation of context-sensitive speculation techniques. Context-sensitive speculation plays a critical role in discovering parallelism in complex programs.

- Optimizations to the run-time system supporting speculative parallelization, which

enable scalable speedup of extensively speculated parallel programs.

- A fully-automatic technique to harness input parallelism by combining program specialization with automatic speculative parallelization. A prototype implementation is evaluated against two real-world script interpreters and demonstrates the effectiveness of the technique.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 describes background information necessary to understand the techniques proposed in the dissertation. Chapter 3 shows an example that delivers the high-level insight behind these techniques. Chapter 4, published in [60], describes the detailed design and implementation of IPLS. Chapter 5 describes the automatic parallelization of IPLS specialized programs, with a focus on the design and implementation of context-sensitive speculation techniques and optimizations to the speculative run-time system. Chapter 6 presents an experimental evaluation of the proposed techniques. Chapter 7 describes a discussion of existing work related to the techniques proposed in this dissertation. Chapter 8 concludes the dissertation and discusses future work.

Chapter 2

Background

2.1 Program Specialization

Program specialization, sometimes referred to as partial evaluation, optimizes a program for the values that remain invariant across the program execution. These invariant values include known, fixed program inputs that remain constant across multiple executions of the program. Constant propagation can be considered a primitive form of program specialization.

Figure 2.1 shows how program specialization exploits program invariants to optimize the program performance. Figure 2.1(a) is a code snippet of a mean filter program. The size of the filter is determined by the argument `size`. However, if we know that the value of `size` is actually determined by the fixed program input and will not be changed across multiple program runs, function `mean` can be specialized against the fixed value of `size` as shown in Figure 2.1(b). Moreover, program specialization may enable more aggressive optimizations that further improves the program's performance. By applying loop unrolling on top of the specialized code snippet in Figure 2.1(b), the inner loop can be completely removed as shown in Figure 2.1(c).

Many prior program specialization techniques rely on compile-time analysis called

```
void mean(uint8_t* in, uint8_t* out, unsigned size)  {
  unsigned h = size / 2;
  for (i = h ; i < WIDTH-h ; i++) {
    uint16_t sum = 0;
    for (j = 0 ; j < size ; j++)
      sum += in[i + j – h];
    out[i] = sum / size;
  }
}
```

(a)

```
void mean_spec1(uint8_t* in, uint8_t* out)  {
  // Value of size is fixed to 3
  for (i = 1 ; i < WIDTH-1 ; i++) {
    uint16_t sum = 0;
    for (j = 0 ; j < 3 ; j++)
      sum += in[i + j – 1];
    out[i] = sum / 3;
  }
}
```

(b)

```
void mean_spec2(uint8_t* in, uint8_t* out)  {
  // Value of size is fixed to 3
  for (i = 1 ; i < WIDTH-1 ; i++) {
    uint16_t sum = in[i – 1] + in[i] + in[i + 1];
    out[i] = sum / 3;
  }
}
```

(c)

Figure 2.1: Example of program specialization. (a) A code snippet describing a mean filter algorithm. `size` is a runtime parameter. (b) A value of `size` is fixed to 3. A code snippet specialized accordingly. (c) More aggressively specialized code snippet by applying loop unrolling.

binding-time analysis to discover program invariants [1, 3, 4, 18, 19, 20, 28, 29, 40, 49, 51, 58, 83]. Binding-time analysis separates *static instructions* — those which depend solely on program invariants in the program — from *dynamic instructions* — those which may depend on non-invariant values. As static instructions always produce the same value across the program's execution, the program specializer can generate a specialized program by replacing static instructions with the precomputed values. However, as the precision of compile-time analysis is limited, existing approaches necessitate manual user annotations to analyze complex programs.

Once binding-time information is collected, specialized code generation can be performed either at compile-time [1, 3, 4, 18, 19, 20, 40, 49, 51, 83] or run-time [18, 19, 20, 28, 29, 58]. Run-time code generation has an advantage over compile-time code generation because it can exploit run-time constants that are not available at compile-time. However, run-time approaches suffer from high overhead of dynamic code generation [58] and cannot function as an enabling transformation for following compile-time optimizations, including automatic parallelization.

Some specializers are not based on binding-time analysis and do not require user annotations [2, 84]. Instead, they identify constant or hot values at run-time and generate specialized programs on the fly. However, the disadvantages of run-time code generation mentioned above apply to these approaches as well. Further, their scope of detecting program invariants is narrower than that of the compile-time techniques.

2.2 Parallelization Transforms

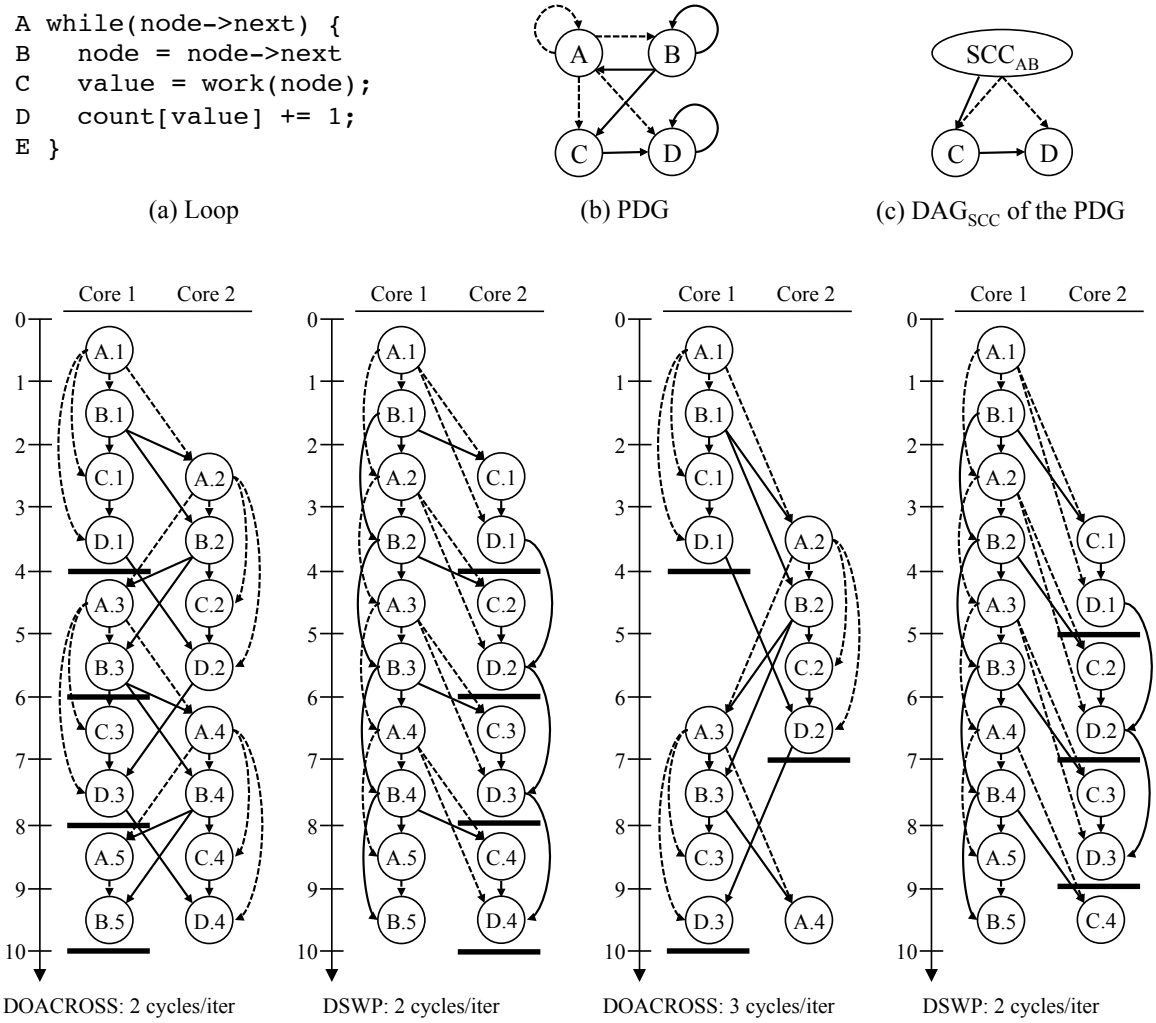
The most basic method of harnessing loop-level parallelism is DOALL parallelization. Iterations of DOALL loops run completely independent of other iterations, thus resulting in scalable parallel speedup. However, DOALL parallelization can only be applied to loops with no loop-carried dependences, often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

DOACROSS [21] and Decoupled Software Pipelining (DSWP) [62] overcome the limited applicability of DOALL parallelization. Both techniques can be applied even when loop-carried dependences exist. DOACROSS schedules the entire loop body of a single iteration on a single thread, while putting distinct iterations in different threads. DSWP divides the loop body into multiple stages and assigns each stage to different threads to create a pipeline — DSWP first builds a DAG_{SCC} of the Program Dependence Graph (PDG) [24] of the target loop by coalescing each strongly connected component (SCC) in the PDG into a single node, then assigns those nodes to stages to identify a pipeline schedule. Loop-carried dependences are modeled as back edges in the PDG, which formulate SCCs.

The different thread partitioning strategies of DOACROSS and DSWP result in a difference in the inter-thread communication pattern of the parallelized program. The partitioning in DOACROSS lets different threads work on different iterations by converting loop-carried dependences into inter-thread dependences. These dependences result in a cyclic communication pattern between threads. If there are N threads, communication from the N-th thread to the first thread is required to handle the dependences from the N-th iteration to the $(N+1)$-th iteration.

In contrast, DSWP's partitioning causes all loop carried dependences to be communicated locally, as instructions from an SCC in the PDG are scheduled to the same thread. However, since the loop body is spread across multiple threads, any intra-iteration dependence that flows between stages must be communicated across threads. Since the stages are arranged into a pipeline, the inter-thread communication of DSWP parallelized programs exhibits an acyclic, or unidirectional pattern, where communication only flows along the pipeline.

Due to its acyclic communication pattern, DSWP is generally more tolerant than DOACROSS to increases in communication latency between threads [97]; DSWP only pays the communication cost once to fill the pipeline whereas DOACROSS must pay the cost on each iteration.



(a) Loop (b) PDG (c) DAG_{SCC} of the PDG

(d) Comm. Latency = 1 cycle (e) Comm. Latency = 2 cycles

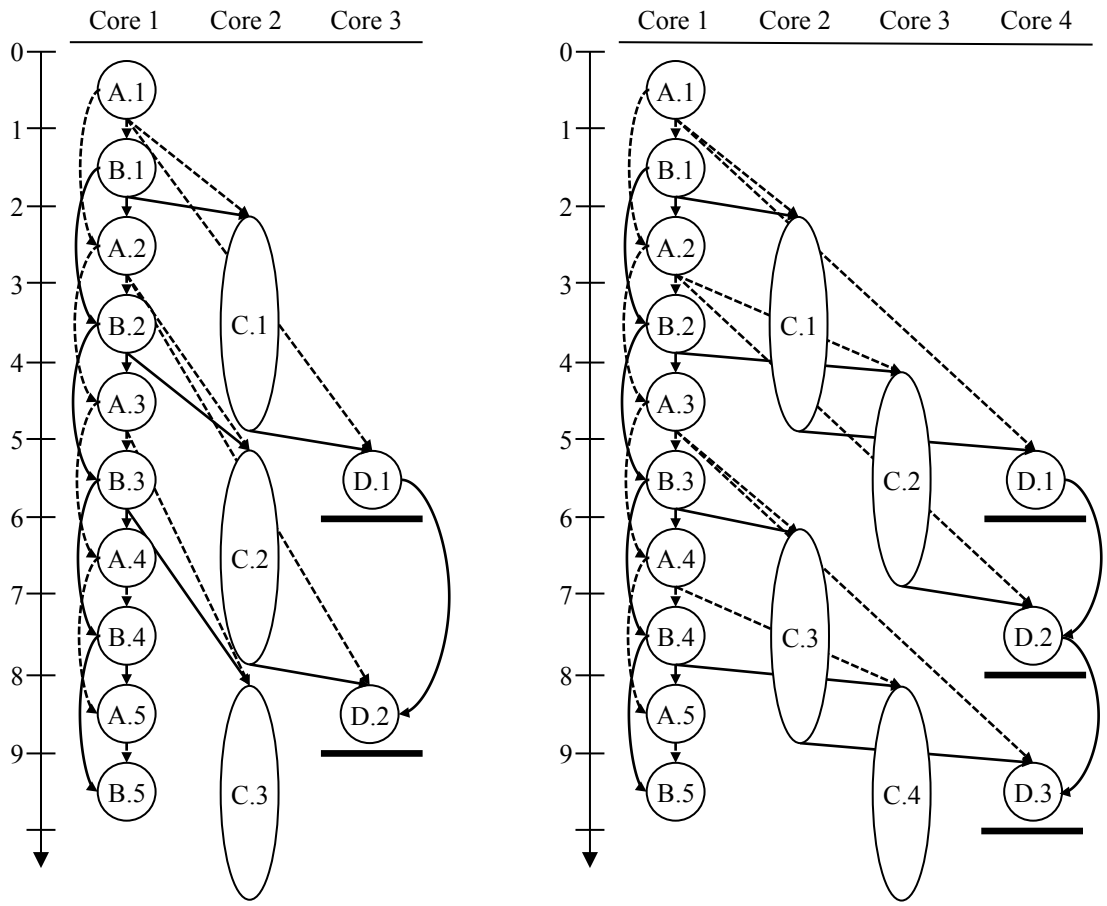
Figure 2.2: DOACROSS and DSWP schedules. A parallelizable loop described in (a) can be parallelized by applying either DOACROSS or DSWP. (b) is the Program Dependence Graph (PDG) of loop (a) and (c) is the DAG_{SCC} of the PDG. Solid lines represent data dependences while dotted lines represent control dependences. (d) shows parallel execution schedules of the loop for DOACROSS and DSWP when communication latency is 1 cycle, while (e) shows the schedules when communication latency is 2 cycles. The letters in the schedules correspond to the lines in (a) and the numbers represent the iteration counts.

Figure 2.2 depicts the difference between DOACROSS and DSWP schedules with an example. Figure 2.2(a) describes a sequential code that can be parallelized with either DOACROSS or DSWP. Figure 2.2(b) and (c) shows the Program Dependence Graph (PDG) of the code and the DAG_{SCC} of PDG. In Figure 2.2(c), nodes A and B in Figure 2.2(b) are merged into a single node SCC_{AB} while nodes C and D form a singleton node. Figures 2.2(d) and (e) illustrate execution plans of DOACROSS and DSWP parallelized versions of Figure 2.2(a), with inter-core communication latency of 1 cycles and 2 cycles, respectively. Each node represents a dynamic instance of a statement in Figure 2.2(a), where the number in each node represents the iteration to which the node belongs. DSWP allocates SCC_{AB} (node A and B) into one pipeline stage and other nodes into another pipeline stage to obtain a balance between stages with a minimum number of stages.

As shown in Figure 2.2(d), both DSWP and DOACROSS generate a schedule that takes two cycles to run each iteration ignoring the pipeline fill time, if communication latency is one cycle. This is a $2\times$ speedup compare to the original sequential execution plan, which takes four cycles per each iteration. However, as illustrated in Figure 2.2(e), DOACROSS only completes one iteration every three cycles, if communication latency becomes two cycles. In contrast, DSWP still completes one iteration every two cycles even with the increased communication latency. Again, DSWP keeps recurrence critical path dependences thread-local, and thus induces acyclic communication pattern. This results in better latency tolerance of DSWP compared to DOACROSS.

Parallel-Stage Decoupled Software Pipelining (PS-DSWP) [75] achieves even better scalability than DSWP. The high-level insight of PS-DSWP is that a DSWP-pipelined stage with no loop-carried dependences can run independently of different iterations of the same stage, and thus can be parallelized in a DOALL fashion. If such a parallel stage exists, DSWP can achieve better scalability by creating a pipeline where the parallel stage dominates loop execution time.

Figure 2.3 illustrates the scalability of a PS-DSWP schedule with a parallel stage. As-



(a) DSWP: 3 cycles/iter

(b) PS-DSWP: 2 cycles/iter

Figure 2.3: Comparison of DSWP and PS-DSWP schedules. (a) DSWP execution plan and (b) PS-DSWP execution plan of Figure 2.2(a) when statement C takes three cycles to execute.

sume that a function call statement C in Figure 2.2(a) now takes three cycles instead of one cycle in the previous example. For this case, to achieve a maximum throughput, DSWP allocates C and D into separate stages and creates a three-stage pipelined schedule, if an additional core is available. Figure 2.3(a) shows the execution plan of the schedule. As shown in the figure, now the second stage becomes a bottleneck and the schedule takes three cycles to complete each iteration. The bigger problem is that even when more cores are available, there is no room to improve the throughput; beyond three cores, additional cores are useless with the schedule.

Unlike DSWP, PS-DSWP can harness additional cores by exploiting the parallel stage. As statement C in Figure 2.2(a) has no loop-carried dependences formulated around it, the second stage of the pipeline shown in Figure 2.3(a) can be a parallel stage. Exploiting this, PS-DSWP generates a schedule where each dynamic instance of the second stage runs in parallel. Figure 2.3(b) illustrates the execution plan of the schedule and shows that PS-DSWP completes one iteration every two cycles.

2.3 Speculative Parallelization

Even with advanced parallelization techniques like PS-DSWP, expected performance improvement is limited by the amount of parallelism extracted from the program. Extracting parallelism is often prevented by imprecise dependence analysis, especially for general purpose programs with irregular data structures and complex control-flows. It is known that these programs have many statically unresolvable dependences that may not manifest at runtime. For example, the PS-DSWP schedule of Figure 2.3(b) is achievable only when the compiler can prove that there are no loop-carried dependences on statement C of Figure 2.2(a). However, in the real-world, such proof is often beyond the capability of the compiler's static analysis even when there are actually no loop-carried dependences formulated around statement C. For such cases, the compiler will end up generating an

inefficiently parallelized version like Figure 2.3(a).

Speculative parallelization [45, 53, 72, 87, 108] can overcome the limits of static analysis. Guided by profiling results, dependences that are unlikely to occur in practice can be speculatively removed, thereby giving the compiler increased freedom to apply parallelizing transforms. However, this freedom comes at a cost. The compiler must insert validation checks to ensure that the dependences that were speculatively removed do not actually manifest at runtime. If the speculative assumptions are violated, the program must signal *misspeculation* and then *recover* to non-speculative state. Recovery is typically handled through a checkpoint and rollback mechanism.

Memory dependence speculation is an important and common type of speculation [17, 34, 38, 42, 44, 45, 64, 88, 98]. Transactional memory systems that observe the order of memory operations to ensure that they were executed in an order consistent with sequential execution can be used to support memory dependence speculation [53, 108]. However, typical transactional systems are designed for transactions that occur within a single thread. This leaves them incompatible with the DSWP transformation, which distributes a single loop iteration, and thus a single transaction, across multiple threads.

To support dependence speculation in conjunction with DSWP, Vachharajani introduces the notion of Multi-threaded Transactions (MTXs) [96]. An MTX may have multiple subtransactions (subTXs), and each thread participating inside an MTX opens its own subTX within the MTX to maintain speculative state.

A MTX system requires to support two features to enable multi-threaded atomicity. The first is called *uncommitted value forwarding*. Uncommitted value forwarding ensures that the results of all the stores executed in earlier subTXs are visible in later subTXs, even if they are executed speculatively. The other feature is *group transaction commit*. This ensures that speculative work done in different subTXs inside an MTX are committed altogether or discarded altogether. Each stage of a DSWP schedule is executed in a separate subTX. As the original MTX proposal requires extensive hardware modifications, and is

therefore not compatible with commodity processors, a software-only implementation of MTX (SMTX [74]) was developed. SMTX maintains speculative and non-speculative state separately using process separation, and a thread for each subTX has the illusion of private memory. A final *commit* thread is used to maintain the committed, non-speculative state. To support *uncommitted value forwarding*, instrumentation is added to each load and store in a subTX to forward stores in prior subTXs to later subTXs within the same MTX. Additionally, to support *group transaction commit*, the instrumentation reports activity of memory operations to the *commit* thread. Once an MTX is closed, the commit thread determines if a conflict has occurred by sequentially re-executing memory operations within the MTX and comparing results with the value reported by the instrumentation. If there is no conflict, the transaction is committed to non-speculative memory. If a conflict has occurred, the commit unit flushes MTXs that come after the MTX with the conflict and restarts the MTXs on the worker threads with a copy of the correct, non-speculative memory.

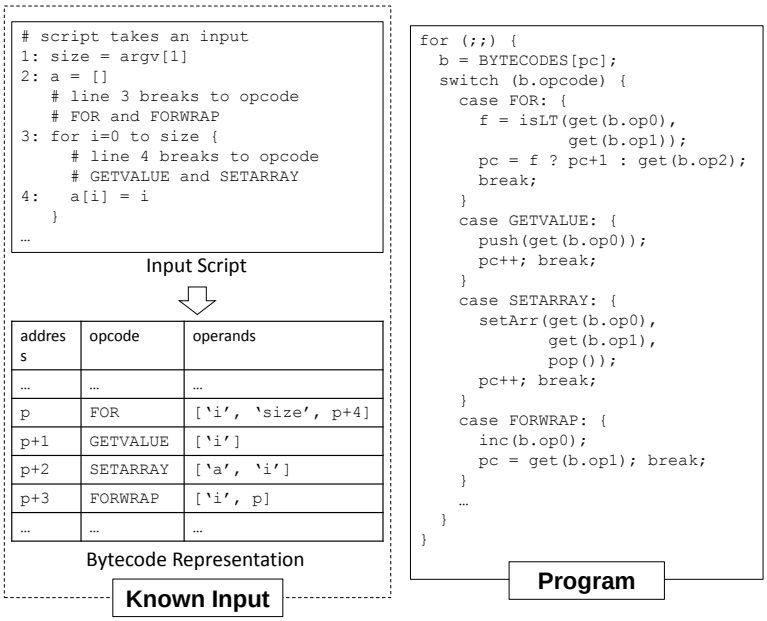
Chapter 3

Insight by Example: Script Interpreter

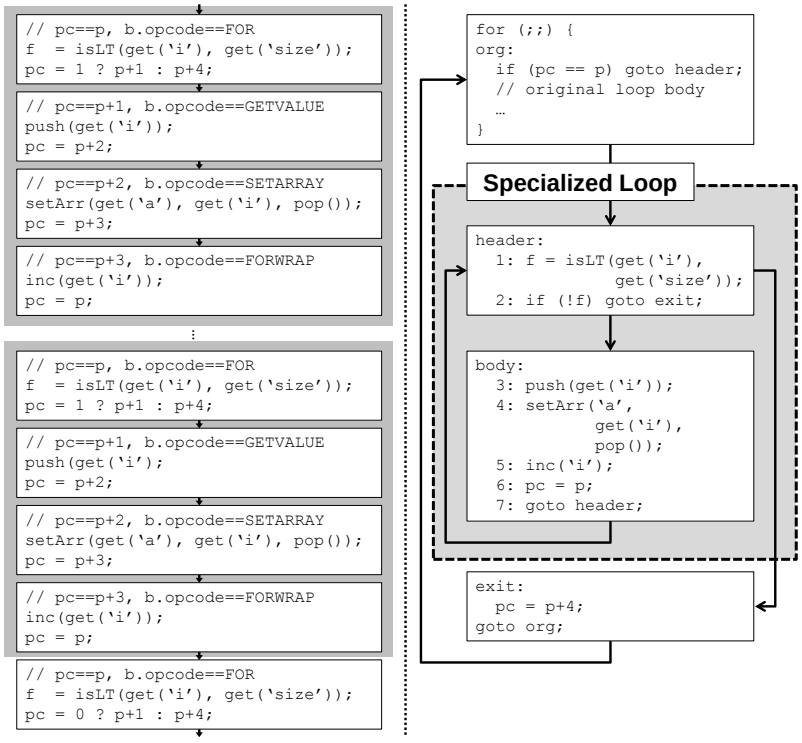
This chapter presents the insight behind the techniques proposed in this dissertation using the example of a script interpreter and an input script. Dynamic scripting languages are widely used because of their ease in programming. In particular, computational scientists heavily rely on dynamic scripting languages to minimize the time they spend writing programs. However, computational scientists also require high performance as they spend days or more waiting for their programs to complete [68]. This problem can be solved if the scripts that scientists write can be automatically parallelized to achieve parallel speedup. The following subsections show how this can be accomplished by coupling program specialization with speculative automatic parallelization.

3.1 Program Specialization as an Enabling Technique

Figure 3.1(a) describes the high level algorithm of the interpreter program and the 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`. As there are no loop carried dependences, the loop contains DOALL parallelism.



(a)



(b) (c)

Figure 3.1: 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.

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 3.1(b), an execution trace of the interpreter program with the input script, the same code blocks with the same values are repeated across iterations of the interpreter's main loop. A program specializer can capture these repeating patterns and generate a customized loop optimized for these patterns, as shown in Figure 3.1(c). Some bookkeeping instructions, like an instruction that increases `pc`, are optimized out during the specialization process.

As the repeating patterns are induced by the loop in the input script, the specialized loop reflects the structure of the input script loop, as well as the parallelism within the loop. Reproduction of parallelism within the input script loop into the specialized loop is enabled by the pattern-based specialization, but no prior specialization techniques before IPLS — which is proposed in this dissertation — performs pattern-based specialization. The following subsection will describe how the parallelism folded into the specialized loop can be harnessed by automatic speculative parallelization.

3.2 Parallelizing Specialized Loops

The variability in the potential dynamic behaviors of the interpreter in Figure 3.1(a) make it a poor choice for automatic parallelization. In contrast, as mentioned in the previous section, the specialized loop in Figure 3.1(c) reproduces the parallelism within the loop in the input script, and thus is more amenable to automatic parallelization. Figure 3.2(a) shows the PDG of the specialized loop. Dashed boxes in Figure 3.2(a) represent basic blocks.

Static analysis may not be able to prove that there is no data 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 3.2(b) is a DAG_{SCC} of Figure 3.2(a). The strongly-connected component formulated

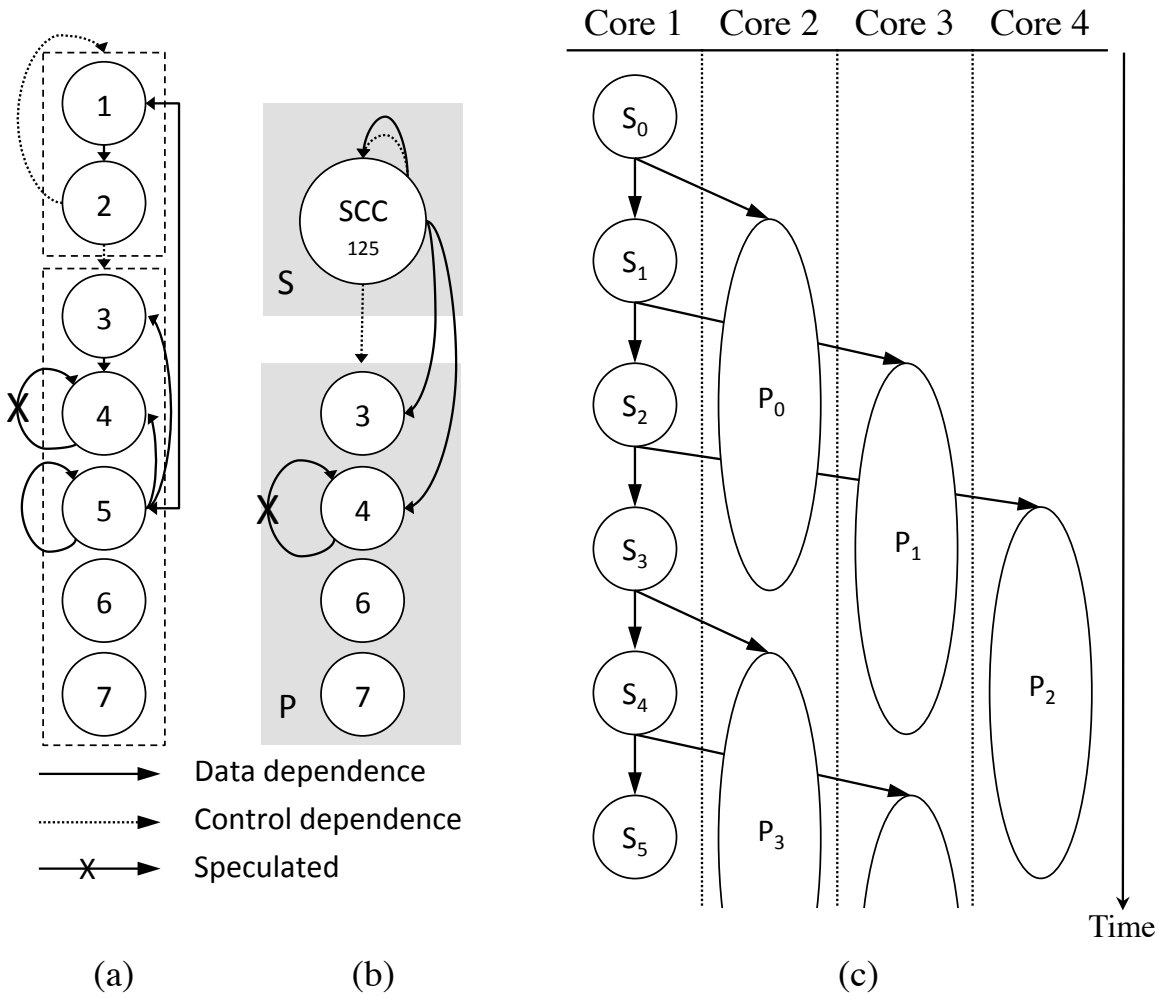


Figure 3.2: (a) PDG of the specialized loop from Figure 3.1(c). Note that the control dependence from node 2 applies to both sets of nodes 1-2 and 3-7. (b) DAG_{SCC} of PDG from (a). (c) Parallel execution plan using PS-DSWP.

around node 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, 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 things like loop induction variables are no longer applicable. However, instructions 3, 4, 6 and 7 do not contain loop-carried dependences, so each of these instructions can run in parallel. Applying PS-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 3.2(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. A later chapter in this dissertation (Chapter 5) discusses the enhancements over prior techniques to enable the automatic parallelization of complex programs such as specialized real-world script interpreters.

Chapter 4

Invariant-Induced Pattern-based Loop Specialization (IPLS)

This chapter presents Invariant-Induced Pattern-based Loop Specialization (IPLS) [60]. IPLS is the first program specialization technique that generates a program specialized for the predictable patterns of values induced by program invariants across loop iterations. IPLS is the first fully-automatic program specialization technique that is practical enough to be applicable to several complex and widely-deployed script interpreters. As a compile-time technique, IPLS can function as an enabling transformation for other compile-time optimizations.

IPLS is composed of three stages: profiling, pattern detection, and code generation. The IPLS profiler identifies static instructions in the program using dynamic information flow tracking and traces the values computed by static instructions. The IPLS pattern detector looks for repeating patterns across different iterations in the value trace, which are characteristics of the static instructions. The IPLS code generator specializes the program by unrolling the loop and specializing each unrolled iteration to the corresponding value from the detected pattern.

4.1 Overview of IPLS

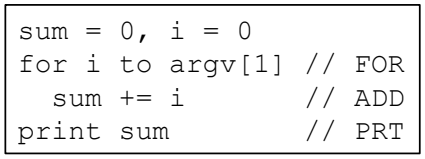
This section describes IPLS using a simple script language interpreter and an input script as an example. Figure 4.1(b) shows the control flow graph (CFG) representation of the main loop of the interpreter. Figure 4.1(a) is an input script to that interpreter. The input script is transformed into a sequence of opcodes: `FOR`, `ADD`, and `PRT`. The interpreter's main loop fetches an opcode, branches to the corresponding opcode handler, and repeats. It is beneficial to specialize the interpreter with respect to the input script if the input script is reused many times.

In Figure 4.1(b), basic blocks `A, B, C, D, ADD, MUL, FOR` and `PRT` compose the main loop while `PRE` is a preheader block. Individual instructions of the basic block `A` are expressed in a medium-level compiler intermediate representation. In the loop header block `A`, `OP` loads the next opcode from address `ADDR`. The next opcode is fetched into `OP` and the interpreter branches to the basic block corresponding to the opcode: if the value of `OP` is `FOR`, the interpreter jumps to the basic block `FOR`, and so on. The value of `ADDR` comes either from the loop preheader `PRE` when the loop is invoked, or from basic block `D` through the loop backedge.

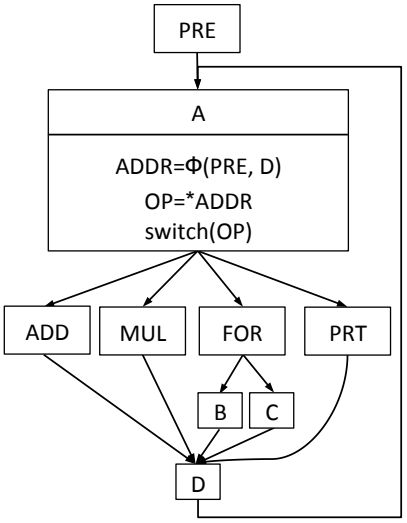
The workflow of IPLS consists of three stages as depicted in Figure 4.2. The rest of this section will provide a high-level overview of these three stages using the example interpreter and script. Figures 4.1(c) through (e) show the output of each stage of the specialization process.

4.1.1 Profiling Overview

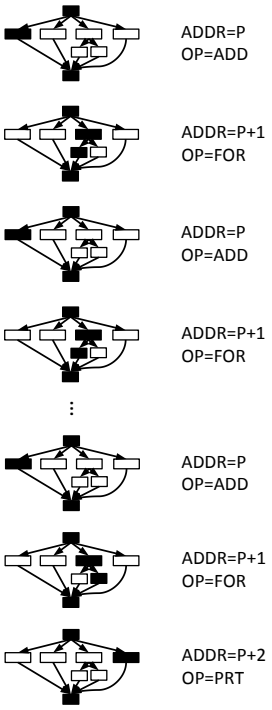
The first stage of IPLS is profiling. The goal of the profiling stage is to identify static instructions and the values they compute, thus enabling aggressive constant propagation and control flow optimization in the later stages. Towards this goal the profiled executable collects the following information:



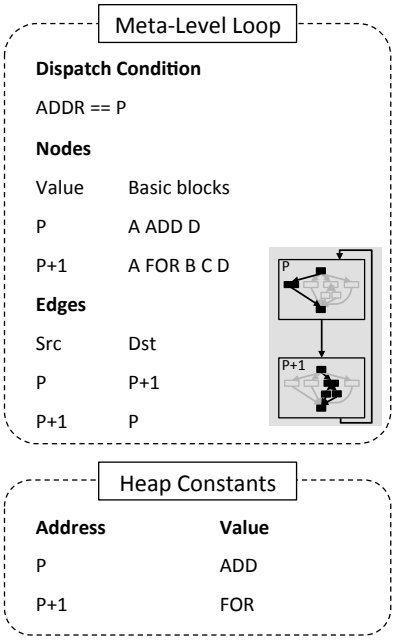
(a) Input Script



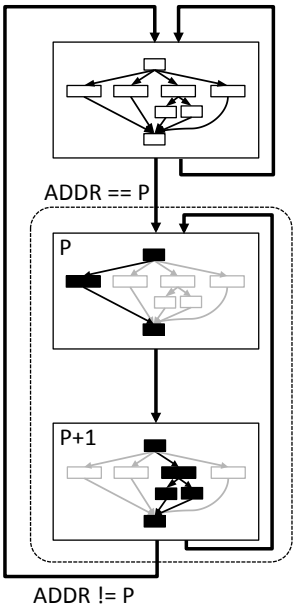
(b) Interpreter CFG



(c) Profile Result



(d) Pattern Detector Output



(e) Specialized Interpreter CFG

Figure 4.1: IPLS Specialization: (a) a fixed, static input script, (b) CFG of a script interpreter, (c) result of profiling, including a pattern of static values and their associated iteration control traces, (d) result of pattern detection, and (e) the loop produced by code generation

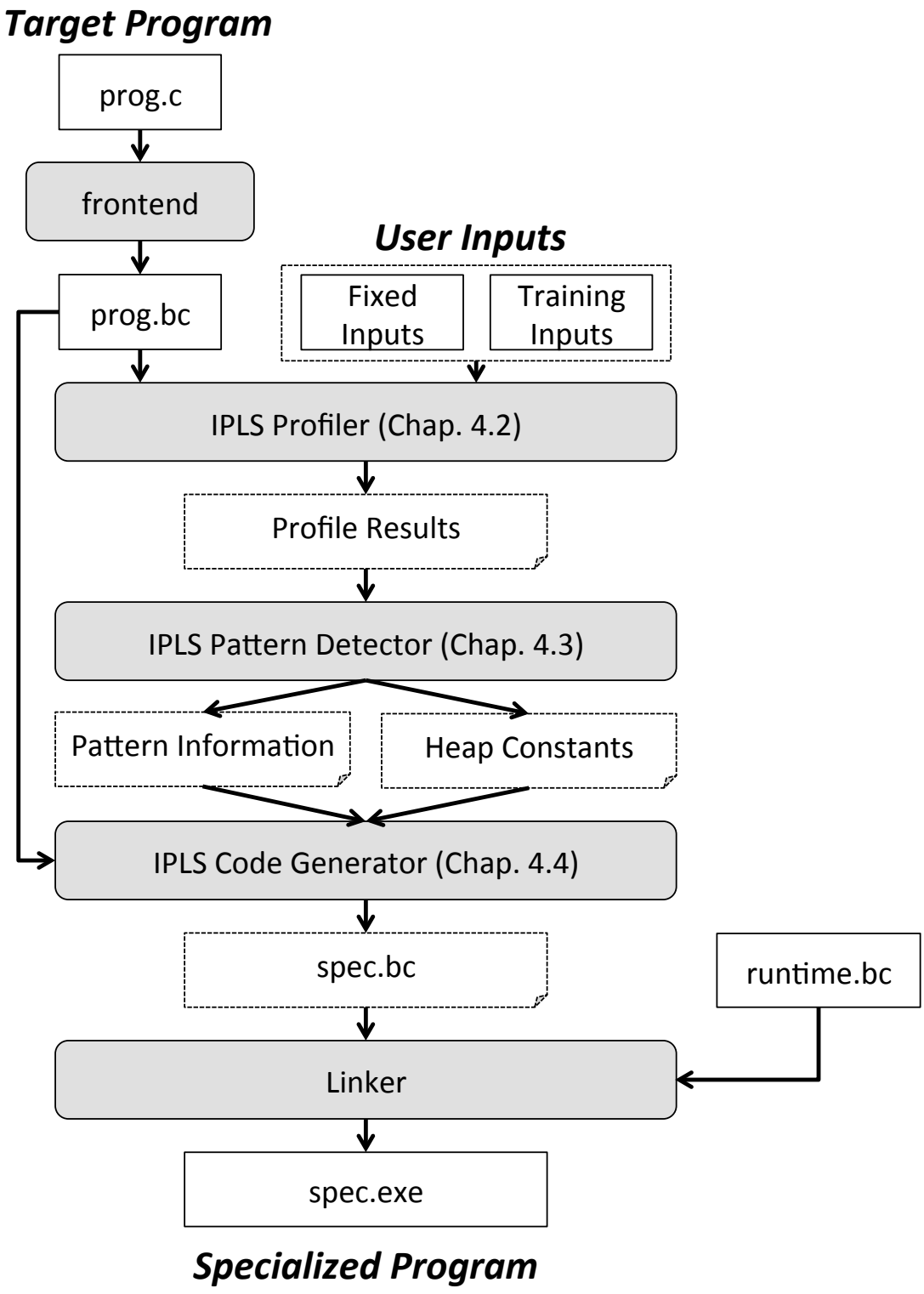


Figure 4.2: The high-level structure of IPLS. Note that `.bc` files are intermediate files containing LLVM bitcode.

- Values computed by the static instructions for each iteration of the loop;
- Address-value pairs for static load instructions (a load instruction is static if the result of the load is computed from a static instruction);
- A set of distinct control-flow paths through each iteration of the loop;
- The number of dynamic uses of values computed by each static instruction in an iteration; and
- Addresses of all basic blocks within the loop and all functions within the program.
- Size of all dynamically allocated memory objects.

To find static instructions within the program precisely without user annotations and/or heroic static analysis, IPLS employs Dynamic Information Flow Tracking (DIFT) [89]. The only information required from the user is which *program inputs* IPLS may assume fixed across different executions. For example, a user simply indicates that a script is a fixed input to the interpreter, but inputs to the script are not. This information is propagated along the data flow of the program by instrumented instructions, hence it is possible to decide whether each instruction depends only on fixed input. Implementation details will be discussed in Chapter 4.2.

Figure 4.1(c) depicts the results of profiling for the example program in Figure 4.1(b). Since `ADDR` points to an opcode derived from the fixed input script throughout execution, both `ADDR` and `OP` are classified as static. Therefore, the compiler instruments the binary for the profiling to trace the values of `ADDR` and `OP` for every iteration. The basic blocks executed in each iteration are also traced as represented by black boxes in Figure 4.1(c). The values of `ADDR` and `OP` constitute an address-value pair of the static load instruction for each iteration. For example, during the first iteration, the address-value pair is `<P, ADD>` since the value of `ADDR` equals to `P` and the value of `OP` to `ADD`.

4.1.2 Pattern Detection Overview

The pattern detection stage interprets profiling results to identify repeating patterns of values computed by static instructions. Such a repeating pattern suggests the existence of an input-driven loop. One can customize this input-driven loop via constant propagation and control flow optimization to generate a specialized loop that efficiently executes those iterations covered by the pattern. The specialized loop can be further optimized by other compiler optimizations, such as automatic parallelization.

Since there are multiple static instructions within the loop, the patterns they generate may suggest multiple ways to specialize the loop. In Figure 4.1(c), two static instructions generate two patterns (`ADDR = [P, P+1]` and `OP = [FOR, ADD]`). These two patterns are of the same length, and produce the same control flow pattern, though this is generally not the case. The specializer frequently must select among several specialization strategies using heuristics described in Chapter 4.3.

The pattern detector's choice of pattern for specialization determines the *dispatching instruction*, i.e. the static instruction whose value will control the path taken during each iteration. While executing the loop, if the dispatching instruction computes the value at the beginning of the pattern, specialized code blocks are dispatched. In the example, `ADDR =` ϕ`(PRE, D)` is the dispatching instruction and the pattern is `ADDR = [P, P+1]`. The dispatch condition compares `ADDR` to `P`, and if equal, dispatches into specialized code.

A detected pattern can be represented by a *Meta-Level Control-Flow Graph*. Each value in the pattern, which is produced by the dispatching instruction, corresponds to a node in the graph. The graph has an edge if the values corresponding to the nodes are generated from consecutive iterations of the loop. By representing the pattern as a graph, it is possible to capture patterns more complex than a repeating sequence of values.

In addition, the pattern detection stage also analyzes the address-value pairs of static load instructions taken in the profiling stage to output information about heap constants. IPLS assumes that, if the value v corresponding to the address a is identical across all

address-value pairs, and the pair of $\langle a, v \rangle$ appeared more than twice, address a holds a constant value v. The code generator uses this information about heap constants to create a specialized program. For example, the static load instruction `OP=*ADDR` in Figure 4.1(c) always loads `ADD` at address `P` and `FOR` at address `P+1`, so the two heap constants are passed to the code generator.

Figure 4.1(d) shows the output of the pattern detection stage. A loop in the meta-level CFG describes the pattern induced by the instruction `ADDR = Φ(PRE, D)`, with two nodes `P` and `P+1`, and two edges `(P, P+1)` and `(P+1, P)`. The code specialized with respect to this meta-level loop will be dispatched if the value of `ADDR` becomes `P`. Each node in the meta loop contains a set of basic blocks which are invoked while tracing the node. If different iterations generating the same value invoke different basic blocks, basic block information stored in the meta-level loop takes a union of them. For example, in Figure 4.1(c), two iterations generating the value `P+1` invoke basic blocks {`A, FOR, B, D`}, while the other invokes {`A, FOR, C, D`}. Therefore, a union of the two is reported as the basic blocks corresponding to the value `P+1` in the meta-level loop. The code generator uses this basic block information when creating code blocks specialized for the corresponding value.

4.1.3 Code Generation Overview

The code generation stage creates specialized codes for each meta-level loop identified in the pattern detection stage. It duplicates the original loop for each node in the meta-level loop and specializes the duplicated loop with respect to the value corresponding to the meta-level node. The code generator also inserts instructions to dispatch the specialized codes.

IPLS specializes a loop by first creating a special version for each iteration (meta-level node). When IPLS duplicates and specializes a loop iteration, it duplicates only those basic blocks listed in the meta-level node. This not only serves to minimize code growth caused

by specialization, but also simplifies the control flow of the specialized loop. These simplifications allow more scheduling freedom and thus enable more instruction level parallelism in a meta-level node.

The code generator then inserts branch instructions to link multiple specialized iterations into a specialized meta-level loop. In the example, it adds branches from the end of the specialized loop for `ADDR == P` to the head of the specialized loop for `ADDR == P+1` and vice versa, reflecting the two meta-level loop edges `(P, P+1)` and `(P+1, P)`.

These branches are *unconditional* if both of the following conditions are met. First, the next iteration must execute. Second, the value computed by the dispatching instruction of this meta-level loop is equal to the value that the next meta-level loop node is specialized for. For example, a branch from the specialized loop for node `P` to node `P+1` will be an unconditional jump, if it is guaranteed at the end of the specialized loop for node `P` that the loop executes at least one more iteration and that the value of `ADDR` will be `P+1` at the following iteration.

If these conditions cannot be guaranteed at specialization time, the branch must be *conditional*. For example, at the end of the specialized loop for node `P+1`, if the value of `ADDR` for the next instruction is only known at runtime, a conditional branch instruction whose predicate value is `ADDR == P` will be inserted to reflect the meta-level loop backedge from node `P+1` to node `P`.

The code generator also exploits heap constants to perform specialization across load instructions. For example, when specializing the duplicated loop with respect to the value `P` of `ADDR` in Figure 4.1, no further specialization is possible without the information that address `ADDR` holds a heap constant value `ADD`. Since information about heap constants is acquired by profiling, IPLS inserts instructions to verify the assumed constant value against actually loaded values into the specialized code. This information breaks dependences between the load instruction and its users, thereby exposing additional instruction-level parallelism.

Together with address information about basic blocks and functions provided by the profiler, information about heap constants is used to specialize indirect branches and indirect function calls. If a branch/function call target address is derived solely from a heap constant and the address matches the starting address of a basic block or function in the program, an indirect branch/function call can be replaced by a direct branch/function call to the target address. The replaced branch/function call is guarded by a comparison instruction to confirm the heap constant value. This transformation potentially reduces pipeline stalls by simplifying the program's control flow.

Figure 4.1(e) depicts the structure of the final optimized code after specialization guided by the meta-level CFG and heap constants in Figure 4.1(d). The main loop of the original interpreter (top white box) dispatches the specialized meta-level loop when the value of `ADDR` is equal to `P`. Specialized codes (round dotted box) are created by duplicating the original main loop twice for each meta-level loop node and specializing each of them using information about meta-level CFG and heap constants. Only necessary basic blocks (colored black) are duplicated during specialization. Under the assumption that the value of `ADDR` of the next iteration is guaranteed to be `P+1` at the end of the specialized loop for node `P`, the branch from node `P` to `P+1` is unconditional. However, the branch from node `P+1` to `P` is conditional because the next value of `ADDR` cannot be determined statically.

4.2 Profiling

The profiling stage of IPLS collects information to enable the compiler to specialize the program with respect to program invariants, including program inputs that are fixed across multiple program executions. As described in Section 4.1.1, the IPLS profiler performs variations of value profiling, load profiling, and path profiling.

Two optimizations distinguish the IPLS profiler from related techniques. First, the IPLS profiler restricts its observations to static instructions. This restriction limits profiling

results to safe specialization assumptions yet still provide information strong enough to support aggressive program specialization. If an instruction is static, then the sequence of values which that instruction generates during program execution is invariant across program executions. This property makes static instructions good candidates for program specialization.

Second, the IPLS value profiler only observes static instructions *in the loop header*. To support the pattern detector (Chapter 4.3), the IPLS profiler performs value profiling on candidate dispatch conditions. The dispatch condition must be computable during every iteration of the loop, and all static operations within the loop header satisfy this constraint. This restriction greatly reduces the amount of data collected, and in turn reduces the processing overhead of the pattern detector. As a corollary, later analysis of profiling results is insensitive to limited profile coverage, since it ignores operations which do not execute at least once per loop iteration. Note, however, that load profiling is still performed on all static loads within the loop, not only those from the header.

To collect information related only to static instructions without the help of heroic compiler analysis or user annotations, IPLS employs DIFT. DIFT tags each intermediate value in the program as either *static*, if the value is computed by static instructions, or *dynamic*, for all other instructions. Profiling instrumentation propagates these tags along the original data-flow of the program. To improve DIFT precision, the profiler tracks information flows along register and memory data dependences, but *optimistically* ignores information flows which potentially occur along control dependences. As a consequence, our implementation may report an instruction as static when a conservative implementation would report that value as dynamic (whether or not it actually is dynamic). This improved precision is desirable, since it allows IPLS to perform value profiling on instructions which are *likely* to be static, thus increasing applicability. The trade-off is the risk that specialization will attempt to predict a dynamic value, with no guarantee that the its sequences of values are invariant across program executions. Code generation guarantees the correctness of the program by

```
FILE* fp = fopen(argv[1], 'r');
metadata_fp = metadata_argv[1];
…
fread(buf, 1, size, fp);
metadata_fread = metadata_size & metadata_fp;
memset(metadata_buf, metadata_fread, size*1);
…
```

(a) Instrumented code that reads input

```
OP = *ADDR;
metadata_op = *(metadata_addr);
if (metadata_op == STATIC)
  profileLoadValue(…);
switch (OP) {
  …
}
```

(b) Instrumented code that uses input

Figure 4.3: Instrumentation added by the IPLS profiler to achieve dynamic information-flow tracking.

falling through to the general loop if misprediction occurs. Experimental results indicate that sequence prediction is robust and that sequence misprediction has negligible effect on the performance of specialized applications (Chapter 6).

Figure 4.3 shows the instrumentation used to track dynamic information flow in an example program. Figure 4.3(a) reads the input file specified by the first command line argument. Figure 4.3(b) uses the input. `ADDR` in Figure 4.3(b) points to each element of `buf` in Figure 4.3(a) through its execution, hence `OP` loads the value read from the input file. Bold and italicized lines are the instructions automatically instrumented by the compiler for tracing metadata and printing profile information.

The code snippet shows that the metadata of `argv[1]` is propagated to the metadata of `OP`, through the instrumented instructions. The profiler reports the information related to `OP` only if its metadata is set to *static*, which is true only when the metadata of `argv[1]` is static. Metadata of command line inputs, which indicates whether the input is fixed or not, is given by the user. Therefore, the profile information of `OP` appears at the output of the profiler only when the user declares that the first command-line input is fixed.

Since instrumentation is added at an intermediate representation level, instrumentation is not possible for functions whose source code is not available at specialization time. For this reason, tracing the flow of metadata across standard library calls are handled exceptionally via custom information flow tracking instructions. For example, Figure 4.3(a) shows custom tracking instructions for calls to `fopen` and `fread`.

Another issue the IPLS profiler is facing is the profiling of pointer values. Recall that

in the example in Figure 4.1 the dispatch condition of the specialized code was `ADDR == P`. However, the value `P` is a memory address, and there is no guarantee that the absolute pointer value `P` is consistent across across multiple executions of the specialized program. To address this problem, IPLS performs a variation of object-relative memory profiling [101] to derive consistent symbolic addresses for such pointers. A symbolic address is a tuple of (Object ID, offset). *Object ID* is an unique ID of each dynamic object. *offset* refers to the offset from the base of the object.

Figure 4.4 describes how the object-relative memory profiling is performed in IPLS. During execution the IPLS profiler traces every memory allocation instruction. For every invocation of a memory allocation instruction, a function call to trace the base address and size of the allocated object is executed. Traced information is maintained as a table shown in Figure 4.4(a).

When the profiler outputs the information related to a pointer, it prints the symbolic address tuple instead of the absolute number by referring to the table. For example, to trace the pointer `P`, which is in range of $[P_0, P_0 + S_n)$, the profiler finds the object pointed to by `P` using the table (An object with ID `n` for the example in the figure), calculates the offset, and records this information as shown in Figure 4.4(b).

To find the actual address of the symbolic address at specialized program execution time, IPLS compares heap constant values. Figure 4.4(c) shows the profiled heap constant values; the symbolic address (n, O_0) should have the 4-byte constant value `x`, while the symbolic address (n, O_1) should have the 4-byte value `y`. Profile results also report that the size of the object with ID `n` is S_n bytes (Figure 4.4(d)). With this information, the IPLS run-time function `findAddress`, which is described in Figure 4.5(a), discovers the base address of the object `n` during the specialized program execution. The preheader of the specialization target loop calls the function for all the objects (object `n` in the example) involved in the dispatch condition of specialized loops, as shown in Figure 4.5(b). The function iterates over all dynamically allocated objects with the same size as object `id`

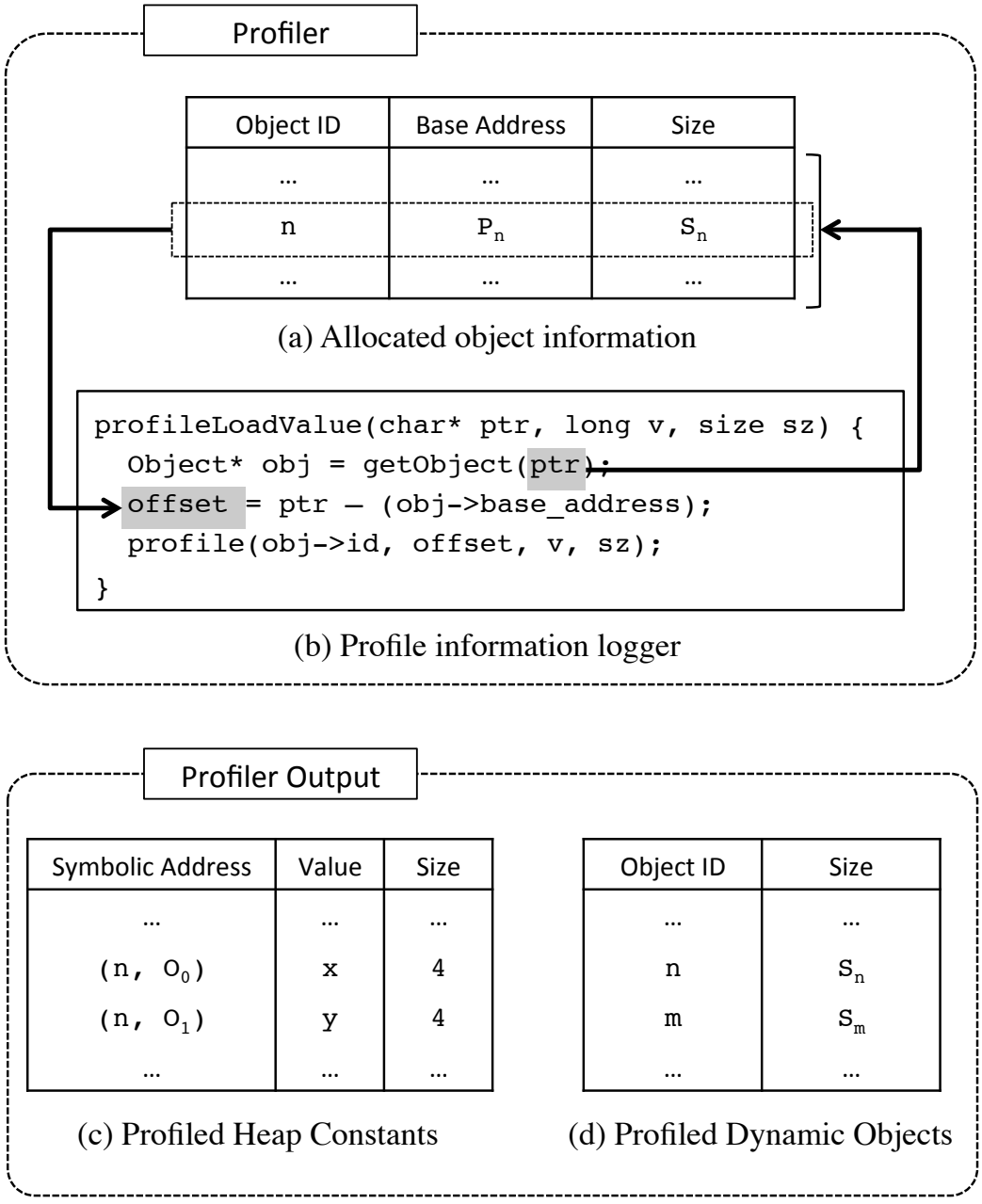
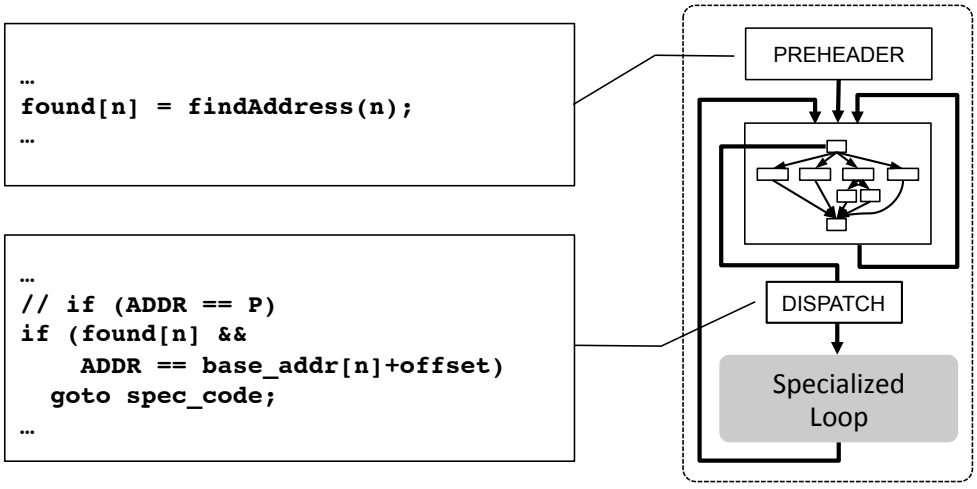


Figure 4.4: IPLS uses object-relative memory profiling to generate repeatable, symbolic names for relocatable address.

```
 1  bool findAddress(ObjectID id) {
 2    ProfiledObject* pobj = getProfiledObject(id);
 3
 4    // "objects" holds all dynamically allocated objects
 5    for (Object* obj in objects) {
 6      if (obj->size != sz) continue;
 7
 8      match = true;
 9
10      // hc is for each profiled heap constant value
11      for (HeapConstant* hc in pobj->heap_consts) {
12        offset = hc->offset;
13        size   = hc->size;
14        value  = hc->value;
15
16        if (obj->getValue(offset, size) != value) {
17          match = false;
18          break;
19        }
20      }
21
22      if (match) {
23        base_addr[id] = obj->base_addr;
24        return true;
25      }
26    }
27
28    return false;
29  }
```

(a)



(b)

Figure 4.5: (a) `findAddress` function to find the object corresponding to the profiled object at the specialized program execution time (b) Use of symbolic address on the specialized program side

(Figure 4.5(a), line 6) and checks if the values stored in the object match the profiled heap constant values. For object `n`, if any object has value `x` at offset O_0 and value `y` at offset O_1, the `findAddress` function concludes that the object is the incarnation of object `n` at specialized program execution time (Figure 4.5(a), line 11-20).

Figure 4.5(b) shows the use of symbolic addresses in the specialized program. Instead of using the absolute value of pointer `P` to check the dispatch condition of the specialized loop, it uses a value `base_addr[n] + offset` that is generated from the symbolic address: `base_addr[n]` is generated by `findAddress` function, and value `offset` comes directly from the output of the profile.

4.3 Pattern Detection

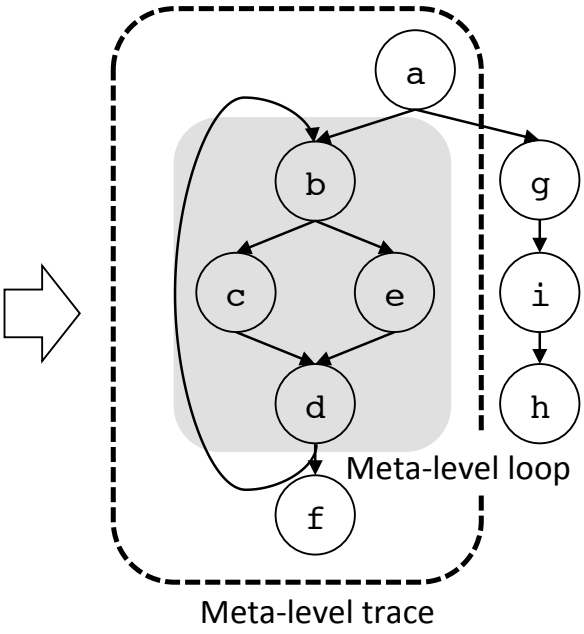
The IPLS pattern detector analyzes profiling information to identify specialization opportunities. The output of this stage includes meta-level loops, which represent repeating patterns within the traced values computed by static instructions, and possible heap constants extracted from the trace of static load instructions.

In addition to the meta-level loop, the pattern detector uses the *meta-level trace* to represent the repeating patterns in the program. While the meta-level loop represents patterns across multiple iterations of a single invocation of a loop, the meta-level trace represents patterns across multiple invocations of the loop. Figure 4.6(a) shows a summarized trace of values generated by a static instruction in a loop across multiple invocations. For the first and second invocations of the loop, the loop iterates for 8 times before it terminates, and for the third invocation it runs for only four iterations. Across the first and second invocation of the loop, the static instruction computes exactly the same sequence of values. Therefore, the specializer can exploit patterns that emerge across the *invocations* along with the patterns detected across the *iterations* of a single invocation.

In order to find a meta-level loop or meta-level trace from a given trace of values, the

	Invocation		
	0	1	2
Iter 0	a	a	a
Iter 1	b	b	g
Iter 2	c	c	i
Iter 3	d	d	h
Iter 4	b	b	
Iter 5	e	e	
Iter 6	d	d	
Iter 7	f	f	

(a) Example trace



(b) Meta-level loop/trace detected from example trace (a)

Figure 4.6: Meta-level loops/traces detection extracts a graph which resembles a control-flow graph in which loops are identified.

pattern detector first transforms the sequence of trace values into a graph: each traced value becomes a node of the graph, and edges are added between two consecutive values in the trace. Figure 4.6(b) depicts a graph generated from the trace in Figure 4.6(a).

To detect a meta-level loop, IPLS runs a natural loop detection algorithm on the graph built from the trace. In the example of Figure 4.6(b), an edge from node `d` to node `b` forms a backedge because its destination node dominates its source node, thus a meta-level loop including nodes `b, c, e`, and `d` can be detected. The dispatch condition for the specialized loop for the meta-level loop is met if the computed value of the static instruction whose profiled values induce the meta-level loop matches the value of the loop header.

Meta-level traces are created by merging all iteration traces that share the same value as the first iteration. The dispatch condition for the specialized codes for a meta-level trace is determined by the common value from the first iterations. Since the traces of the three invocations shown in Figure 4.6(a) share the same value `a` generated from the first iteration, Figure 4.6(b) itself can be a meta-level trace. However, if the value diverges after the first iteration for different invocations, and some of the values appear with a very low probability across invocations, specialized loops for those values will merely increase the program size without much benefit. To prevent this case, only the trace of invocations which share an identical iteration trace are included in the meta-level trace. In Figure 4.6(a), traces of the first and second invocations are identical, hence are combined in the meta-level trace, yet the trace of the third invocation is excluded because there is no common trace.

At this point the pattern detector may have found several patterns in the trace data. However, the loop can only be specialized according to one pattern. To find the right pattern for specialization, IPLS uses a heuristic based on two measures: (1) coverage of the pattern and (2) precision of the pattern. IPLS chooses the pattern for which the product of these measures is greatest.

The *coverage* of the pattern is calculated by taking a ratio of the number of iterations covered by the pattern to the total number of iterations in the trace. This measure is related

to the benefit of specialization since a higher coverage means that the specialized code likely accelerates a greater portion of the total iterations. In the example of Figure 4.1(c), the pattern of `ADDR = [P, P+1]` and the pattern of `OP = [FOR, ADD]` have the same coverage of $6/7$. If the coverage of the pattern is less than a certain threshold, IPLS discards the pattern (In the current implementation, the threshold value is set to 0.01).

The *precision* of the pattern represents a degree of possible control-flow simplification when the program is specialized against the pattern. The precision of the pattern is computed by taking the average precision of all values within the pattern. The precision of each value is calculated by averaging the precision of all the iterations that compute the value. To get the precision of iteration `I`, IPLS pattern detector divides the number of basic blocks executed for iteration `I` by the size of the set of basic blocks that have been invoked across all the profiled iterations that compute same value as `I`. In the example of Figure 4.1(c), the precision of value `OP = FOR` is calculated by averaging the precision of iteration 2, 4, and 6, as those are the three iterations that compute the value `OP = FOR`. While iteration 2 and 4 executed four basic blocks, {`A, FOR, B, D`}, iteration 6 executed four basic blocks, {`A, FOR, C, D`}. Therefore, the set of basic blocks that have been executed across all iterations computing `OP = FOR` is {`A, FOR, B, C, D`}. Thus iterations 2, 4, and 6 all have the same precision value of 0.8, which is also the average. By applying the same algorithm, the precision of `OP = ADD` is 1.0. The precision of the pattern of `OP = [FOR, ADD]` is the average of these values, 0.9. The pattern of `ADDR = [P, P+1]` has a precision of 0.9 as well.

If the product of coverage and precision is same for multiple patterns, as in the case of the pattern `OP = [P, P+1]` and `OP = [FOR, ADD]`, the number of dynamic uses of the value computed by the static instruction generating the pattern is employed as a tie-breaker. The profiler measures the invocation count of the instructions use the value computed by the static instruction *s*. This number is correlated to the benefit of specialization since a higher number implies more computation to be optimized away via precomputation.

In the example of Figure 4.1(c), the instruction computing `ADDR` has two use instructions (i.e., `OP=*ADDR` and `switch(OP)`), while the instruction `OP=*ADDR` has only one use instruction (`switch(OP)`). Therefore, IPLS chooses to optimize the loop with respect to the pattern of `ADDR = [P, P+1]`.

4.4 Code Generation

Code generation uses meta-level loop/trace information and possible heap constant information passed by the pattern detector to specialize codes. Specialized codes are expected to have less computation than the corresponding original codes and be better structured to exploit instruction-level parallelism.

Figure 4.7 describes, step-by-step, how the program represented as a control flow graph in Figure 4.7(a), which is taken from Figure 4.1(b), is specialized using the information in Figure 4.1(d). Throughout this section the code generation process will be explained by walking through the figure.

Step 1: Splitting header and latch block First, IPLS splits the original loop header block and loop latch blocks (blocks which are the source of a loop backedge). As of now, IPLS targets only natural loops for specialization. Natural loops can be canonicalized to have only one backedge and only one latch block. Dispatch instructions for specialized codes are added to the new header block, and the new latch block becomes a point where control-flow is merged after the execution of specialized codes. Figure 4.7(b) shows a new control flow graph after splitting the header and latch blocks.

As shown in the figure, ϕ-nodes placed in the original loop header are moved to the new header after splitting. If the dispatch instructions of the specialized codes depend on ϕ-nodes, no other modifications are required. However, if dispatch instructions depend on other instructions, the instruction and the instructions upon which it depends must be cloned into the new header block.

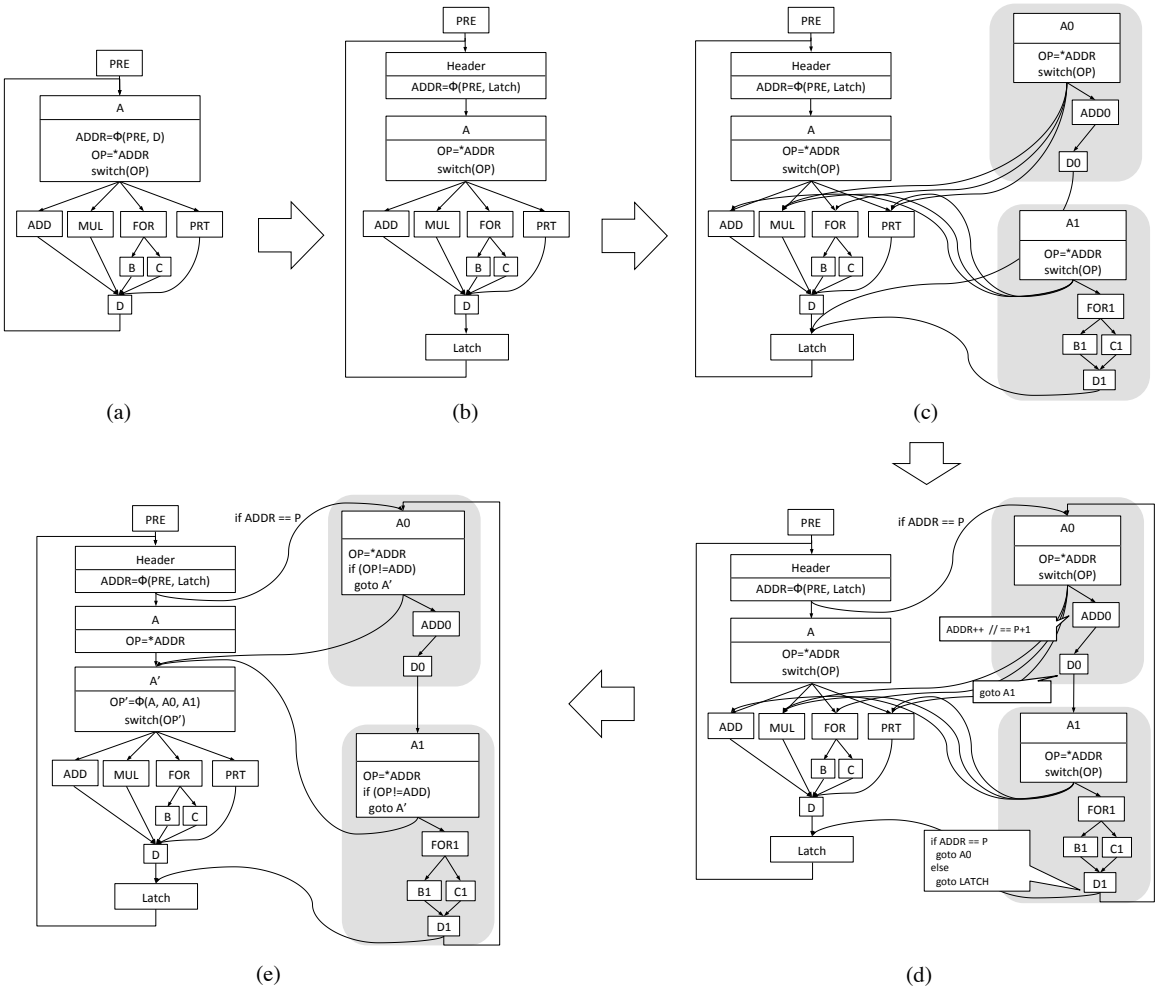


Figure 4.7: The code generation process: (a) original loop, (b) splitting the loop header and latch, (c) cloning and specializing iterations from a pattern, (d) adding dispatch conditions and stitching specialized iterations into a loop, and (e) adding unexpected exit conditions.

Step 2: Cloning basic blocks To create versions of the loop iteration corresponding to observed traces, the compiler clones basic blocks in the nodes in a meta-level loop/trace. As described in Chapter 4.1.3, the basic blocks to be cloned for each meta-level node are provided by the pattern detector, which is based on profiling. For a branch instruction in a cloned block, if the original branch target block is also cloned then the instruction is modified to branch to the corresponding cloned block. If not, the branch jumps to the target block in the original loop. Figure 4.7(c) depicts the control flow graph after cloning of basic blocks and adjusting branches.

Instead of cloning basic blocks invoked in each node during profiling, one alternative can be analyzing executable basic blocks for each meta-level node and cloning all executable blocks, to minimize the number of returns back to the unspecialized loop. This happens when a branch jumps to a basic block that is never executed during profiling. This approach makes sense because cloned blocks for each meta-level node are specialized for a specific value of the instruction inducing meta-level loop/trace, and specialization may mark a large portion of basic blocks within the loop as unreachable. In practice, the analysis marks every basic block to be executable for non-negligible cases, especially if the specialized loop contains an inner loop. Since cloning the entire loop may lead to an explosion in code size, IPLS clones only the profiled basic blocks.

Step 3: Adding dispatch instruction and meta-level edges After cloning basic blocks for each meta-level node, dispatch instructions are inserted at the loop header created in Step 1 to invoke the specialized loops. The pattern detector informs the code generator of a dispatch condition for each meta-level loop and trace. Figure 4.7(d) depicts the control flow edge inserted to dispatch the specialized loop, which is taken when the value of `ADDR` becomes `P` (Recall the value acquired by the symbolic representation of value `P` is used in dispatch instructions, as described in Chapter 4.2, but we use the absolute value `P` here to simplify explanation).

Branches for meta-level edges are also added in this step. The branches are conditioned on the dynamic value of the dispatch condition. The branch at the end of block `D1` in Figure 4.7(d) shows this. The branch checks whether the value of `ADDR` will actually be `P` on the next iteration. If it matches, it jumps to the specialized loop corresponding to the value of `ADDR` being `P`, or returns to the original loop otherwise. If a meta-level node has multiple outgoing edges, IPLS uses a switch statement instead of a branch.

Sometimes a simplified control flow between cloned basic blocks makes branches for meta-level edges unconditional. The branch added at the end of block `D0` in Figure 4.7(d) is the example. `D0` is only reachable via the path `A0`→`ADD0`→`D0`. By basic data-flow rules, we know `ADDR == P+1` along this path. Therefore, without requiring checking, the loop specialized for the condition of `ADDR == P+1` is invoked after the execution of `D0`. Such simplification of the control flow opens opportunities to exploit instruction-level parallelism across different iterations.

Step 4: Exploiting possible heap constants As the last step of code generation, IPLS exploits possible heap constant information.

Figures 4.7(d) and (e) differ in block `A0`, where the switch instruction has been replaced with a conditional branch. This replacement is possible because the specializer knows (i) `ADDR` must point to `P` and (ii) the pattern detector reports that the memory at `P` holds a heap constant value `ADD`. However, the switch cannot be simply replaced by an unconditional branch. The heap constant information is derived via profiling, which must be verified at runtime; instructions to check the validity of possible heap constant information must remain.

Though it seems that there is no benefit by using heap constant information in block `A0`, performance is improved by replacing switch instructions (often lowered to a jump table in assembly) with conditional branches, which improve the performance of branch prediction. Particularly, the branch prediction is nearly perfect for such cases since the profiled

heap constant information is generally accurate. Although it is not clear in this example, heap constant information also breaks dependences between a load instruction and its uses, because the loaded value can be safely assumed to be the expected heap constant after the checking instruction. Breaking dependences creates more optimization opportunities, including better chances for instruction level parallelism.

As an alternative to pre-checking heap constants and branching, speculation can be employed to assume all possible heap constants. When using speculation, a log of the comparison result is maintained rather than branching on the comparison result. Then the program occasionally checks the logged value at runtime to see if a *misspeculation* has occurred, and if so, the program *rolls back* to the previous checkpoint of the program, where the program maintains correct state. By removing conditional branches speculative execution opens additional opportunities to exploit instruction level parallelism. However, the overhead of logging and periodic checking of the comparison results may negate the benefit resulting from more instruction level parallelism. For this reason, checking and branching is used in this thesis instead of speculative techniques.

Chapter 5

Parallelizing Specialized Programs

This chapter describes enhancements over existing automatic parallelization techniques to handle complex programs, such as IPLS specialized programs. Although the motivation for inventing these techniques is to exploit input parallelism folded into IPLS specialized programs, the techniques are generally applicable to other programs as well.

Chapter 5.1 describes the overall workflow of the system that exploits input parallelism in a fully automatic fashion. The system applies automatic speculative parallelization to the IPLS specialized program. Chapter 5.2 presents context-sensitive speculation techniques. Supporting context-sensitivity improves the precision of speculative analysis, and thus improves the applicability of the automatic parallelization. In particular, context-sensitive speculation plays a critical role in automatic parallelization of IPLS specialized loops, which have been aggressively unrolled. Chapter 5.3 proposes optimizations to the run-time system supporting speculative parallelization. Although speculation is necessary to parallelize complex programs, the high performance overhead of the run-time to support speculation nullifies the benefit of parallelization. Optimizations described in Chapter 5.3 reduce the overhead of memory dependence speculation validation and enables parallel speedup for even aggressively speculated programs.

5.1 Overall Workflow

Figure 5.1 depicts the workflow of the system to automatically exploit input parallelism. The user provides C or C++ source code, a fixed input, and a representative training input. IPLS specializes the code against the fixed inputs, as described in the previous chapter. Next, the system speculatively parallelizes the code, which entails an enabling transformation (Chapter 5.1.1), planning (Chapters 5.1.2–5.1.3), and transformation (Chapters 5.1.4–5.1.5).

5.1.1 Enabling Transformation: Loop Peeling

As explained in Chapter 4, IPLS emits the specialized program in response to invariant-induced execution patterns. However, even when the loop-level parallelism is available in specialized loops, the specialized loops may not be perfectly amenable to automatic parallelization. When specializing the Perl interpreter, for example, the IPLS code generator creates loops that store many values during the first iteration and then reference those values during all subsequent iterations. This memory usage pattern induces loop-carried memory dependences from the loop's first iteration to all subsequent iterations. Loop-carried memory dependences inhibit many thread extraction techniques [75].

Fortunately, peeling the first iteration of specialized loops converts loop-carried dependences sourced from the first iteration of the loop into dependences from live-in values. The system uses a memory dependence profiler to identify cases where loop peeling eliminates such loop-carried dependences. Loop peeling also helps support context-sensitive speculation, as discussed in Chapter 5.2.

5.1.2 Profiling

The system further enables parallelization using high-confidence, profile-guided speculation of biased conditional branches, predictable values, and memory dependences. Profil-

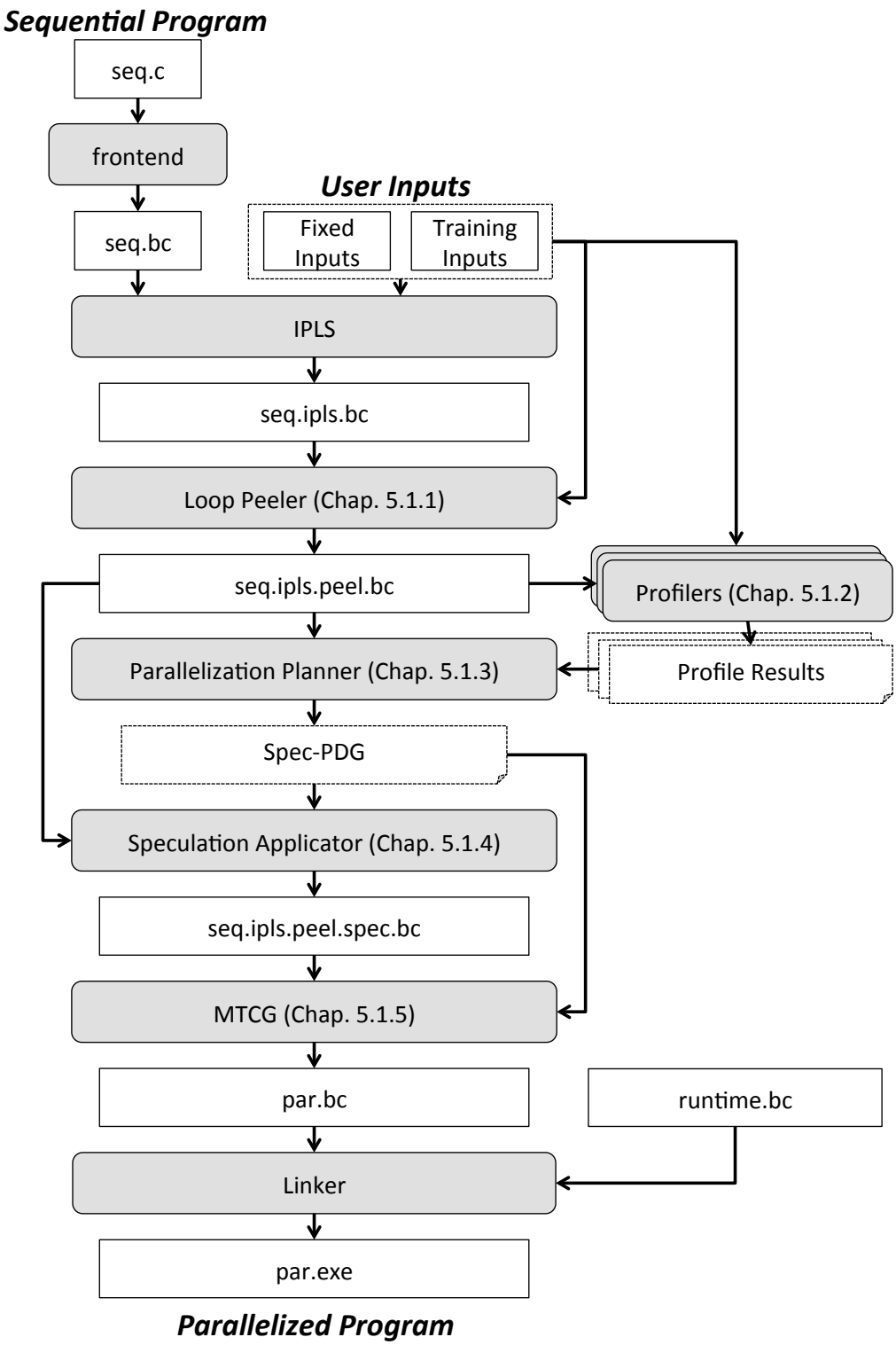


Figure 5.1: The workflow of the system to automatically exploit input parallelism. A sequential program, fixed inputs, and training inputs are inputs to the system. After undergoing IPLS specialization and an enabling transformation (5.1.1), the system profiles (5.1.2) the program and then performs speculative parallelization (5.1.3-5.1.5). Note that `.bc` files are intermediate files containing LLVM bitcode.

ers run on the specialized program with a representative input to estimate the program's expected-case behavior. Whereas a Program Dependence Graph (PDG) [24] 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 dependencies have minimal effect on the applicability of parallel transformations [37]. This phase does not modify the IR.

A **control-flow edge profiler** (LLVM's `-insert-edgeprofiling` [43]) identifies heavily biased branches. By speculating that heavily biased branches are essentially unconditional branches, unlikely instructions become dead code and some control dependences are removed from the speculative PDG. A **loop-invariant value profiler** determines whether ϕ-nodes compute the same loop-invariant values and whether load instructions always read the same loop-invariant values during every iteration. Similarly, a **linear value prediction profiler** discovers load instructions whose value can be predicted as a linear function of the loop's canonical induction variable. These three profilers drive value prediction speculation to eliminate loop-carried data dependences from the speculative PDG. Finally, a **memory dependence profiler** [50] records the memory dependences observed during a training run. The parallelization system speculates that memory dependences not observed during profiling do not exist and removes them from the speculative PDG.

Profiling results are *context-sensitive* for memory dependence, loop-invariant load, and linear value prediction profiles. Chapter 5.2 describes how the parallelization system leverages context-sensitivity.

5.1.3 Parallelization Planner

The parallelization planner formulates a concrete plan to achieve parallel execution based upon the speculative PDG (from Chapter 5.1.2). The planner considers the applicability of the PS-DSWP parallelization technique [75] on the speculative PDG of each hot loop. PS-DSWP supports the classical DOALL transformation as a pipeline with a single parallel

stage. The planner rejects any parallelization that it does not expect to yield a performance improvement. Expected performance improvement 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. The planner also rejects nested parallelism: if the system can parallelize two loops, where one occurs within the other, it chooses to parallelize the one that will yield the higher expected performance improvement.

The parallelization planner finds a set of parallelizable loops with non-overlapping invocations while maximizing the expected performance improvement across the whole program. The planner builds an undirected graph in which nodes represent loops in the program. The weight of the node indicates the expected performance improvement of the loop. Edges are added between nodes if the loops represented by the nodes are not nested within each other. An extended Bron-Kerbosch algorithm [33], an algorithm to find exact solutions to the maximum weighted clique problem, is applied to the graph to find the set of loops that maximizes expected performance improvement across the whole program.

5.1.4 Speculation Applicator

Once the parallelization planner has selected a parallelization plan (Chapter 5.1.3), the system proceeds to the transformation phase. First, it transforms the sequential IR into speculative sequential IR. The speculation applicator inserts new instructions 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.

For control-speculation, the speculation applicator adds instructions in speculatively dead basic blocks that trigger misspeculation. For value-prediction speculation, the speculation applicator inserts instructions to compute a predicted value, replaces uses of the original value with the prediction, and inserts instructions that validate that the predic-

tion is correct. To validate memory dependence speculation and to provide misspeculation recovery, Software Multi-Threaded Transactions (SMTX) [74] are used. The validation applicator inserts instructions that commit transactions after each loop iteration and inserts `mtx_write` and `mtx_read` calls on speculated store and load instructions.

5.1.5 Multi-Threaded Code Generator

After applying speculation, the system parallelizes the speculative sequential IR. The Multi-Threaded Code Generation (MTCG) algorithm transforms the speculative, sequential IR to create one or more threads [61]. For pipelined parallelization, like DSWP or PS-DSWP, instructions assigned to each stage of the pipeline partition 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.

Our implementation also employs the *replicable-stage* extension [32], which serves as a generalization of induction variable expansion. The code generator emits instructions into each worker that rematerialize side-effect free, loop-carried values such as induction variables or pointer-chasing recurrences. These instructions are redundant across worker processes, but reduce the core-to-core communication of values that are more easily recomputed. Although each of n parallel stages primarily executes $1/n$ iterations, the parallel stage also executes the rematerialized instructions for the remaining $(n-1)/n$ iterations.

5.2 Context-Sensitive Speculation

This section describes context-sensitive speculation. Chapter 5.2.1 motivates context-sensitive speculation with an example of specialized interpreter program. Chapter 5.2.2 explains how the profilers used in the automatic parallelization system generate context-sensitive profiling results. Chapter 5.2.3 proposes a run-time system to support context-sensitive

speculation.

5.2.1 Motivating Context-Sensitive Speculation

Figure 5.2 depicts the importance of context-sensitive speculation. Figure 5.2(a) is a code snippet of an interpreter program specialized for a script that includes a simple loop, `L`, whose body is `arr[i]+=1`, and 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` (Figure 5.2(b)) is called twice in the loop (lines `(4)` and `(5)`). Once to increment of the induction variable `i`, which has a loop carried data dependence across iterations of `L`, and the other to increment the value of `arr[i]`, which has no loop carried data dependence across iterations of `L`. Note that these were the same callsite in the non-specialized version of the interpreter.

To achieve more precise dependence information between instructions that is beyond the capability of static analysis, memory dependence profiling is necessary. A profiler observes that there are loop-carried dependences 9→7 and 10→10, which are formulated only around the object `o_i` that models the induction variable `i`. However, if profiling results ignore the calling context and only report that there are loop-carried dependences 9→7 and 10→10, the compiler will conservatively assume that those dependences exist across *every* callsite of `add1`. Figure 5.2(c) is the speculative PDG of `L` for such a case. [1] Due to the lack of context information for the `add1` function, dependence edges are added from 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 5.2(d)) shows that only node 1, 4 and 6 form an SCC, thus parallel transformation can be applied. [2]

[1] The run-time system supporting memory versioning removes all false dependences [74] and thus they are not displayed.

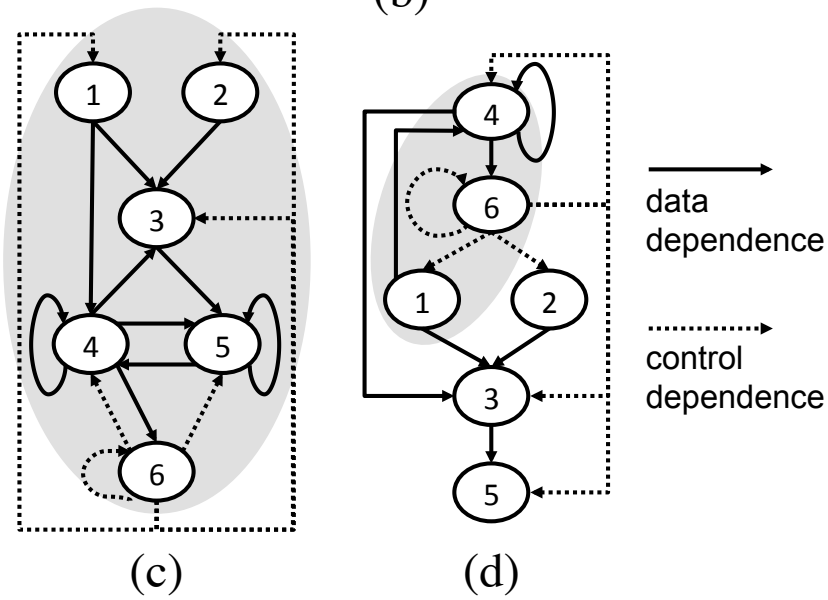
[2] Note that `flag` will always be `INT` for line `(7)` when `add1` is called from line `(5)`. The context-

```
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)



(c) (d)

Figure 5.2: Benefits of context-sensitivity. (a) Example code from a specialized interpreter. (b) Source code for the `add1` function. (c) PDG constructed using profiling results with no context-sensitivity, which results in no opportunity for parallelism. The grey region represents an SCC. (d) PDG using profiling results with context-sensitivity. Note that the self edge on node 5 and the mutual edge between nodes 4 and 5 have been eliminated, which reduces the size of the SCC and enables parallelism.

5.2.2 Context-Sensitive Profiling

As described in Chapter 5.1.2, loop-invariant load, linear value prediction, and memory dependence profilers used in the automatic parallelization system proposed in this dissertation generate context-sensitive profiling results. To support context-sensitivity, these profilers trace execution contexts along with the core information.

The execution context specifies a position within the profiling target loop. For a certain instruction I in the target loop L, if I is syntactically contained in L, then I itself is the execution context. Alternatively, if instruction I is buried within a procedure call, then the execution context references the call site in loop L. In the example of Figure 5.2(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 profilers do not trace the execution context for nested invocations of the target loop.

Loop-invariant load profiler A loop-invariant load profiler enables speculative optimization by removing dependences incident on loop-invariant load instructions. A loop-invariant value profiler discovers a load instruction LD in execution context C that meet the loop invariant load criterion:

Loop-Invariant Load Criterion: Let LD be the load instruction in loop L. LD is speculatively loop-invariant *in context C iff throughout the profiling run, every execution of LD in context C reads the same value V from the same memory address A.*

Note that $C == LD$ if LD is syntactically contained in the target loop, by the definition of execution context. Algorithm 1 profiles loop-invariant load values. Loop invariants are

sensitive loop-invariant value profiler can capture this fact and thereby speculatively remove the dependence edge from 9→7. However, this will not affect the PDG of loop `L` because the dependence edge 4→4 will still exist due to the dependence 10→10.

Algorithm 1: $LoopInvariantLoadProfile(C, LD, A, V)$

```
1 let Key ≔ ⟨C, LD⟩;
2 if Key in LoopInvs then
3     if LoopInvs[Key].A != A then
4         let LoopInvs[Key].valid ≔ false;
5     if LoopInvs[Key].V != V then
6         let LoopInvs[Key].valid ≔ false;
7 else
8     let LoopInvs[Key] ≔ LoopInvariant(A, V);
```

traced for each pair of execution context C and load instruction LD. If the loaded value and the memory address are identical across all executions of $\langle C, LD\rangle$, the profiler assumes that LD under execution context C loads a loop-invariant value.

Linear value prediction profiler A linear value prediction profiler finds load instructions that meet the following criterion:

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

Algorithms 2 and 3 describe how the linear value prediction profiler determines coefficients m, b to characterize each speculatively linear load. Algorithm 2 runs for each dynamic load instruction in the target loop; V and A refer to the loaded value and the memory address of the load instruction, respectively. A separate predictor is assigned to each pair of $\langle C, LD\rangle$. A predictor finds m and b by computing the linear interpolant from the first two sample points (Algorithm 3, lines 14-18), then checks the consistency of the linear interpolant with additional samples (Algorithm 3, line 20-21). Parameter x of func-

Algorithm 2: $LinearValuePredictionProfile(C, LD, ITER, A, V)$

```
1  let Key ≔ ⟨C, LD⟩;
2  if Key not in Predictors then
3      let Predictors[Key] ≔ LinearValuePredictor(A);
4  AddSample(Predictors[Key], ITER, V, A);
```

Algorithm 3: $AddSample(lp, x, y, addr)$

```
1  if lp.addr != addr then
2      let lp.valid ≔ false;
3      return;
4  if not lp.init then
5      let lp.x ≔ x;
6      let lp.y ≔ y;
7      let lp.init ≔ true;
8  else if not lp.interpolated then
9      if lp.x = x and lp.y != y then
10         let lp.valid ≔ false;
11         return;
12     if lp.x = x and lp.y = y then
13         return;
14     let x_interval = x − lp.x;
15     let y_interval = y − lp.y;
16     let lp.m ≔ y_interval / x_interval;
17     let lp.b ≔ y − lp.m × x;
18     let lp.interpolated ≔ true;
19 else
20     if (lp.m × x + lp.b) != y then
21         lp.valid ≔ false;
```

tion `AddSample` represents the target loop iteration count, while parameter `y` represents a loaded value. If the target loop contains nested loops, a static load instruction can be executed multiple times within a single iteration of the target loop. Lines 9-13 in Algorithm 3 checks if a load instruction always loads the same value within a single target loop iteration.

Memory dependence profiler A memory dependence profiler employs a *shadow-memory* based profiler to trace memory accesses within the profiler. Each byte of the profiled program's memory corresponds to 8 bytes of metadata in the shadow memory. Allocation and deallocation of the shadow memory is performed by instrumentation to the allocation/deallocation events of the original memory. For each dynamic instruction that writes to memory, the profiler records the execution context and the iteration count of the target loop as metadata, as described in Algorithm 4; C_W, $ITER_W$, and A refer to the execution context, iteration count of the target loop, and the memory address to write to, respectively.

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 5.2, it is not important to know that instruction `(9)` and `(10)` write a value but it is important that the execution context was `(4)`. Address translation between the given address and the corresponding shadow address is performed by a small number of bit-wise operations.

Memory dependences are logged at each dynamic execution of an instruction that reads from memory. If instruction LD reads a value from the memory address A at the target

Algorithm 4: $ShadowMemoryWrite(C_W, ITER_W, A)$

1 **let** $Metadata \coloneqq pack(C_W, ITER_W)$;
2 **let** $SA \coloneqq Translate(A)$;
3 **let** $*SA \coloneqq Metadata$;

loop iteration count of $ITER_R$, the profiler reads the metadata from the shadow memory corresponding to A and computes the dependence information as described in Algorithm 5. Dependences are characterized by the execution context of the write event (C_W), the execution context of the read event (C_R), the instruction that reads the memory (LD), and whether the dependence is either loop-carried (i.e. $ITER_W$!= $ITER_R$) or not.

Unlike memory write instructions, memory read instructions are traced along with execution contexts. The motivation behind this is that if loop-invariant load or linear value prediction speculation is applicable to the memory read instruction, the dependence formulated around the read instruction can be speculatively removed. Figure 5.3 illustrates such a case. A memory dependence exists between the store instruction in function `foo` and the load instruction in function `bar`, which results a dependence between callsites of `foo` and `bar` in loop `L`. Assuming that the store in `foo` writes a heap-constant value and the load in `bar` always reads a value written by the store in `foo`, a dependence from the store to the load can be speculatively removed by loop-invariant load speculation. If a memory dependence profiler reports that the load instruction in function `bar` is the only cause of the dependence between the callsites of `foo` and `bar`, i.e. if the profiler reports the read instruction itself along with execution contexts, the client of the memory dependence profiling results can tell that the dependence between callsites can be speculatively removed along with the dependence between the store in `foo` and the load in `bar`.

Algorithm 5: $MemoryDependenceProfile(C_R, LD, ITER_R, A)$

1 **let** $SA \coloneqq Translate(A)$;
2 **let** $\langle C_W, ITER_W \rangle \coloneqq unpack(*SA)$;
3 **let** $Dependence \coloneqq \langle C_W, C_R, LD, ITER_W$!= $ITER_R \rangle$;
4 **let** $Dependences \coloneqq Dependences \cup \{Dependence\}$;

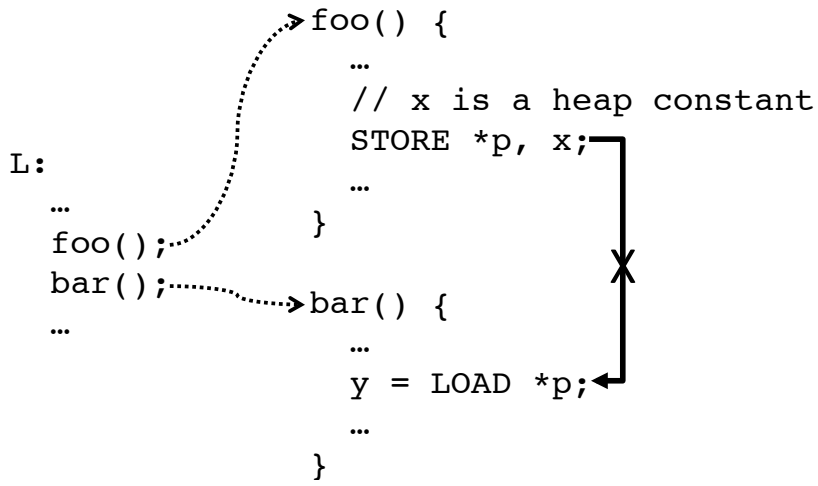


Figure 5.3: As the memory dependence profiler observes a dependence from the store instruction in `foo` to the load instruction in `bar`, it observes a dependence from the callsite of `foo` to the callsite of `bar` in loop `L` as well. However, these dependences can be speculatively removed by applying loop-invariant load speculation to the load instruction.

5.2.3 Run-time Support for Context-Sensitive Speculation

Speculative parallelization requires run-time system support to validate speculation assumptions during execution of the parallel program. The following paragraphs explain the run-time support for each type of context-sensitive speculation.

Loop-invariant load speculation The run-time system for context-sensitive loop-invariant load speculation verifies that each speculated instruction satisfies the loop-invariant load criterion described in Chapter 5.2.2 at parallel program execution time.

Profiling results can tell which instruction LD and execution context C to apply speculation to. However, for many cases, memory address A and the loaded value V can only be known at run-time. This problem can be overcome through loop peeling. The automatic parallelization system instruments the first iteration of the target loop to observe the loaded address A and the loaded value V, and emits code that saves them into the *loop-invariant table*, which is index by each pair of load instruction LD and execution context C.

Speculative iterations predict the loaded value V according to this table. Before executing the first stage of each iteration, a worker for the first stage enforces the prediction by initializing its private memory to match with the loop-invariant table: for each (A, V) in the table, it stores V to address A. This enforcement is propagated to the following stages by the SMTX run-time system.

There is no need to validate intra-iteration dependencies, as only loop-carried dependencies are removed in the speculative PDG. To validate loop-carried dependencies, it is sufficient to check i) each invocation of a speculated instruction LD under execution context C loads the value from the address in the loop-invariant table indexed by (LD, C), and ii) each pair of (A, V) within the loop-invariant table is valid. The commit process validates each pair of (A, V) in the table by loading the value of A from the speculative iteration, V', and compares V' with V after each iteration. It triggers misspeculation if $V' \neq V$.

Linear value prediction speculation To verify if each speculated instruction meets the linear value prediction criterion (Chapter 5.2.2), linear-value prediction speculation employs a table, much like loop-invariant load speculation. The first *peeled* loop iteration captures the effective address A for each speculatively-linear load instruction LD under execution context C and saves it into the linear-prediction table alongside coefficients m, b, which are provided by the profiling results. Before execution of the first stage of each iteration I, a worker 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 $V' \neq m \times (I + 1) + b$. For each invocation of instruction LD under C, the run-time checks if the loaded address matches the address stored in the linear-prediction table.

Memory dependence speculation Although the SMTX run-time system implementation did not intentionally support context-sensitivity, its memory dependence validation scheme is strong enough to be adapted to context-sensitive memory dependence speculation; SMTX replays all speculative memory operations sequentially and guarantees that all speculatively executed loads compute the same value as the non-speculated execution.

However, as context-sensitivity enables more aggressive speculation, supporting context-sensitivity increases the performance overhead of the run-time system. To support memory dependence speculation, validation instructions are inserted around speculated memory operations. These validation instructions introduce overhead both in terms of sequential latency of a single iteration and an increase in the communication among processes. Recall the example of Figure 5.2. If context-sensitive speculation is not supported (Figure 5.2(c)), no validation instructions are required simply because parallelization is not applicable. If context-sensitive speculation is supported (Figure 5.2(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 Chapter 5.3.

5.3 Optimizing Run-time System Supporting Speculative Parallelization

As described in the previous chapter, while aggressive memory dependence speculation is necessary to automatically parallelize complex programs, it increases run-time validation overheads. Excessive validation overheads may offset the parallelization benefits. This chapter describes static and dynamic optimization techniques to reduce these overheads and enable scalable parallel speedup even with aggressively speculated programs.

5.3.1 Static Optimization

Algorithm 6: Find redundant load-validation

```
1 let Redundant := ∅;
2 let AllMemOps := all load instructions in the function;
3 foreach Load l1 ∈ AllMemOps do
4     foreach Load l2 ∈ AllMemOps do
5         let SD := strictDominators(l2);
6         if l1 ∈ SD and alias(l1.ptr,l2.ptr)==MustAlias then
7             let Redundant := Redundant ∪{l2};
8 return Redundant;
```

Static optimization reduces validation overheads by making use of the observation that when one transaction accesses the same memory cell multiple times, validation is only necessary on the first load and the last store. Algorithm 6 identifies load instructions whose validation is unnecessary. If a load instruction is dominated by another load instruction,

and their loaded addresses are guaranteed to point the same memory address, validation for the load is redundant. An analogous algorithm identifies redundant stores.

Kim et al.'s static communication optimization [38] is based on the same insight. If a parallelization target loop has inner loops, their technique hoists load and store instructions out of the inner loop if the address operand is proved to a loop-invariant. The optimization proposed here does not perform hoisting, but is more generally applicable than the optimization proposed by Kim et al.

5.3.2 Dynamic Optimization

To support uncommitted value forwarding and group transaction commit, which are the two features of MTX to enable multi-threaded atomicity, the existing SMTX implementation requires expensive inter-process communication for each store instruction and each speculative load instruction in the parallelized loop. SMTX communicates the address-value pair of every store operation executed in an earlier subTX to later subTXs to implement uncommitted value forwarding. Additionally, SMTX forwards the address-value pair of every store and speculative load operation executed in a worker process to the commit process in order to enable group transaction commit.

Dynamic optimization is 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 to when each subTX communicates the values to be committed to the commit process. Second, checking the pipeline scheduling assumption is sufficient to validate memory dependence speculation. To validate speculation, the existing SMTX commit thread replays every store and speculated load instruction sequentially and compares the result with the result from speculatively parallel execution. However, so long as no loop-carried dependences manifest from a later subTX to an earlier subTX (or within a subTX if the subTX is running a parallel stage), it is safe to assume that there is no memory dependence misspeculation, given that intra-iteration dependencies are not

speculatively removed in the PDG.

Exploiting these insights, the optimized run-time system minimizes 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 corresponds to 3-bits of metadata in the shadow memory. Each bit indicates if the byte is *read*, *written*, or *read-before-write* during the subTX execution. If i) *read-before-write* bit is set, or ii) *read* bit is set but *written* bit is not set, it implies that the corresponding byte is not written by the subTX but read within the subTX. For such a case, the system needs to guarantee that the byte has not been written by a later subTX (or by the same subTX for a different iteration if the subTX is running a parallel stage) to validate memory dependence speculation.

Figure 5.4 describes the run-time validation functions to be executed for every speculative memory read (`mtx_read`) and write (`mtx_write`) in the parallelization region. For reads, the instrumentation simply marks the metadata as *read*. For writes, instrumentation marks the metadata as *written*, and if the *read* bit has been set but the *written* bit has not been set, it also sets the *read-before-write* bit. As only a small number of bit operations and memory operations are added to each dynamic memory operation, the validation overhead is much smaller than performing inter-process communication. The shadow memory address can be computed with a single bit-wise XOR instruction with the help of customized memory allocation functions. The run-time functions are designed to be more efficient for memory reads than writes because there are generally more reads than writes in the program [36].

With the optimized run-time system, inter-process communication happens only at the 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

```
#define READ              0x04;
#define WRITTEN           0x02;
#define READ_BEFORE_WRITE 0x01;
#define GET_SHADOW_OF(x) ((ADDR)(x)^SHADOW_MASK);

/* Validation function for speculative reads */

void mtx_read(Byte* ptr) {
  Byte* shadow = (Byte*)GET_SHADOW_OF(PTR);
  *shadow = *shadow | READ;
}

/* Validation function for writes */

void mtx_write(Byte* ptr) {
  Byte* shadow = (Byte*)GET_SHADOW_OF(ptr);

  Byte mask1 = *shadow >> 2;
  Byte mask2 = ~(*shadow >> 1);

  // RBW is set only when READ bit has been set
  // but WRITE bit has not been set
  Byte RBW = mask1 & mask2 & READ_BEFORE_WRITE;

  *shadow = *shadow | WRITTEN | RBW;
}
```

Figure 5.4: Validation functions for speculative reads and writes within the parallelized region

later subTX receives the pages, it scans the shadow pages to find the values that have been written by the earlier subTX and updates its private memory accordingly. Uncommitted value forwarding is accomplished via this update.

Accessed pages and their corresponding shadow pages are forwarded to the commit process as well. The commit process analyzes subTXs and processes pages communicated from each subTX in order. The processing algorithm is described in Figure 5.5. If the metadata for a byte indicates that the byte is i) only read, or ii) was read before being written during the subTX execution, the commit process checks if misspeculation occurred. The commit process maintains its own shadow memory that holds the ID of the subTX that last wrote to memory at byte-level granularity. With this shadow information, the commit process compares the IDs of the subTX to ensure that the memory write occurred in an earlier subTX than the memory read. Otherwise, misspeculation has occurred. If the metadata forwarded from the worker process indicates that a byte is written, the commit process updates its own memory with the written value and writes the subTX ID to the shadow memory. If no misspeculation occurred throughout the transaction, the commit process writes its memory state to the global state. This write represents group transaction commit.

Figure 5.6 explains how the optimized run-time system discovers memory dependence misspeculation. Figure 5.6(a) shows a parallelization target loop. With the assumption that the loop-carried dependence between the store and the load in Figure 5.6(a) can be speculatively removed, pipeline parallelization is applicable to the loop. Figure 5.6(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 5.6(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

```
void mtx_commit(
    /* address of the page to be processed */
    Byte* addr,
    /* contents of the page coming from the worker process */
    Byte* data[PAGE_SIZE],
    /* metadata of the page coming from the worker process */
    Byte* metadata[PAGE_SIZE],
    /* ID of the subTx that running on the worker process */
    ID subTX
) {
  Bool isPStage = isParallelStage(stage);

  for (unsigned i = 0 ; i < PAGE_SIZE ; i++) {
    Byte md = metadata[i];
    Bool rbw = md & READ_BEFORE_WRITE;
    Bool written = md & WRITTEN;
    Bool read = md & READ;
    Bool validate = rbw || (read && !written);

    if (validate) {
      Byte* pShadow = (Byte*)GET_SHADOW_OF(&addr[i]);
      ID    writtenSubTX = (ID)(*pShadow);
      Bool  misspec = *writtenSubTX > stage;

      if (isPStage && (*writtenSubTX == stage))
        misspec = true;

      if (misspec)
        mtx_misspeculation("Memory Dependence Misspeculation");
    }
    if (written) {
      addr[i] = data[i];
      Byte* pShadow = (Byte*)GET_SHADOW_OF(&addr[i]);
      *pShadow = (Byte)subTX;
    }
  }
}
```

Figure 5.5: Algorithm for the commit stage. Definitions of READ, WRITE, READ_BEFORE_WRITE, and GET_SHADOW_OF are same as the ones in Figure 5.4

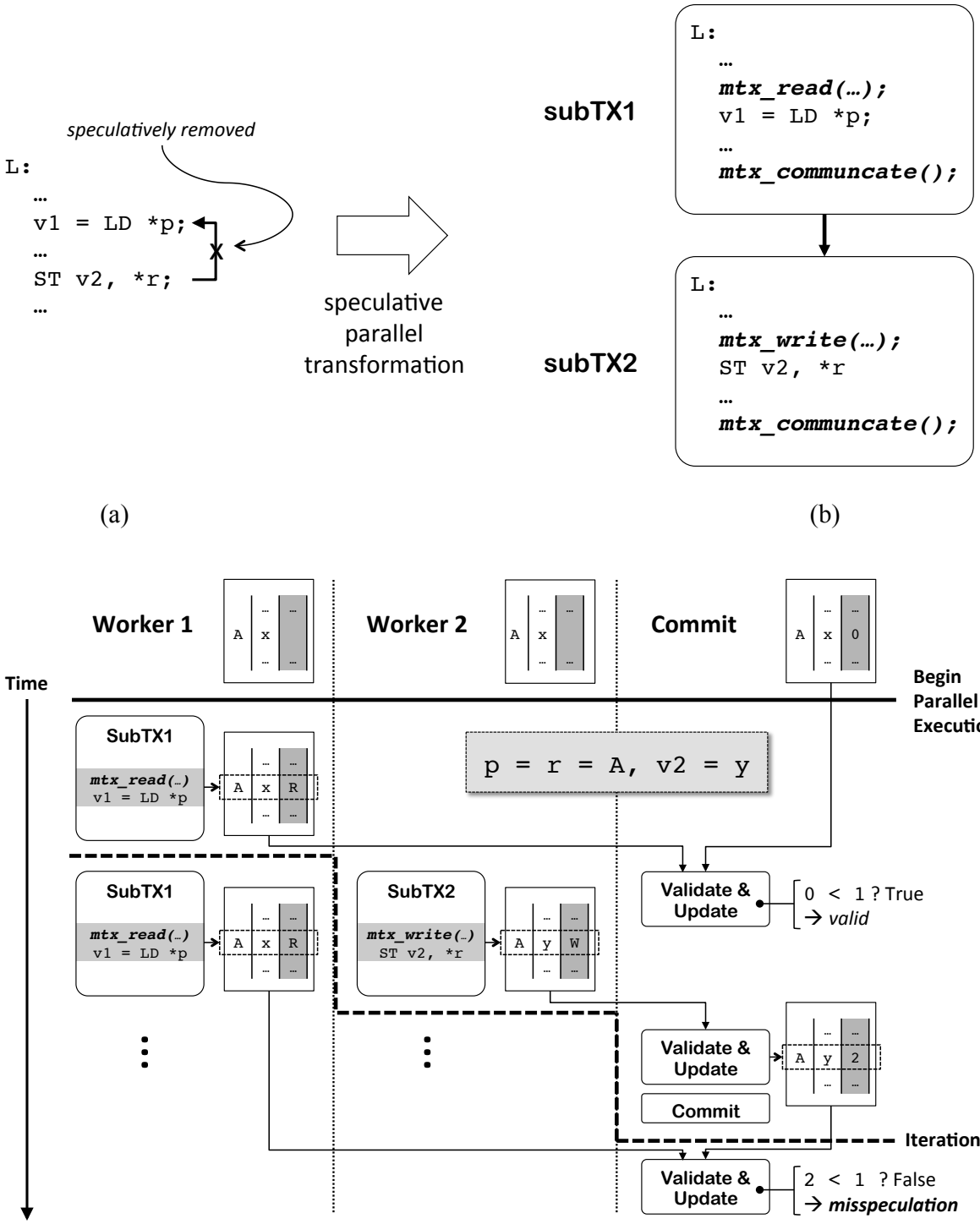


Figure 5.6: Example demonstrating how the commit process detects misspeculation. (a) Parallelization target loop. (b) Multi-threaded loop using the optimized run-time system. (c) A schematic time line of parallel execution. Rectangular boxes next to subTXs represents the memory state of each process at a given time. Shadowed column indicates shadow memory for each byte.

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; 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. Forwarded information tells 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 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. The implementation of misspeculation recovery follows the algorithm described in [74].

There are two further improvements that have been applied to the optimized run-time system. If the entire page is read-only, there is no need to communicate the entire page and its corresponding shadow page. Communicating the address of the page to the commit process is sufficient to validate memory dependence speculation. In addition, if only a small fraction of the page is accessed during the subTX execution, communicating only the accessed fraction can be more efficient than communicating the entire page. This is a trade-off between the total number of communications and the total number of bytes communicated. Forwarding the entire page can be performed with a single jumbo packet, while finer-granularity forwarding will require multiple communications. In the current implementation, the entire page is sent only when more than 64 bytes of the 4K sized page are accessed.

Chapter 6

Evaluation

The system proposed in this dissertation is evaluated on a shared-memory machine with four 6-core Intel Xeon X7460 processors (24 cores total) running at 2.66 GHz and 24 GB of memory. It runs 64-bit Ubuntu 9.10. The compiler is implemented in the LLVM compiler framework [43] revision 164307.

Two open-source C programs are used to evaluate the system: the Lua script interpreter version 5.2.3 [47] and the Perl script interpreter 5.20.1 [65]. IPLS specializes each program against 6 input scripts. The IPLS-specialized interpreter is then parallelized by the automatic parallelization system. Neither interpreter was modified, though Perl was compiled with non-default configuration options (see Section 6.5). 6 programs were chosen from Polybench [66] that are known to be amenable to DOALL parallelism [38] and reimplemented as Lua and Perl scripts.

Table 6.3 describes the six input scripts for the Lua and Perl interpreters that were used to test the system. In each case, the main interpreter loop has been specialized and parallelized, resulting in a large amount of coverage for each program. The table also describes the input to the script that is used in profiling and evaluation. *train-small* is for the profilers with relatively high execution overhead, including the IPLS profiler and the memory dependence profiler. *train-large* is used to run light-weight profilers, such as

Input Script	P'loops	Coverage	Size (LLVMIRs)	Input to the Script train-small	train-large	ref
Lua-5.2.3 (20,258 LOC, Interpreter main loop: 2,499 LLVMIRs)						
2mm	2	96.00%	1,028 / 973	32	256	512
3mm	3	96.61%	973 / 973 / 973	32	256	512
correlation	1	95.17%	1,224	50	400	800
covariance	1	93.14%	1,231	50	400	800
doitgen	1	95.58%	1,337	16	64	100
gemm	1	94.43%	2,229	32	256	512
Perl-5.20.1 (296,166 LOC, Interpreter main loop: 8 LLVMIRs)						
2mm	2	96.22%	927 / 1,087	32	256	512
3mm	3	95.17%	1,140 / 1,063 / 927	32	256	512
correlation	1	94.90%	1,427	50	400	800
covariance	1	90.73%	1,562	50	400	800
doitgen	1	95.28%	1,332	16	64	100
gemm	1	89.44%	2,209	32	256	512

Table 6.1: 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 runtime spent in the parallelized loops compared to total program execution time. *Size* denotes the original size of the parallelized loops, in units of LLVM IR instructions. *train-small*, *train-large*, and *ref* denotes the input to the script for heavy-weight profilers, light-weight profilers, and the actual evaluation executions, respectively.

control-flow and value profilers. *ref* is used in actual evaluation runs. The input to the script determines the size of the data structure to be computed (e.g. size of matrices to be multiplied).

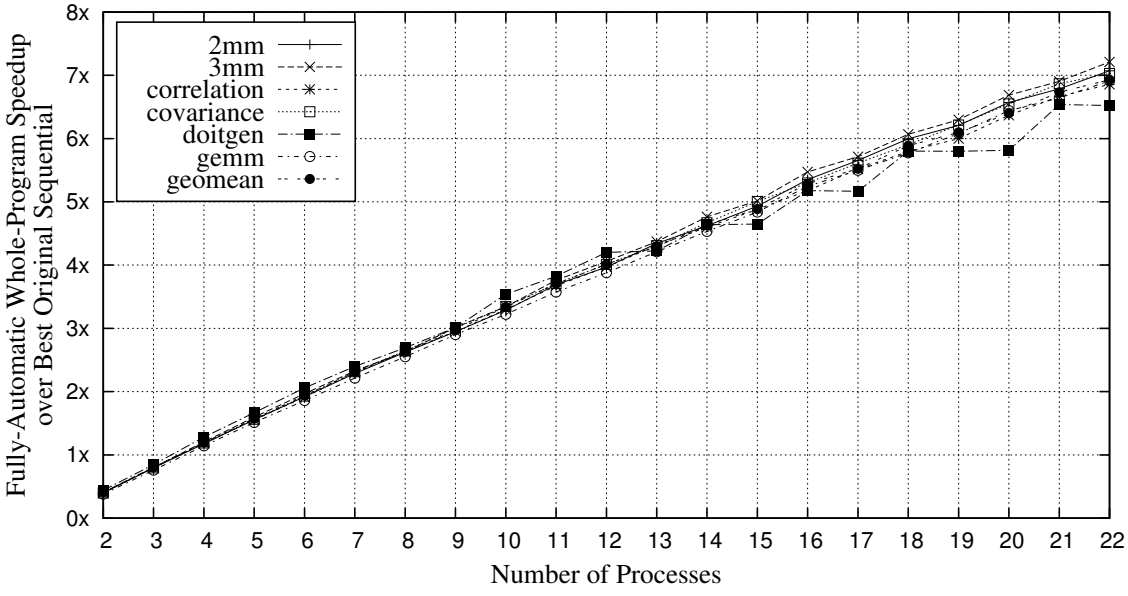
Note that the main interpreter loop of Perl contains only 8 LLVM IR instructions due to the use of indirect function calls indexed by the instruction OP-code. Without specialization, this loop would be exceedingly difficult to parallelize. By exploiting fixed inputs, the proposed system is able to build and then parallelize a specialized loop for each input.

6.1 Performance Results

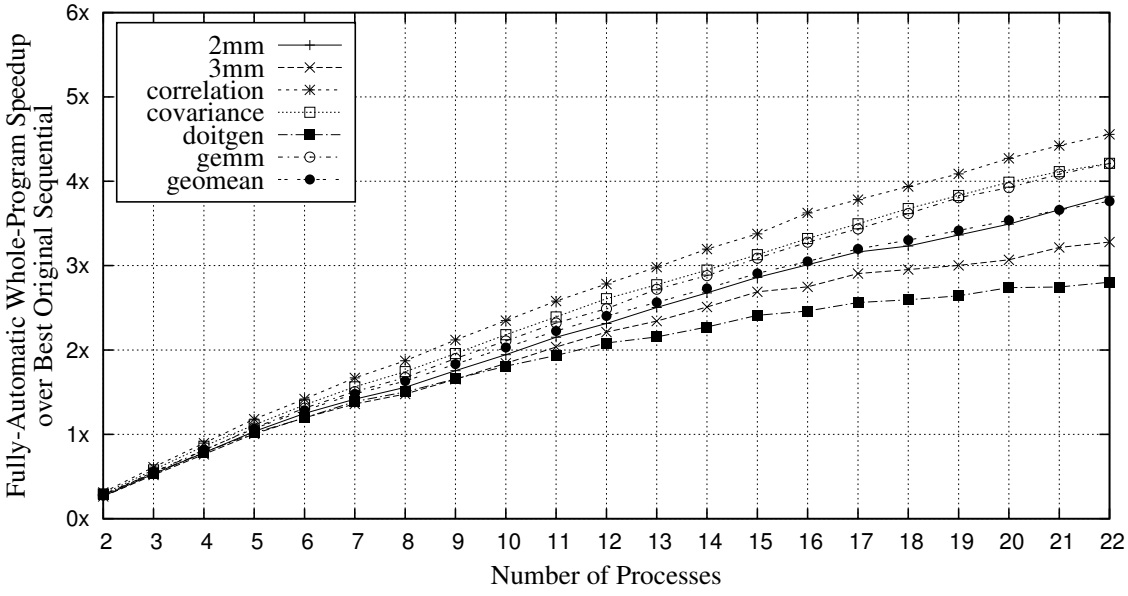
Figure 6.1 presents whole-program speedup. These speedups are normalized against the sequential, unspecialized version of the Lua and Perl interpreters compiled with *clang -O3*. The majority of the performance improvement is due to parallelization. Specialization provides up to 24% and 19% sequential improvement over Lua and Perl performance, respectively.

The DOALL parallelism in the input scripts becomes pipeline parallelism in all 12 specialized programs (6 for each of Lua and Perl). PS-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, which are loaded from memory. The parallel stage handles the actual workload of the input script; the presence of a parallel stage is consistent with exploiting the DOALL parallelism from the input script. Manual inspection confirms that the automatically parallelized code executes iterations of the input script across several cores concurrently. Since the parallel stage contains the majority of each iteration's execution time, this parallelization scales well.

Across all scripts, parallelization achieves greater improvement on Lua than on Perl due



(a) Lua-5.2.3



(b) Perl-5.20.1

Figure 6.1: 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 worker processes excluding the commit process.

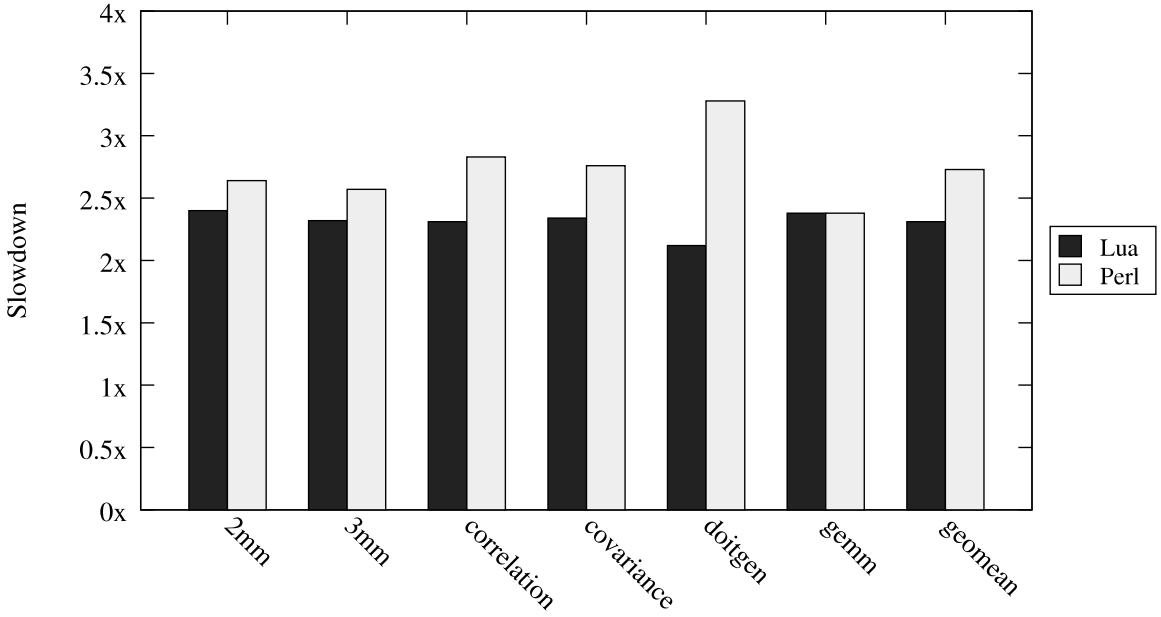
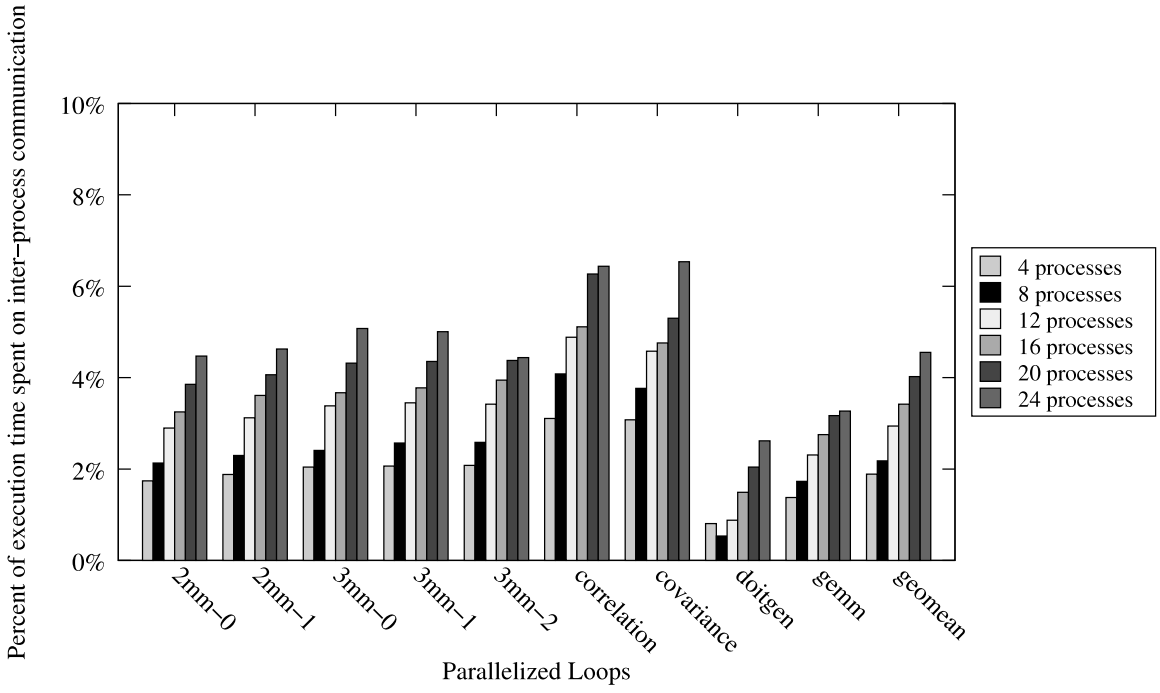


Figure 6.2: Sequential slowdown after inserting speculation checks

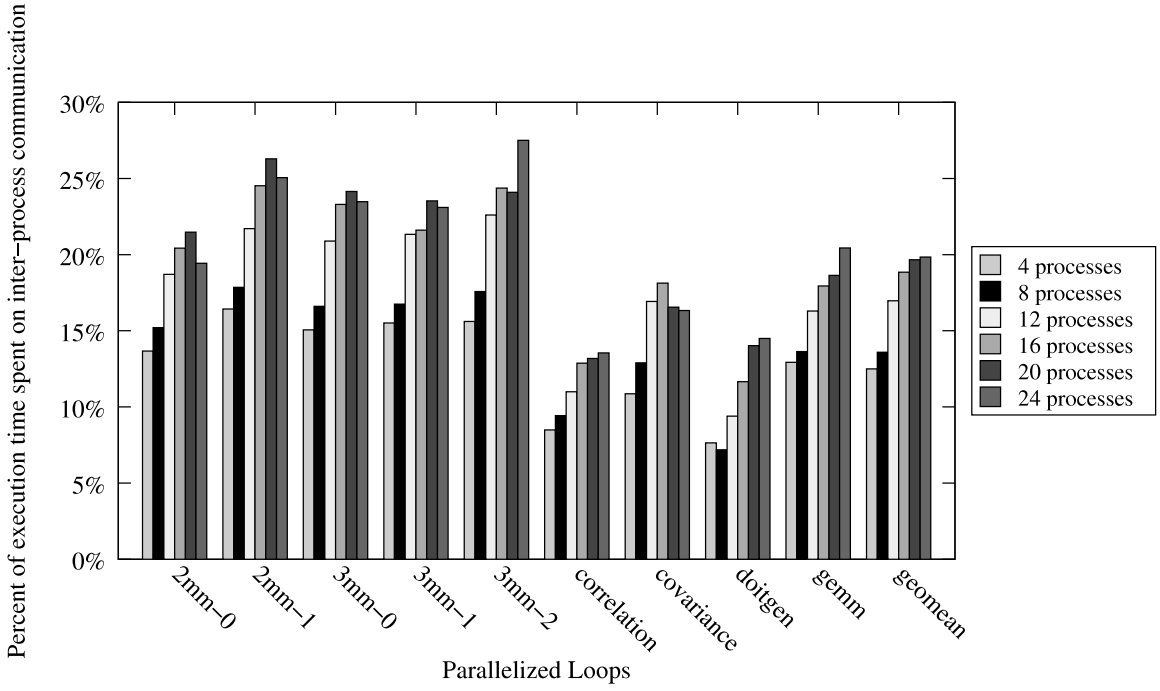
to lower speculation validation overhead for Lua. There are two types of validation overheads; one is the overhead from executing validation instructions that are inserted for each speculated memory operation, and the other is the overhead observed at subTX boundaries to communicate data and their corresponding metadata. Perl experiences higher overhead than Lua for both types of overheads.

Figure 6.2 shows that the overhead of executing validation instructions is higher for Perl than Lua. The numbers in the figure represent sequential slowdown (i.e., executing the program after inserting speculation checks but before parallelization) due to validation instructions. For Perl the geomean slowdown was 2.73×, while for Lua it was 2.31×. The slowdown was greater for Perl across all scripts.

Communication overhead is higher for Perl than Lua as well. Figure 6.3 presents communication overhead as a percentage of the execution capacity that parallel worker processes spend at subTX boundaries when using 4, 8, 12, 16, and 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 when



(a) Lua-5.2.3



(b) Perl-5.20.1

Figure 6.3: Percentage of the parallel execution capacity that parallel workers spend on inter-process communication. The number is averaged across all parallel worker processes.

using 24 cores while Perl spent 19.84%.

There are three primary components contributing the communication overhead observed at subTX boundaries of worker processes. First is the overhead to forward pages, and their corresponding shadow pages, that are accessed during execution of the subTX to later subTXs and the commit process. Second, there are overheads to update process's local memory based on the data coming from earlier subTXs, which is required for uncommitted value forwarding. Third is the overhead to set the protection of all allocated pages. In order to keep track of pages that are accessed during the subTX execution, the current implementation of the run-time system sets the permissions of all allocated pages to `PROT_NONE` at the beginning of each subTX. When the page is accessed for the first time in the subTX, a page fault occurs and a custom interrupt handler is invoked. The interrupt handler marks the page as *touched*, then retrieves the original protection of the page. Under the assumption that a single page is generally accessed multiple times during the subTX execution, this implementation is more efficient than the alternative approach where every dynamic memory operation marks the accessed page, because the alternative approach performs the marking operations redundantly if the same page is accessed multiple times in the subTX.

Perl experiences higher overheads for all three components of overhead at subTX boundaries. The overhead of forwarding pages is directly related to the number of pages accessed during the subTX execution. When averaging across all input scripts, the worker processes of Perl access 9,066,011 dynamic pages during the program execution. For Lua, the worker processes access only 765,532 dynamic pages. The overhead of processing incoming data is dominated by number of incoming pages from earlier subTXs. While Perl's parallel stage worker processes receive 4,140 dynamic pages on average from earlier stages, Lua's parallel worker processes receive only 1,026 dynamic pages when averaged across all scripts. The overhead of setting the page protection at the beginning of the subTX is determined by the total number of allocated pages when subTX begins. For Perl, each subTX observes 27,551 pages at the beginning on average. For Lua, it is 8,986 pages.

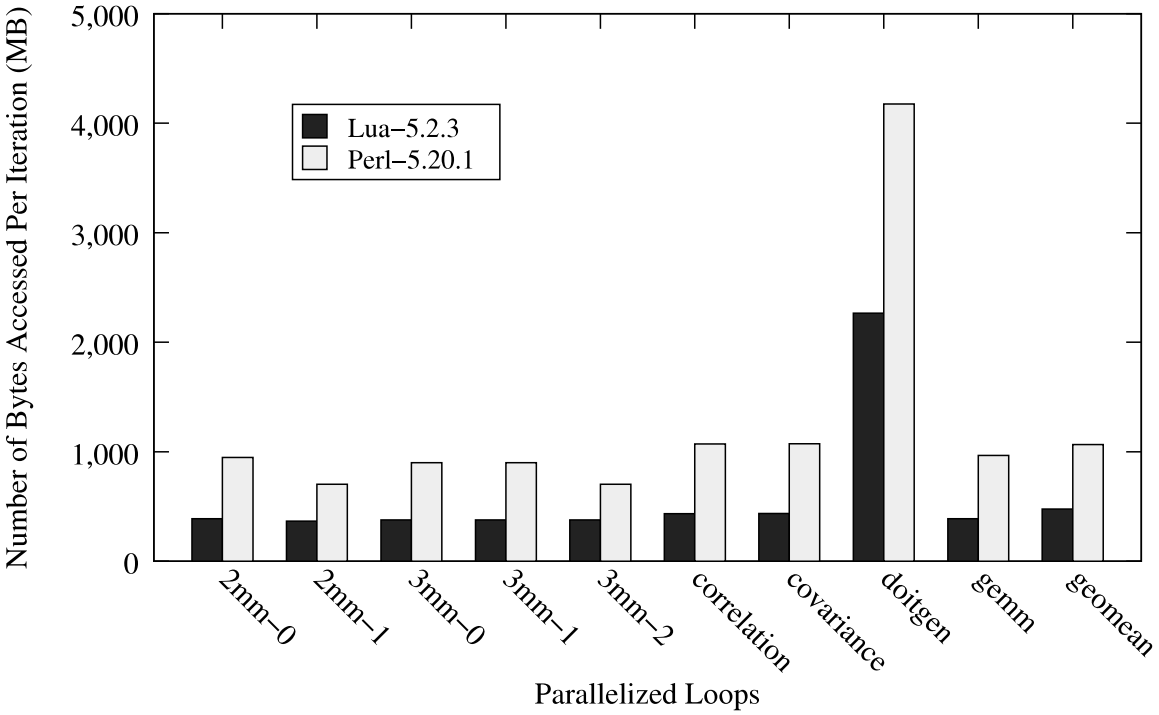
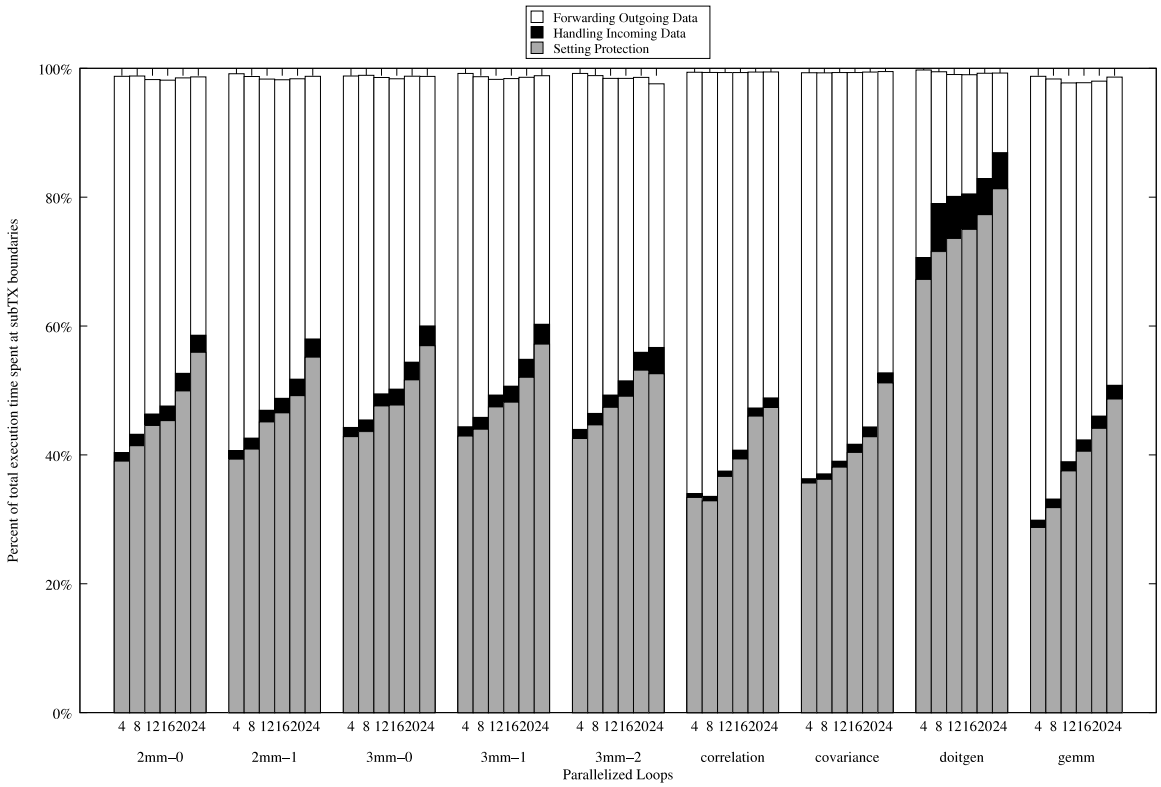


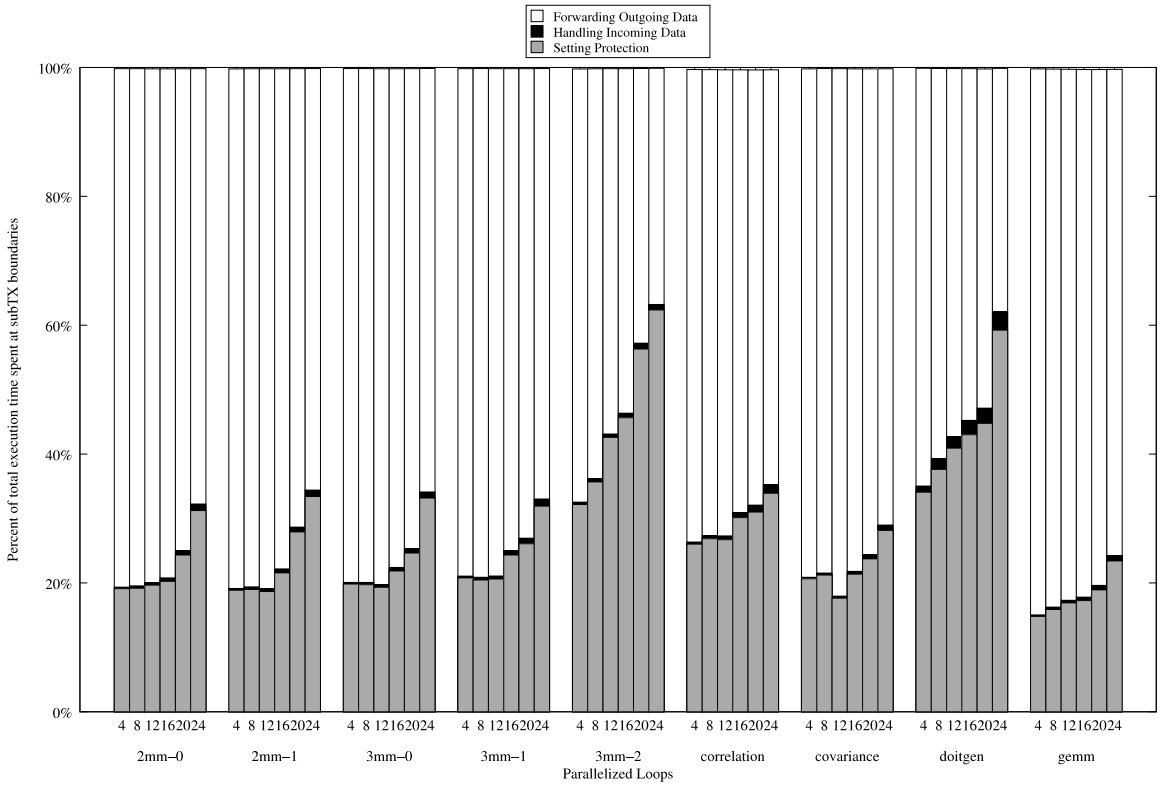
Figure 6.4: Number of bytes accessed (in MB) per iteration for each parallelized loop

The higher communication overhead for Perl can be explained by memory requirements. All three components affecting the communication overhead are likely to be increased if the sequential program requires more memory. Figure 6.4 compares the number of accessed bytes per iteration of the loops targeted for parallelization after they are specialized for the same input script and the same input to the script. Perl accesses 2.23× more bytes by geomean than Lua, implying that Perl consumes much more memory than Lua to run the same algorithm (the sequential version of Perl takes at most 26% more time than the sequential version of Lua across the same input sets). This suggests that memory efficiency of the sequential program greatly affects parallel performance.

Figure 6.5 shows the contribution of each component to the total execution time spent on the subTX boundaries. The figure confirms that the sum of the three components mentioned above takes almost 100% of the execution time spent on the subTX boundaries. The figure also shows that the *fraction* of the overhead for setting page protection increases as the number of processes increases. This suggests that the overhead for setting page protec-



(a) Lua-5.2.3



(b) Perl-5.20.1

Figure 6.5: Fraction of execution time spent on subtransaction boundaries.

tion in each parallel process remains constant even when the number of workers running the parallel stage increases. As the number of worker processes assigned to the parallel stage increases, the overhead for handling incoming and outgoing pages is reduced for each process since each worker process only touches the pages relevant to the subTX it runs. However, each worker process needs to set the protection of all allocated pages, regardless of the parallel factor, in order to detect relevant pages.

Thus, the scalability of parallel speedup is most sensitive to the overhead of setting the protection for every allocated page at the beginning of each subTX. Figure 6.5(a) shows that the fraction of the overhead for setting protection is exceptionally high in *doitgen* compared to other scripts. In Figure 6.1(a), all Lua scripts except *doitgen* show similar scalability, while *doitgen*'s scalability is slightly worse than the others. This is consistent with Perl. *3mm* and *doitgen*, the two scripts that contain loops with the highest fraction of protection setting overhead in Figure 6.5(b), present distinctly worse scalability than the other scripts. High memory bandwidth requirement of *doitgen* shown in Figure 6.4 implies the poor scalability of *doitgen* for both Perl and Lua.

The difference in speedup between distinct input scripts for Perl can be explained by considering the overhead of executing validation instructions and communication overhead. Except doitgen, all scripts have similar overhead for executing validation instructions, as seen in Figure 6.2. The speedup rank between these scripts using 24 processors approximately follows the rank of communication overhead shown in Figure 6.3(b). *2mm* and *3mm*, scripts which experience much higher communication overhead than others, show less speedup. *gemm* has relatively high communication overhead, but its overhead for executing validation instructions is the lowest. *doitgen* has relatively low communication overhead. This is because the parallelized loop in *doitgen*' is quadruple-nested, while all other parallelized loops are triply nested. Consequently, *doitgen* spends relatively less time at subTX boundaries. However, it does not mean that *doitgen*'s absolute communication overhead is low. Figure 6.4 suggests that the absolute communication overhead of *doitgen*

Program	Input Script	Bytes Accessed (MB)	Bytes Communicated (MB)
Lua	2mm	193674.7	36.3
	3mm	193344.4	38.3
	correlation	348311.0	172.3
	covariance	348883.4	144.1
	doitgen	226535.9	73.4
	gemm	199580.7	38.2
Perl	2mm	422705.7	288.9
	3mm	427264.7	288.5
	correlation	857042.7	278.1
	covariance	858525.6	5256.9
	doitgen	417550.5	426.1
	gemm	494880.1	258.7

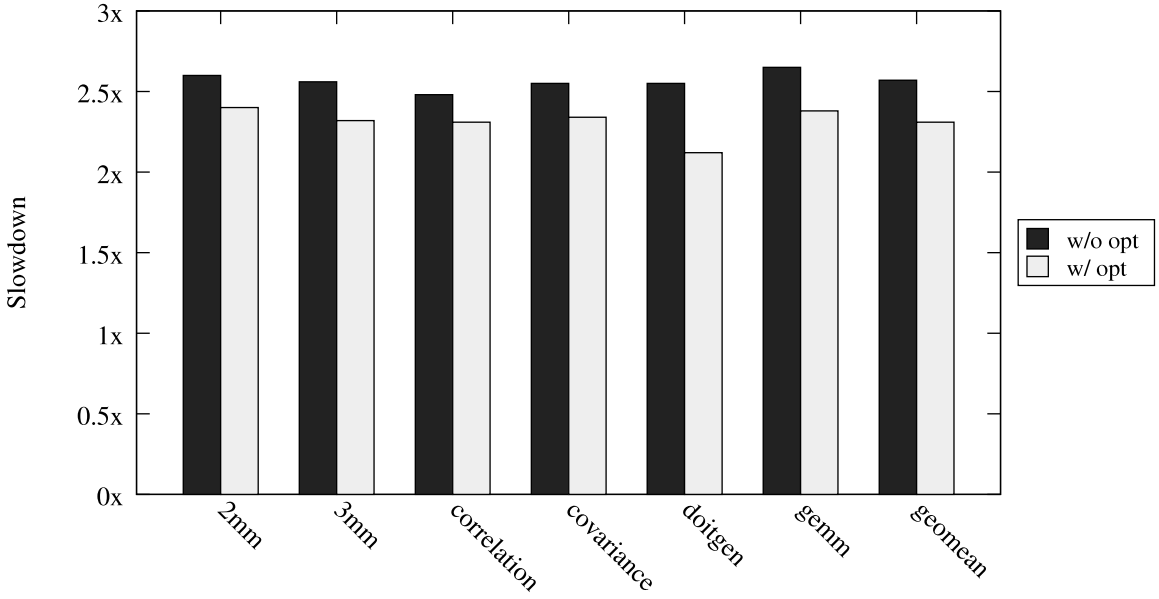
Table 6.2: Total accessed bytes and total communicated bytes during the parallel program execution

is likely to be greater than any other scripts.

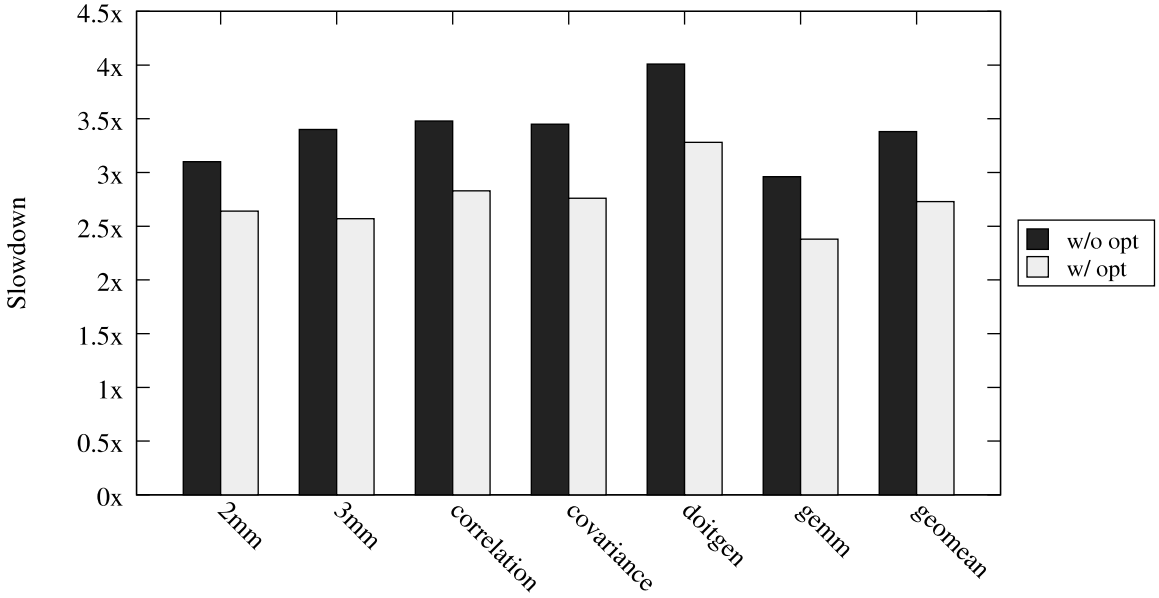
6.2 Optimization of Speculation Validation

The numbers in Figure 6.6 represent the slowdown of sequential execution with validation instructions, i.e., execution after inserting speculation validation instructions but before parallelizing, with and without static validation optimization (see Chapter 5.3.1). The static optimization reduces overhead by approximately 11% for Lua and 23% for Perl. The benefit of static optimization is likely to increase if a program executes more memory operations. Perl benefits more than Lua because Perl executes about 72% more memory operations at run-time than Lua to run a script describing the same algorithm.

Figure 6.7 presents the effectiveness of static validation optimization in terms of application speedup. When running on 24 processors, the optimization results in approximately 5.8% speedup for Lua and 4.0% for Perl. Though Perl benefits more from the optimization for sequential execution, Lua benefits more for parallel executions. This implies that the communication overhead dominates the validation overhead.



(a) Lua-5.2.3



(b) Perl-5.20.1

Figure 6.6: Overhead of memory dependence checking instructions before and after the static optimization

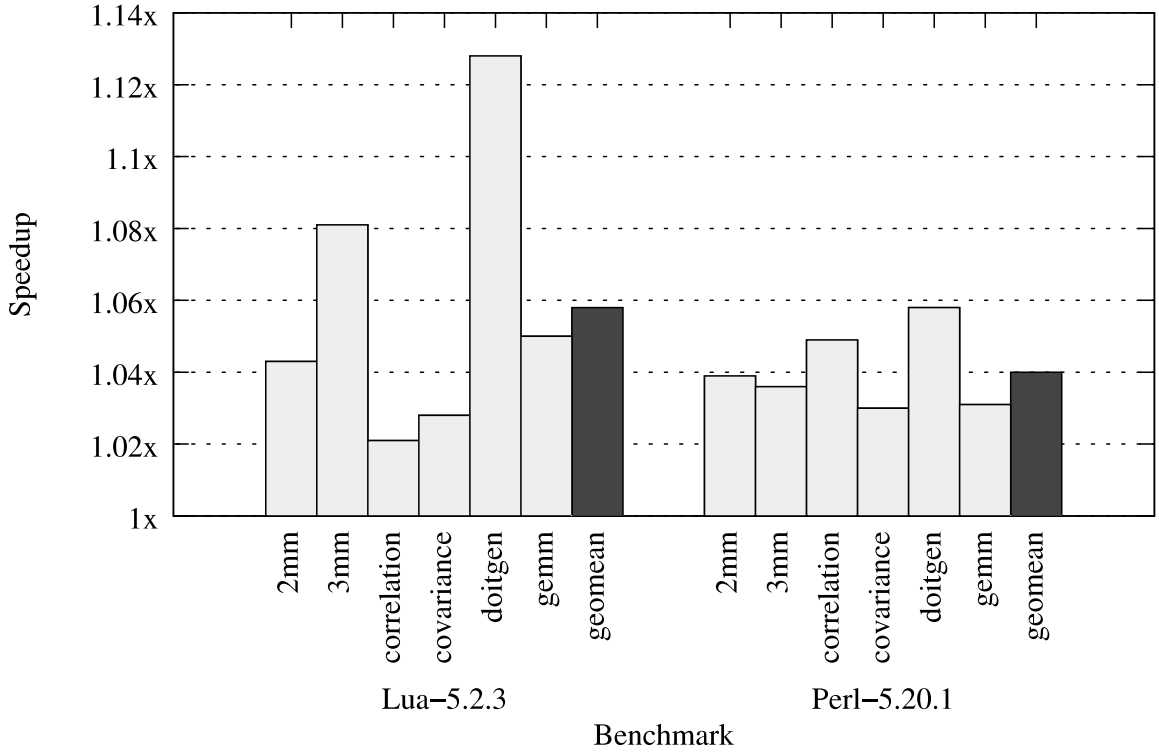


Figure 6.7: Effect of memory dependence speculation optimization for parallel executions

Table 6.2 shows the effect of the dynamic optimizations for the run-time system proposed in this dissertation. The table presents the total number of bytes accessed by the program and total number of bytes communicated between processes during parallel execution. Based on the fact that dependence analysis of the proposed system relies heavily on speculation, the total number of communicated bytes would be **at least** as much as the total number of bytes accessed if the dynamic optimization is not applied to the run-time system. This is because the existing run-time system supporting multi-threaded transactions [74] requires inter-process communication for every speculative load and store. The table shows that the dynamic optimization reduces the data communication by several orders of magnitude.

The optimization for read-only pages described in Chapter 5.3.2 greatly reduces the amount of inter-process data communication. With the optimization, if a page is only read during the subTX execution, the run-time system communicates the address of the page rather than sending the entire page and the corresponding shadow page. In the current

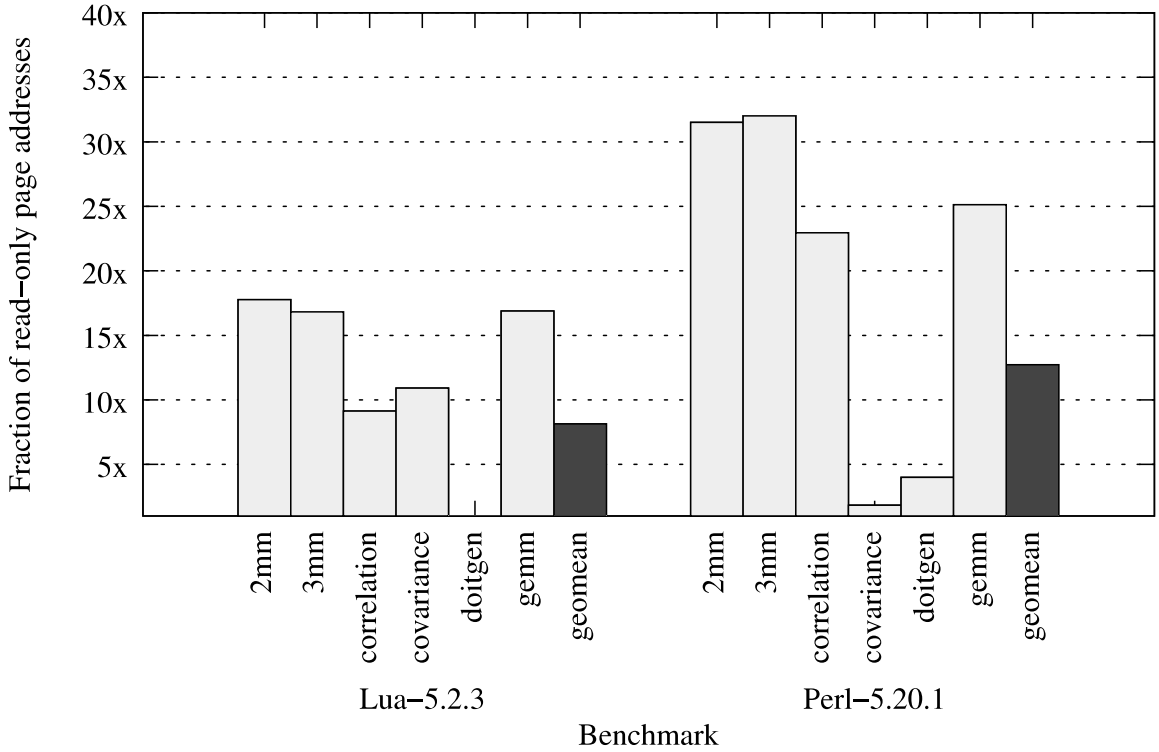
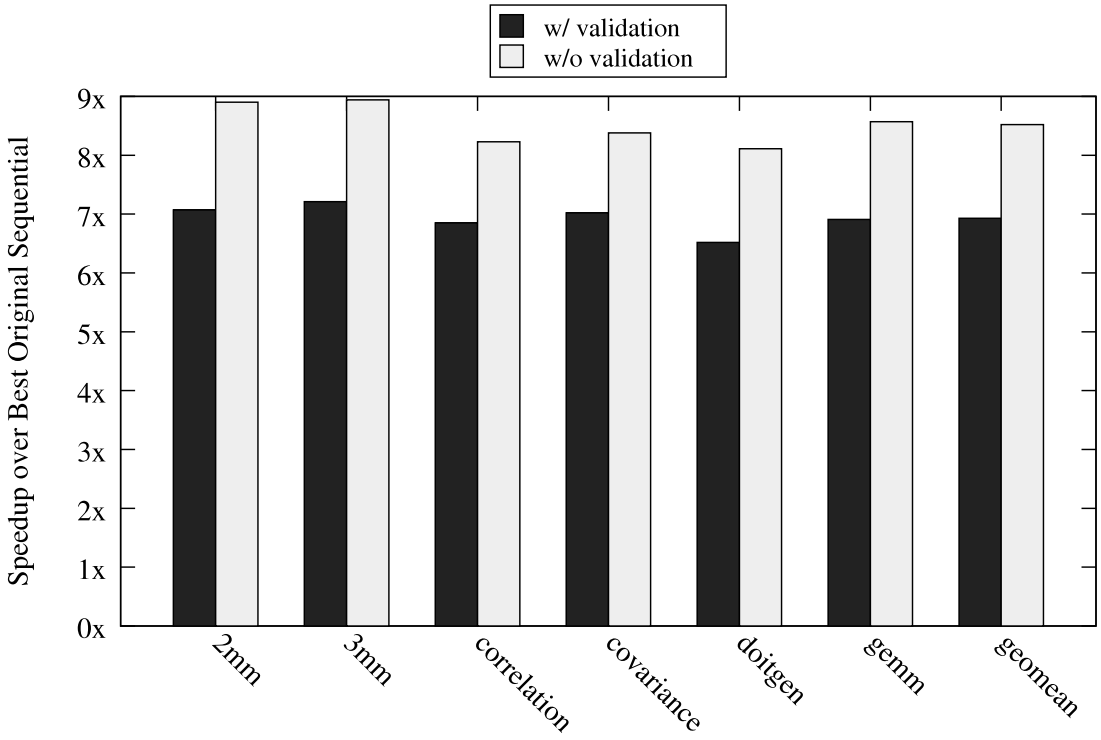


Figure 6.8: Fraction of read-only page addresses in total communicated bytes.

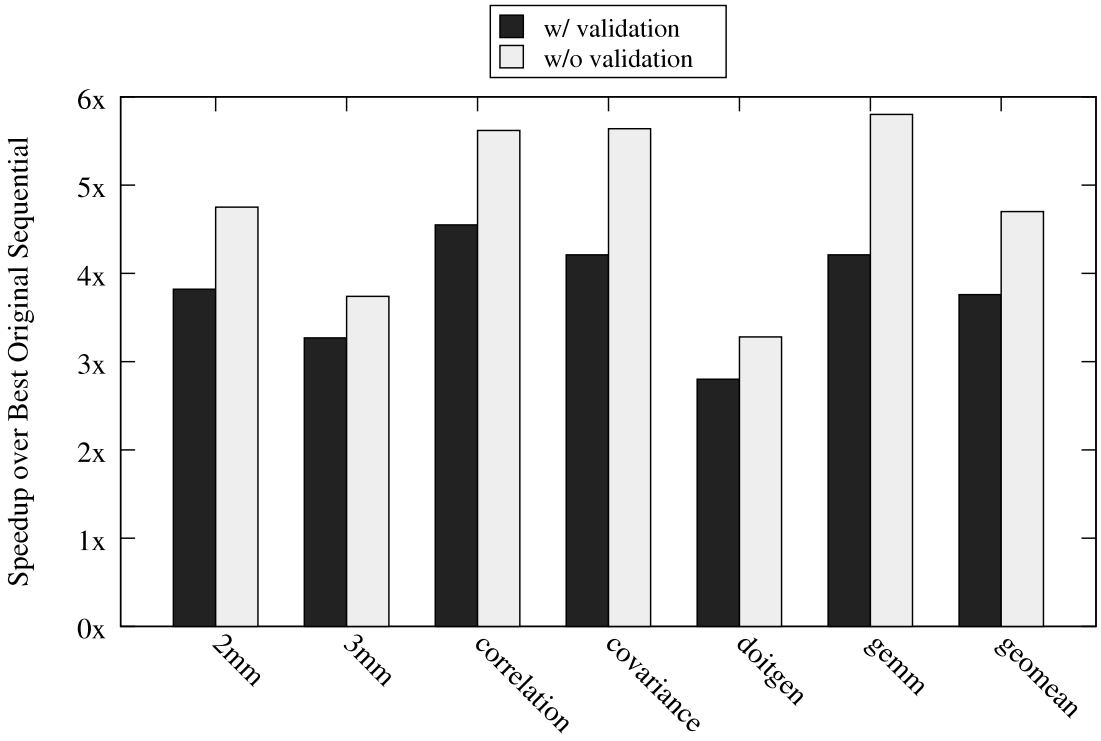
implementation, the read-only optimization reduces the amount of communication from 8196 bytes to 8 bytes for each page. Figure 6.8 shows that 8.1% (Lua) and 12.7% (Perl) of inter-process communication bandwidth is used to forward read-only page addresses, in geomean. This implies that the total number of communicated bytes could be around 100× larger than in Table 6.2 without the optimization for read-only pages.

6.3 Limit Study

Figure 6.9 presents the speedup that could be achieved if there is no unnecessary validation overhead. To compute these numbers, the compiler gets feedback from the original parallel execution and generates a version that skips the validation instrumentation of load instructions if the load never violates speculation assumptions, even with the *ref* input. In other words, these numbers are based on the *oracle* analysis that perfectly predicts the run-time behavior. As no memory dependence misspeculation has happened across all evaluated



(a) Lua-5.2.3



(b) Perl-5.20.1

Figure 6.9: Whole-program speedup using 24 processes compared to the sequential, unspecialized version compiled with `-O3`, with and without validation overheads.

programs, no validation instructions for loads are inserted. Speedup numbers with validation overhead are presented in Figure 6.9 as well for comparison.

Without validation overhead, the geomean speedups are 8.52× for Lua and 4.70× for Perl using 24 processes. Considering that the parallel stage dominates the program execution time, and 21 among 24 processes are dedicated to the parallel stage, these numbers are still much lower than the optimal speedup of 21×. This is because each worker runs in a separate *process* and has its own, separate address space. Value forwarding from the process running the earlier stage to the process running the later stage is required regardless of whether speculation is used or not. To support value forwarding, store instructions still need to be instrumented if the value written by the store is read by later stages. Moreover, process based parallelization requires all live-out values to be merged to the main process. This means that all stores that write live-out values need to be instrumented. Additionally, inter-process communication for live-out values is necessary. If the run-time system is designed in a way that all workers share the same address space, better speedup can be achieved with the oracle analysis. However, there is no practical reason to design a run-time system that assumes the availability of oracle analysis.

Figure 6.10 and Figure 6.11 support the numbers in Figure 6.9. Figure 6.10 shows the sequential slowdown of the program when only store instructions in the parallelization target loops are instrumented. Based on the assumption that every store contributes to either value communication between stages or updating live-out values, the numbers in the figure approximate the overheads resulting from the necessary instrumentation even with the oracle analysis. Figure 6.11 presents the fraction of i) value forwarding between pipeline stages and ii) live-out forwarding to the main process in the amount of total data communication. As mentioned above, communication for these two is unavoidable even when the oracle analysis is available. The figure shows that with the oracle analysis the amount of data communication can be reduced to 65.8% and 36.7% in geomean for Lua and Perl, respectively.

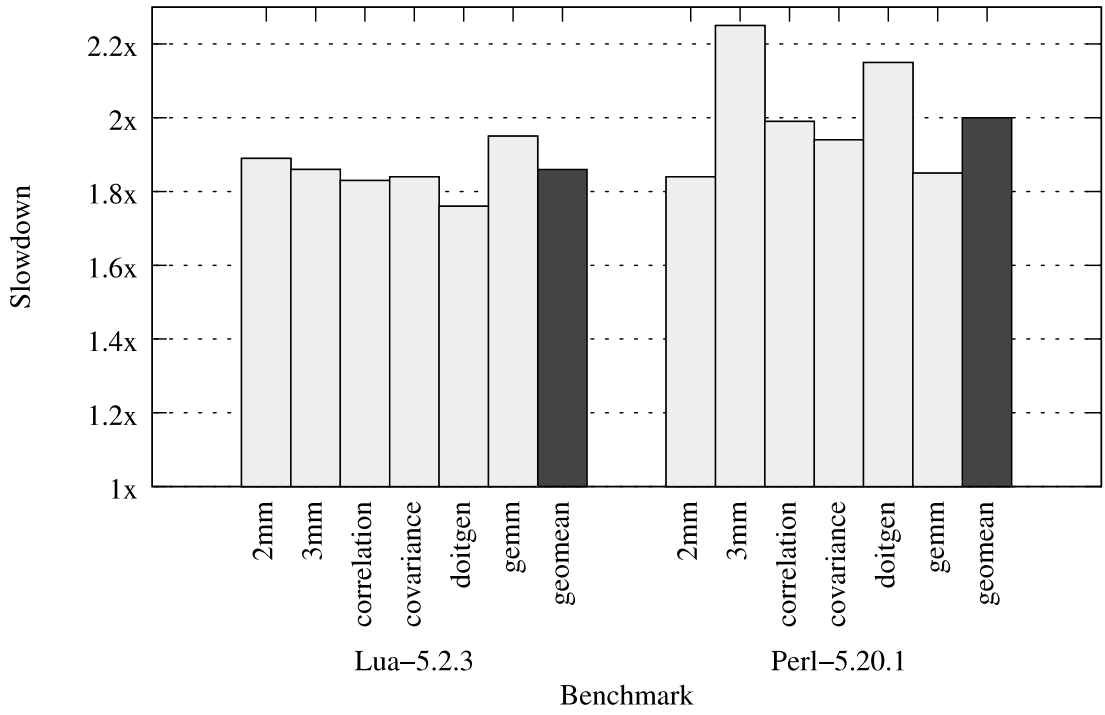


Figure 6.10: Sequential slowdown after inserting speculation checks assuming oracle analysis.

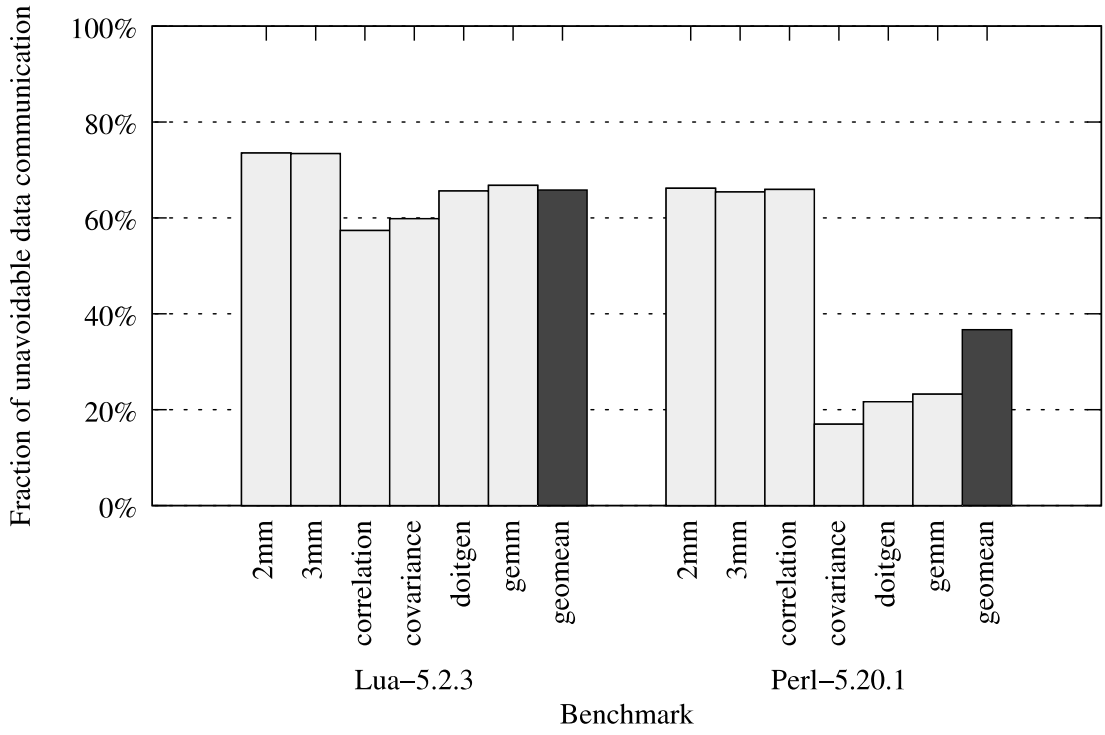


Figure 6.11: Fraction of *unavoidable* data communication, including value forwarding between pipeline stages, and live-out forwarding to the main process

6.4 Performance Optimization Effect of IPLS

Although IPLS is used as an enabling transformation for automatic parallelization in the proposed system, IPLS can be used as a standalone program specializer to improve program performance.

Standalone IPLS is evaluated by specializing Lua, Perl, and Python script interpreters [71] against eleven input scripts. The eleven input programs are selected from the Computer Language Benchmarks Game [16], which are commonly available for all three interpreters and single-threaded (IPLS does not support specialization of multi-threaded programs). Table 6.3 describes the characteristics of each interpreter and each input script.

Figure 6.12 shows whole program speedup for the programs specialized with IPLS over the original, non-specialized program compiled with `clang -O3`. Figure 6.13 depicts the program size increase after specialization. As shown, IPLS achieves a geomean speedup of 14.1% in program execution with 7.0% of program size increase.

Speedup of the specialized program correlates to the fraction of iterations executed in the specialized loop. Table 6.3 shows this trend. The *Iteration coverage* column shows the fraction of all iterations of the main loop which execute within specialized code. For Lua, there is a clear distinction between cases where iteration coverage is less than 3% and cases where iteration coverage is greater than 70%. In the latter case, specialization yields 7.4%–138% speedups, while in the former case, specialization yields a performance degradation. For Python, among the 5 scripts whose iteration coverage is greater than 30%, 4 showed better speedup than the other scripts: `fannkuch-redux`, `mandelbrot`, `nbody`, and `spectralnorm`.

Low iteration coverage is caused by two factors: unexpected exits due to the limited coverage of path profiling, and value mispredictions, which prevent dispatch into the specialized loop.

Profile coverage may be limited when a path does not occur during training. Since IPLS generates specialized loops according to path profiling, the program may take an

Input Script (Lines of Code)	Iteration Coverage(%)	Meta-level-loops/traces
Lua-5.2.0 (19,832 LOC)		
binary-trees (50)	78.01%	5
fannkuch-redux (48)	74.75%	6
fasta (98)	42.63%	5
k-nucleotide (66)	98.50%	2
mandelbrot (27)	99.99%	4
meteor (223)	0.76%	7
nbody (121)	97.57%	4
pidigits (104)	99.61%	9
regex-dna (46)	2.93%	5
reverse-complement (40)	0.00%	2
spectral-norm (43)	88.90%	5
Perl-5.14.2 (201,786 LOC)		
binary-trees (47)	99.99%	4
fannkuch-redux (55)	99.99%	4
fasta (122)	98.99%	11
k-nucleotide (29)	99.93%	2
mandelbrot (77)	99.90%	4
meteor (235)	99.90%	10
nbody (107)	99.90%	10
pidigits (47)	95.06%	11
regex-dna (49)	99.92%	3
reverse-complement (29)	99.99%	4
spectral-norm (49)	99.99%	4
Python-2.7.2 (314,921 LOC)		
binary-trees (70)	9.36%	2
fannkuch-redux (56)	35.02%	3
fasta (118)	3.03%	4
k-nucleotide (57)	5.15%	1
mandelbrot (55)	52.09%	1
meteor (205)	2.06%	5
nbody (116)	76.48%	6
pidigits (40)	1.15%	3
regex-dna (44)	11.23%	8
reverse-complement (37)	64.83%	10
spectral-norm (56)	48.77%	5

Table 6.3: Execution characteristics of each interpreter and each static input: *Iteration coverage* denotes the fraction of hot loop iterations that are executed in the specialized code. *Meta-level loops/traces* denotes the number of identified patterns.

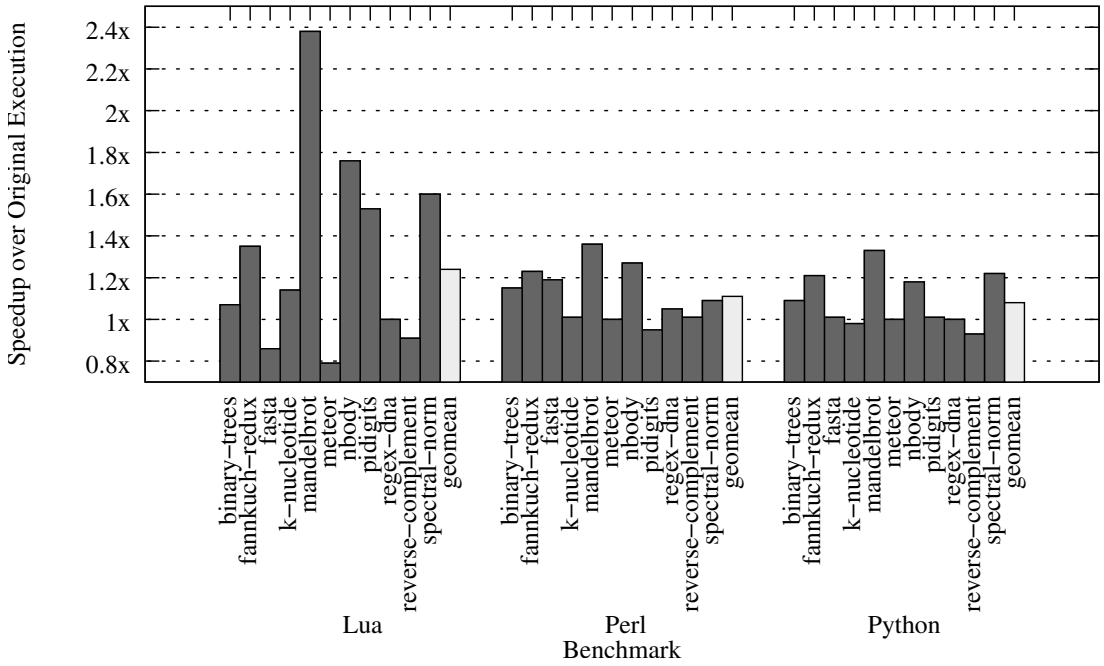


Figure 6.12: Whole-program speedup with three interpreters: Lua, Perl, and Python, and 11 input scripts for each.

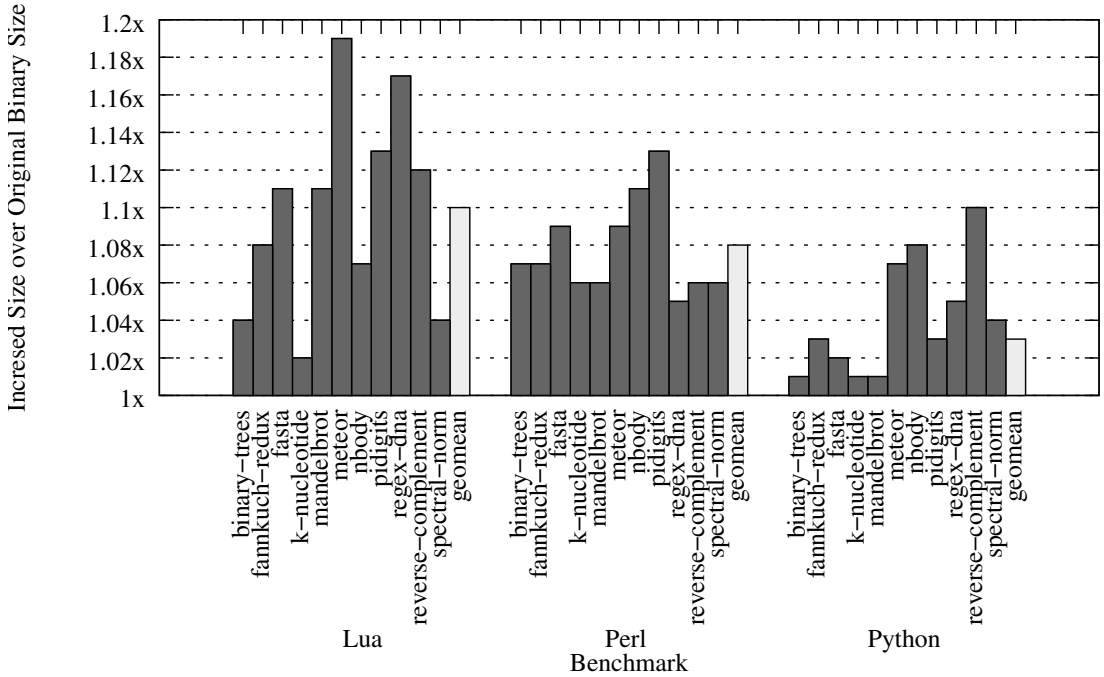


Figure 6.13: Code size increase after specialization for three interpreters: Lua, Perl, and Python, and 11 input scripts for each.

Input script	Lua (%)	Perl (%)	Python (%)
binary-trees	19.46	0.00	20.10
fannkuch-redux	2.13	0.00	3.32
fasta	12.50	0.00	0.99
k-nucleotide	0.00	0.00	46.90
mandelbrot	0.00	0.00	1.10
meteor	4.22	0.00	8.67
nbody	0.00	0.00	0.30
pidigits	0.03	0.00	46.83
regex-dna	9.71	0.00	1.66
reverse-complement	0.57	0.00	0.00
spectral-norm	1.25	0.00	8.33

Table 6.4: Unexpected exits from the specialized loop as a fraction of the number of iterations running in a specialized loop.

unexpected path within an iteration of its hot loop. To guarantee correctness, the code generator inserts tests to detect this case, and conservatively branches to the unspecialized code. We call such exits *unexpected*. Since the dispatch condition may only enter the specialized loop at the beginning of a pattern, if a specialized loop experiences an unexpected exit the remainder of that pattern must execute in non-specialized code before there is an opportunity to re-enter specialized code. The occurrence of unexpected exits among the total number of iterations is shown in Table 6.4.

Additionally, value misprediction may prevent the main loop from dispatching into the specialized loop. This occurs for some input scripts in which control dependences carry information from dynamic input to the dispatch condition. In other words, our optimistic implementation of DIFT occasionally misclassifies a dynamic value as static. As a result, the specialized program may experience a pattern that did not occur during profiling. In such cases, the main loop does not dispatch to the specialized code, decreasing iteration coverage. For example, the hottest loop of `reversecomplement` implemented in Lua includes an `if` statement that is predicated on an input that varies across program invocations. This induces two different control paths in the script and foils IPLS value prediction.

`Fasta` implemented in Lua experiences performance degradation even though the pro-

Input script	Ratio of dynamic instruction counts
binary-trees	1.11
fannkuch-redux	1.13
fasta	1.01
k-nucleotide	1.16
mandelbrot	2.28
meteor	0.99
nbody	1.55
pidigits	1.58
regex-dna	1.00
reverse-complement	1.00
spectral-norm	1.65

Table 6.5: Ratio of dynamic instruction count of the original program to that of the specialized program for Lua-5.2.0. Larger numbers indicate a greater reduction in dynamic instructions.

gram has good iteration coverage. The difference in dynamic instruction counts after specialization, shown in Table 6.5, explains the slowdown. The table shows the ratio of the dynamic instruction count of the original program over the specialized program. Therefore, larger numbers in the table mean that fewer dynamic instructions were executed in the specialized program compared to the original. Unlike other programs with high iteration coverage, the ratio is close to 1 for `Fasta`. This implies that for `Fasta`, specialization was not able to find enough precomputable static instructions to amortize the specialized loop dispatch overhead introduced in the original loop.

Python `reversecomplement` is another case that experiences a slow down despite high iteration coverage. This is because the hottest loop in `reversecomplement` consists of a single meta-level node. Specialization of a singleton meta-level loop has negligible benefit since there are no opportunities for optimization over multiple iterations.

Perl is unique in the sense that it executes more than 99% of its main loop iterations in specialized code for almost all inputs, as shown in Table 6.3. This indicates the effectiveness of value prediction based on detected patterns. This high predictability stems from the unique implementation of the Perl interpreter's intermediate representation. For instance, `reverse-complement` is implemented using Perl's `split` operation instead of a syn-

tactic `if` statement. Since the Perl interpreter implements large operations such as `split` as a single opcode, the control flow in Perl scripts is typically less dynamic.

The Perl interpreter's main loop is structured differently than others. While other interpreters load the opcode, parse it, and branch to the appropriate handler, each handler in the Perl interpreter returns a function pointer that serves as a continuation to the next operation. Hence, the Perl interpreter repeatedly performs indirect calls. This is beneficial for IPLS since there can be no unexpected exits. On the other hand, it limits IPLS since there are fewer opportunities to optimize precomputable instructions.

This suggests that most performance improvements for Perl come from increased instruction-level parallelism exposed by unrolling the loop and by better branch prediction caused by replacing the indirect function calls with a conditional branch and direct function call. The ratio of dynamic instructions before and after specialization for Perl ranges from 0.96 to 1.02, except 0.90 in `pidigits`, which shows no benefit from specialization.

6.5 Limitations

While no modification has been made to the interpreter source code, Perl is configured with the `PURIFY` and `NO_PERL_PRESERVE_IVUV` compile time options. The `PURIFY` flag compiles Perl with 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 the proposed system'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.

Python interpreter (Python-2.7.5, 537,450 LOC) has also been tried to be parallelized with the proposed technique. However, Python-2.7.5 contains various implementation

characteristics that prevent parallelization, even after specialization has been applied. For example, the main loop of the Python interpreter handles asynchronous events (e.g. signal handler invocation) after every `n` Python bytecode executions. This logic introduces extra dependences that are unrelated to the behavior of the input script. Another example is Python-2.7.5's handling of integer multiplication overflow. If overflow occurs, the interpreter may create a new object that holds the correct value and use the new object going forward, which introduces another memory dependence. The additional dependences introduced by the aforementioned characteristics are independent of fixed input behavior and prevent parallelization.

Another limitation comes from the applicability of IPLS. Abstract syntax tree interpreters cannot be parallelized with the proposed technique, because IPLS targets high iteration count loops and assumes the existence of an instruction in the loop that computes values in each iteration forming repeating patterns. While bytecode interpreters satisfy these conditions with their main loops and the instructions representing interpreter's program counter, abstract syntax tree interpreters do not have such structures. In addition, IPLS only looks for patterns of values generated by a single instruction. If an interpreter uses complex structures to represent its program counter which cannot be computed by a single instruction, IPLS will fail to recognize the pattern. Lastly, if the implementation of the interpreter main loop is optimized with `goto` statements, IPLS may not be applicable. IPLS only observes instructions in the loop header to detect repeating patterns, but `goto`s may place the instruction for interpreter's program counter outside the main loop header.

Chapter 7

Related Work

Recent advances in both the fields of program specialization and automatic parallelization have made great strides in increasing program performance. This dissertation proposes the first technique to make use of their natural synergy.

This chapter describes summarizes prior work on program specialization and automatic parallelization. As script interpretation is a key application domain of the proposed technique, techniques about parallelizing script interpretation are described as well.

7.1 Program Specialization

Program specialization techniques can be classified into two categories: *compile-time specialization* that generates specialized programs at compile-time, and *run-time specialization* that performs specialization at run-time. IPLS, a program specialization technique proposed in this dissertation, is a compile-time technique.

7.1.1 Compile-time Specialization

Compile-time specialization requires binding-time information, which classifies all instructions in the target program as either static or dynamic. To obtain binding-time information,

C-Mix [1, 49] and Tempo [18, 19] rely on compile-time analysis and user annotations. Unlike C-Mix and Tempo, IPLS exploits profiling information to obtain the binding-time information, so IPLS is complementary to these previous works.

Berlin et al. [3, 4] propose a specializer that optimizes scientific programs written in high-level languages such as LISP. Since control flows in scientific programs are not affected by input values, they unroll loops to expose parallelism inherent in the underlying numerical computation. However, the loop unrolling may cause code explosion, so they explicitly exclude loops with high iteration counts from unrolling. Although the authors propose a heuristic to stop unrolling beyond a certain threshold, the heuristic is not evaluated. C-Mix [1, 49] and Tempo [18, 19] also suffer from code explosion, and rely on user annotations to avoid the problem. Since IPLS uses pattern based loop unrolling, this work neither has the code explosion problem, nor does it require any user annotation.

JSpec [83] specializes Java using C as an intermediate language and uses Tempo for binding-time analysis. Kleinrubatscher et al. [40] propose a specializer for a subset of FORTRAN using abstract interpretation to gather binding-time information.

Run-time Specialization Run-time specialization [2, 20, 26, 28, 29, 51, 84] has an advantage over compile-time specialization because it can exploit run-time constants that are not available at compile-time. However, run-time specialization suffers from high overhead of dynamic code generation. Tempo [20] supports both compile-time and run-time specialization sharing binding-time analysis together, but run-time specialization of Tempo achieves about 80% of the speedup of compile-time specialization, due to the run-time overheads [58]. It shows that specializing the program statically as much as possible can maximize the potential performance of specialized programs. Exploiting profiling results at compile-time, IPLS avoids the run-time overheads.

DyC [28, 29] is a run-time specializer, primarily focused on reducing run-time overheads from dynamic code generation and optimization. DyC requires user annotations to

direct optimization policy and improve the precision of binding time information given by compile-time analysis. Although Calpa [55] automatically generates the annotations for DyC with profile information, it is limited to annotations about optimization policy only. Therefore, without hints from programmers, Calpa's final result is still limited by compile-time analysis. Unlike DyC, IPLS is a fully-automatic specializer that does not require any user annotation.

Bala et al. [2] and Shankar et al. [84] propose run-time specializers that, like IPLS, do not require any user annotation. The specializers automatically find and optimize frequently executed traces by exploiting the information available at run-time only. Since they detects hot values to find traces, multiple traces in hot code regions can be generated increasing dispatching overheads. However, IPLS detect patterns before specializing codes, so IPLS can reduce dispatching overheads. In addition, while Shankar et al. [84] rely on strong type systems of Java to optimize program with possible heap constants, IPLS can specialize programs written in C without type system supports.

Bolz et al. and Yermolovich et al. propose Just-In-Time compilers which are optimized to the specific requirements of extracting performance benefits from script interpreters [7, 103]. Script interpreters cannot benefit from tracing-JIT techniques [26, 106] which do not trace across multiple iterations. That is because each iteration in main loops of interpreters has diverse control flow due to different instructions in the script. Addressing the problem, these compilers find frequently executed traces in the scripts that stretch over multiple iterations of the interpreter main loops, and specialize the interpreters for the traces. This approach is similar to IPLS, but the JIT compilers require user annotations in the interpreter program at branch instruction handlers to find boundaries of loops in the scripts, and at data structures to find static values.

7.2 Automatic Parallelization

There has been a large body of work on automatic parallelization. Most of the early works concentrated on parallelizing programs manipulating regular, analyzable data structures like array-based scientific applications [6, 12, 13, 22, 27, 52, 76, 77, 82, 100, 102]. However, recent approaches target general purpose programs with complex control-flow and irregular data structures. Some techniques [11, 62, 75] rely solely on static program analysis to identify the applicability of parallel transformations. These techniques are able to achieve considerable speedups for programs in which precise dependence information can be obtained at compile-time. For example, HELIX [11] achieves an average of 2.25× speedup for thirteen benchmarks from SPEC CPU2000 [86] on a commodity six-core machine.

Still, imprecision and fragility of static analyses limits the applicability of automatic parallelization [37]. Several techniques have been proposed to overcome this limitation by resorting to programmer's help [9, 41, 67, 92, 95, 98, 105]. Some of the techniques [9, 41, 67] ask programmers to add annotations to the program to relax serializing constraints that inhibit parallelization. Paralax [98] requires users to give hints on dependence analysis, while Tournavitis et al. [95] and Yu et al. [105] run dependence profiling and ask users to verify the profiling result. Theis et al. [92] proposed annotations that directly specify the boundaries of partition for pipelined parallelism. 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 compiler.

Speculation alleviates the limitations of static analysis without manual intervention. Many automatic parallelization systems using speculation [8, 25, 30, 31, 34, 35, 37, 38, 45, 53, 72, 87, 93, 94, 97, 99, 107, 108] have been proposed, but most of them [8, 25, 30, 31, 35, 45, 72, 87, 97, 99, 107, 108] require specialized hardware to support speculative parallel execution. SUDS [25] is an automatic parallelization system targeting Raw microprocessors [91]. Steffan et al. [87] proposed a system where compiler and TLS (Thread-Level

Speculation) supporting architecture work together to achieve efficient speculative parallelization. However, the work is more focused on designing scalable TLS hardware than parallelizing compiler. Mitosis compiler [72] partitions the target program into speculative threads under the assumption of hardware support for speculation. Mitosis compiler generates pre-computation slices to predict live-in values for each thread, which reduces the overhead from inter-thread data dependences. POSH [30, 45] is another parallelizing compiler targeting architectures with speculation support. POSH exploits program structures (e.g. subroutines, loops) to simplify the generation of concurrent tasks, and uses profiling to ignore non-profitable tasks. Johnson et al. [35] proposed a more elaborate profiling algorithm to discover the most profitable parts of the program to perform speculative parallel execution. Speculative DSWP presented by Vachharajani et al. [97] enables applying speculation to pipeline parallelization. However, a hardware versioned memory system is required to run the parallelized program. Zhong et al. [108] proposed code transformations to make loops in the target program amenable to speculative DOALL parallelization, assuming the availability of TLS hardware. Wang et al. [99] and Hertzberg and Olukotun [31] showed a potential of exploiting hardware support to enable run-time speculative parallelization for legacy binaries.

Another class of speculative parallelization methods [34, 37, 38, 53, 93, 94] does not require any hardware extensions and is applicable to commodity machines. In the CorD execution model [93, 94], 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. The main thread is responsible for in-order commit and misspeculation detection as well. Mehrara et al. [53] proposed a system based on the code generation framework presented by Zhong et al. [108]. To support commodity machines, the system uses a light-weight software transactional memory instead of TLS hardware. Privateer [34] is the first fully automatic system that supports speculative reduction and privatization. The system rely on software-based run-time system to validate speculative assumptions. Cluster

Spec-DOALL [38] and ASAP [37] proposed an automatic parallelizing compiler and run-time system to enable speculative parallel execution on commodity clusters. While Cluster Spec-DOALL only supports DOALL parallelization, ASAP supports pipelined parallelization as well. Unlike the system proposed in this dissertation, none of these techniques are capable of context-sensitive speculation. Supporting context-sensitivity is critical to parallelizing complex loops that contain many function calls, as a significant number of non-manifesting dependences can only be identified using context-sensitive information. The context-sensitive speculation developed in this dissertation played a vital role in achieving speedup for the evaluated programs.

Several works focus on proposing software run-time systems to support speculative parallelization. LRPD [78] and R-LRPD [22] supports speculative DOALL execution by validating the absence of loop-carried dependences at the end of parallel execution. For LRPD, if speculation fails then the entire loop needs to be re-executed sequentially. R-LRPD addresses this problem by adapting sliding window strategy. If N processors are available, R-LRPD distributes only first N iterations to the processors instead of distributing the entire iteration space, then validates speculation at the end of the parallel execution of N iterations. This strategy avoids re-execution of successful iterations, but parallel performance is penalized by frequent interruption of sequential validation. Cintra and Llanos [14] proposed a run-time system that performs *eager* memory management that each speculative operation checks if misspeculation occurred and updates the non-speculative memory directly when there is no misspeculation. To support this, an expensive memory fence operation is required for each speculative memory operation. These techniques are only applicable to array-based scientific programs, as the techniques cannot address the memory access using pointers.

Oancea et al. [59] proposed SpLIP to reduce various memory and performance overheads of prior software-based run-time systems (e.g. overhead to buffer the speculative state, overhead of executing synchronization operations for every speculative memory ac-

cess). However, as SpLIP still performs expensive hash operations for each speculated load and store, it has high performance overheads when applied to aggressively speculated programs. Yiapanis et al. [104] proposed two software run-time systems, MiniTLS and Lector. MiniTLS relies on eager memory management where each speculative operation updates the non-speculative memory directly, while Lector uses a lazy approach that buffers the speculative state and commits them later. However, both MiniTLS and Lector requires every speculative memory operation to acquire a lock, which results in high execution overhead. STMLite [53] is based on software transactional memory (STM) [23, 81, 54, 85]. It optimizes performance by providing enough functionality to support speculative loop parallelization without implementing the whole spectrum of transactional memory features. STMLite, however, relies on a centralized commit unit, which prevents this technique from scaling to large numbers of threads. None of these techniques support multi-threaded atomicity, which is necessary to enable speculative pipeline parallelization.

Raman et al. resolve this problem by developing SMTX [74], a software run-time system that allows multi-threaded atomicity, as described in Chapter 2.3. Kim et al. proposed DSMTX [39], a SMTX implementation targeting clusters. The optimized run-time system proposed in this dissertation is based on SMTX.

7.3 Parallelizing Script Interpretation

There are a variety of parallel libraries or parallel language extensions [15, 57, 46, 69, 70, 79, 80] 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 scripts. These techniques are focused on parallelizing the R programming language [73], because of R's popular use in scientific computing and machine learning com-

munity. In [48], 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. ALCHEMY [63] is another platform that supports the automatic parallelization of R programs. Talbot et al. present Riposte [90], 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.Though these techniques are able to successfully parallelize some scripts, they require manual changes or extensions to the interpreting environment. 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.

Chapter 8

Conclusion and Future Directions

This dissertation proposes techniques to exploit input parallelism in a fully automatic fashion. Evaluation of these techniques with a prototype implementation demonstrates the potential of research on automatic extraction of input parallelism. This chapter concludes the dissertation and describes potential future avenues to make further advances in the field.

8.1 Conclusion

As multi-core processors become the norm in all-levels of computing, extracting thread-level parallelism from the application is necessary to achieve performance improvement. However, writing multi-threaded program is much harder than sequential program, which makes automatic parallelization of sequential program an attractive alternative to harness multiple cores. Despite the recent advances in automatic parallelization systems, existing techniques cannot exploit parallelism in program inputs. This prevents the application of automatic parallelization to some key application domains, including script interpretation.

This dissertation proposes a fully automatic technique to exploit the parallelism within fixed program inputs. By coupling program specialization with speculative parallelization techniques, input parallelism can result in parallel speedup. First, the dissertation presents IPLS, a program specialization technique that harnesses the repeating pattern induced by

program invariants including fixed program inputs. IPLS materializes parallelism within the program invariants into the specialized program, thereby functioning as an enabling transformation for automatic parallelization. Second, the dissertation proposes context-sensitive speculation to improve the applicability of automatic parallelization. Context-sensitivity plays a critical role in extracting parallelism from programs that hard to reason about, such as IPLS specialized programs. Lastly, the dissertation proposes optimization to the run-time system that supports speculative parallelization. Proposed optimizations greatly reduce the performance overhead of the run-time system and enable scalable speedup even with aggressively speculated programs.

The prototype implementation of the proposed technique has been evaluated against two widely-used open-source script interpreters with 6 input scripts, which describe parallel algorithms, each yielding a geomean speedup of 5.10× over the best sequential version. This proves that the automatic exploitation of input parallelism is both feasible and worthy of further investigation.

8.2 Future Research Directions

- Handling dependences resulting from fixed-input-independent program behavior: As discussed in Chapter 6.5, dependences between instructions caused by program behavior independent of fixed program inputs are primary obstacles to the parallelization of specialized programs. Inventing program transformation techniques that cleanly separate parts of the program that depend upon fixed program inputs from the other parts will be one way to alleviate this problem. The DIFT-based profiling technique proposed in this dissertation, which distinguishes instructions dependent solely upon program invariants from other instructions, can be extended to support such transformations.

- Exploiting complex forms of input parallelism: Evaluation shows that the specializa-

tion process introduces dependences that change DOALL parallelism in input scripts into pipeline parallelism in the specialized interpreter program. If the input script contains complex forms of parallelism, e.g., pipeline parallelism, more non-trivial dependences manifest. Techniques proposed in this dissertation can be extended to resolve such dependences to enable extraction of complex forms of input parallelism.

- Applying the techniques to the domain outside of script interpretation: Techniques proposed in this dissertation are only evaluated against script interpreters and their input scripts. Though script interpretation is the application domain that motivates this research, the techniques can be applied to other applications as well.

  Scientific computations such as numerical simulations, computational optimizations, and signal processing systems represent another application domain that may have input parallelism and can benefit from the technique proposed in this dissertation. Programs in this domain are implemented in highly parameterized fashion to maintain generality [5], and specific values of each parameter are given as program input. For many cases, some parameters have fixed values across multiple executions of the program. Prior work [5, 83] showed that specializing scientific programs against fixed values of parameters simplifies data- and control-flow of the program. It is possible that simplification of data- and control-flow removes dependences between instructions and unlocks parallelism. For such a case, automatic parallelization can be enabled by applying the techniques presented in this dissertation. As programs in scientific computation domain are computation intensive, a substantial speedup is expected when they are parallelized.

Bibliography

[1] L. O. Andersen. Program analysis and specialization for the C programming language, May 1994.

[2] 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*, pages 1–12, June 2000.

[3] A. Berlin. Partial evaluation applied to numerical computation. In *LISP and Functional Programming '90*, 1990.

[4] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23, December 1990.

[5] A. A. Berlin and R. J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In *PEPM'94 - ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Walt Disney World Vilage, Orlando, Florida, USA, 25 June 1994, Proceedings. Technical Report 94/9*, pages 133–141, 1994.

[6] 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 *Proceedings of the workshop on Languages and Compilers for Parallel Computing*, 1994.

[7] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09. ACM, 2009.

[8] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[9] M. J. Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[10] M. J. Bridges, N. Vachharajani, Y. Zhang, T. B. Jablin, and D. I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, January 2008.

[11] 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 10th International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93. ACM, 2012.

[12] D.-K. Chen and P.-C. Yew. On effective execution of nonuniform doacross loops. *IEEE Trans. Parallel Distrib. Syst.*, 7(5):463–476, May 1996.

[13] D.-K. Chen and P.-C. Yew. Redundant synchronization elimination for doacross loops. *IEEE Transactions on Parallel and Distributed Systems*, 10(5):459–470, 1999.

[14] M. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):562–576, June 2005.

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

[16] Computer Language Benchmarks Game. http://shootout.alioth.debian.org/.

[17] D. A. Connors. Memory profiling for directing data speculative optimizations and scheduling. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[18] 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 Internaltional Seminar on Partial Evaluation*, pages 54–72, London, UK, 1996. Springer-Verlag.

[19] C. Consel, J. L. Lawall, and A.-F. Le Meur. A tour of tempo: a program specializer for the c language. *Sci. Comput. Program.*, 52:341–370, August 2004.

[20] C. Consel and F. Noel. A general approach for run-time specialization and its application to c. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[21] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–884, August 1986.

[22] 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*, pages 20–29, 2002.

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

[24] 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*, 9:319–349, July 1987.

[25] M. I. Frank. *SUDS: Automatic Parallelization for Raw Processors*. PhD thesis, MIT, 2003.

[26] 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 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478, 2009.

[27] M. Girkar and C. D. Polychronopoulos. Extracting task-level parallelism. *ACM Trans. Program. Lang. Syst.*, 17(4):600–634, July 1995.

[28] 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)*, pages 163–178, June 1997.

[29] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 293–304, May 1999.

[30] L. Han, W. Liu, and J. M. Tuck. Sspp:1772954.177297opeculative parallelization of partial reduction variables. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 141–150, New York, NY, USA, 2010. ACM.

[31] B. Hertzberg and K. Olukotun. Runtime automatic speculative parallelization. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, april 2011.

[32] T. B. Jablin. *Automatic Parallelization for GPUs*. PhD thesis, 2013.

[33] B. J. Jain and K. Obermayer. Extending bron kerbosch for solving the maximum weight clique problem. *CoRR*, abs/1101.1266, 2011.

[34] 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.

[35] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 205–214, New York, NY, USA, 2007. ACM.

[36] A. Kejariwal, A. Veidenbaum, A. Nicolau, X. Tian, M. Girkar, H. Saito, and U. Banerjee. Comparative architectural characterization of spec cpu2000 and cpu2006 benchmarks on the intel® core™ 2 duo processor. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on*, pages 132–141, July 2008.

[37] H. Kim. *ASAP: Automatic Speculative Acyclic Parallelization for Clusters*. PhD thesis, 2013.

[38] 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.

[39] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. Scalable speculative parallelization on commodity clusters. In *In Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010.

[40] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. *SIGPLAN Not.*, 30:61–70, April 1995.

[41] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[42] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, 4:812–826, July 1993.

[43] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[44] S.-W. Liao, A. Diwan, R. P. B. Jr., A. M. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Principles Practice of Parallel Programming*, pages 37–48, 1999.

[45] 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*, pages 158–167, 2006.

[46] 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*, PLDI '14. ACM, 2014.

[47] Lua. http://www.lua.org/.

[48] 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.

[49] H. Makholm. Specializing c - an introduction to the principles behind c-mix/ii. Technical report, University of Copenhagen, Department of Computer Science, 1999.

[50] T. R. Mason. Lampview: A loop-aware toolset for facilitating parallelization. Master's thesis, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, United States, August 2009.

[51] H. Masuhara and A. Yonezawa. Run-time bytecode specialization. In *Proceedings of the Second Symposium on Programs as Data Objects*, PADO '01, pages 138–154, London, UK, 2001. Springer-Verlag.

[52] K. S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, pages 54–63, New York, NY, USA, 1994. ACM.

[53] 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*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[54] 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.

[55] 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*, pages 100–109, November 1999.

[56] G. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82 –85, Jan 1998.

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

[58] F. Noel, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 132–, Washington, DC, USA, 1998. IEEE Computer Society.

[59] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 223–232, New York, NY, USA, 2009. ACM.

[60] 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*, ASPLOS '13. ACM, 2013.

[61] G. Ottoni. *Global Instruction Scheduling for Multi-Threaded Architectures*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, September 2008.

[62] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO '05: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[63] F. Padberg and M. Mirold. An experimentation platform for the automatic parallelization of r programs. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, Dec 2012.

[64] A. K. Peng Wu and C. Cascaval. Compiler-driven dependence profiling to guide program parallelization. In *LCPC*, pages 232–248, 2008.

[65] Perl. http://www.perl.org/.

[66] L.-N. Pouchet. PolyBench: The Polyhedral Benchmark suite. http://www-roc.inria.fr/ pouchet/software/polybench/download.

[67] 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 2011 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, 2011. ACM.

[68] 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.

[69] PyMPI. http://pympi.sourceforge.net/.

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

[71] Python. http://www.python.org/.

[72] 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 Pro-*

gramming language design and implementation, pages 269–279, New York, NY, USA, 2005. ACM.

[73] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004. 3-900051-07-0.

[74] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS '10: Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.

[75] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, April 2008.

[76] L. Rauchwerger, N. M. Amato, and D. A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming (IJPP)*, 26:537–576, 1995.

[77] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995.

[78] 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.

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

[80] 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*, HotPar'12. USENIX Association, 2012.

[81] 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*, MICRO 39. IEEE Computer Society, 2006.

[82] J. Saltz, R. Mirchandaney, and R. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40, 1991.

[83] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25:452–499, July 2003.

[84] 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.

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

[86] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[87] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, February 2005.

[88] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 1–12, June 2000.

[89] E. Suh, J. W. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[90] 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*, PACT '12. ACM, 2012.

[91] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuit and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[92] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[93] 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*, PLDI '10. ACM, 2010.

[94] 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*, MICRO 41. IEEE Computer Society, 2008.

[95] 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*, PLDI '09, 2009.

[96] N. Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[97] 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*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[98] 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 architectures and compilation techniques*, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.

[99] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T. fook Ngai, and J. Fang. Dynamic parallelization of single-threaded binary programs using speculative slicing. In *ICS'09*, 2009.

[100] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. The suif compiler system: A parallelizing and optimizing research compiler. Technical report, Stanford, CA, USA, 1994.

[101] Q. Wu, A. Pyatakov, A. N. Spiridonov, E. Raman, D. W. Clark, and D. I. August. Exposing memory access regularities using object-relative memory profiling. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2004.

[102] C.-Z. Xu and V. Chaudhary. Time stamp algorithms for runtime parallelization of doacross loops with dynamic dependences. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):433–450, May 2001.

[103] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09. ACM, 2009.

[104] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Luján. Optimizing software runtime systems for speculative parallelization. *ACM Trans. Archit. Code Optim.*, 9(4):39:1–39:27, Jan. 2013.

[105] 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*, PLDI '13. ACM, 2013.

[106] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: a graduallY extensible trace interpreter. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 83–93, 2007.

[107] A. Zhai, J. G. Steffan, C. B. Colohan, and T. C. Mowry. Compiler and hardware support for reducing the synchronization of speculative threads. *ACM Transactions on Architecture and Code Optimization*, 5(1):1–33, 2008.

[108] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA '08: Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.