







# Approximate Data Dependence Profiling Based on Abstract Interval and Congruent Domains

Mostafa Abbas<sup>1</sup>(✉) , Rasha Omar<sup>1</sup>(✉) , Ahmed El-Mahdy<sup>1,3</sup>(✉) ,  
and Erven Rohou<sup>2</sup>(✉) 

<sup>1</sup> Egypt-Japan University of Science and Technology (E-JUST), Alexandria, Egypt  
{mostafa.abbas,rasha.omar,ahmed.elmahdy}@ejust.edu.eg

<sup>2</sup> Univ Rennes, Inria, CNRS, IRISA, Rennes, France  
erven.rohou@inria.fr

<sup>3</sup> Alexandria University, Alexandria, Egypt

**Abstract.** Although parallel processing is mainstream, existing programs are often serial, and usually re-engineering cost is high. Data dependence profiling allows for automatically assessing parallelisation potential; Yet, data dependence profiling is notoriously slow and requires large memory, as it generally requires keeping track of each memory access. This paper considers employing a simple abstract single-trace analysis method using simple interval and congruent modulo domains to track dependencies at lower time and memory costs. The method gathers and abstracts the set of all memory reference addresses for each static memory access instruction. This method removes the need for keeping a large shadow memory and only requires a single pair-wise analysis pass to detect dependencies among memory instructions through simple intersection operations. Moreover, the combination of interval and congruent domains improves precision when compared with only using an interval domain representation, mainly when the data is not accessed in a dense access pattern. We further improve precision through partitioning memory space into blocks, where references in each block abstracted independently. An initial performance study is conducted on SPEC CPU-2006 benchmark programs and polyhedral benchmark suite. Results show that the method reduces execution time overhead by 1.4× for polyhedral and 10.7× for SPEC2006 on average; and significantly reduces memory by 109780× and 6981× for polyhedral and SPEC2006 respectively; the method has an average precision of 99.05% and 61.37% for polyhedral and SPEC respectively. Using memory partitioning resulted in improving mean precision to be 82.25% and decreasing memory reduction to be 47× for SPEC2006 suite.

**Keywords:** Data dependence profiling · Dynamic binary analysis · Congruent domains · Interval domains

---

A. El-Mahdy—On-leave from Alexandria University.

© Springer Nature Switzerland AG 2020  
A. Brinkmann et al. (Eds.): ARCS 2020, LNCS 12155, pp. 3–16, 2020.  
[https://doi.org/10.1007/978-3-030-52794-5\\_1](https://doi.org/10.1007/978-3-030-52794-5_1)

# 1 Introduction

Data dependence profiling is an essential step towards deciding on parallelising loops, especially for heterogeneous parallel platforms, where the cost of re-engineering originally serial programs is high. However, obtaining the profiles is generally a costly operation as it requires instrumenting all memory access instructions; hence it suffers from significant memory consumption and runtime overhead (i.e., separate records for every accessed memory address). Therefore, profilers rely on approximation methods, trading-off accuracy with analysis overhead. One typical method is sampling the execution trace, where only a portion of the trace is analysed [5, 23]. However, sampling is prone to missing some dependence arcs and losing the most recent relationships. Other profilers tackle the runtime overhead by parallelising runtime analysis [10, 17, 24].

This paper considers performing a form of abstract analysis for all memory references over the whole program. The analysis borrows from the well-known abstract interpretation static analysis method, which is used to generate an abstract collective program state for all possible execution traces [1, 6, 18]. Here, we specialise the method to analyse only one execution trace (profiling trace). Moreover, we do not perform abstract computations or interpretations over the abstract state; instead, we rely on the underlying execution system to generate the current state and use a corresponding abstract operation (Union) to gather and approximate the state. In other words, the main aim is to generate an abstract single-trace semantics, from the current execution trace.

The advantage of this approach is that we dramatically reduce the memory size required for the analysis, as there is no need to have a shadow memory, which is typical in profilers. Moreover, we only perform pair-wise operations on the static memory access instructions after gathering their trace state, to identify data-dependence, thus also significantly reducing profiling time. The trade-off here is precision or false dependencies, while the sensitivity or true dependencies are never missed.

In this paper, for each dynamic memory access operation, we use a corresponding abstract operation that joins the current memory access address with the current abstracted set of seen addresses for the corresponding memory access instruction, thus generating an abstract single-trace semantic for all static memory instructions. The abstraction uses a composite abstract domain, consisting of interval and congruent domains. The former provides an approximation of the covered range, and the latter provides information about the access pattern. We partition the memory address space into blocks and associate a value from the composite domain for each block, to further improve precision.

This paper has the following contributions:

- Utilise abstract interval and congruent domains to approximate the accessed memory location for each static memory access instruction.
- Conduct an initial performance investigation using polyhedral benchmark and SPEC CPU-2006 benchmark suites using the Pin system [14].

- Utilise the gathered semantics to detect pair-wise memory data dependencies (RAW, WAR, and WAW) from binary files at runtime for all static memory access instructions.

This paper is organised as follows: Sect. 2 discusses related work. Section 3 provides an overview of abstract interpretation and defines the interval and congruent domains and their corresponding operations. Section 4 discusses our profiling algorithm. Section 5 presents our initial results. Finally, Sect. 6 concludes our paper and discusses future work.

## 2 Related Work

### 2.1 Static Analysis

Abstract interpretation is a static analysis method that generates a collective program semantics at each program point. The method relies on generating abstract equations that generate the collective program semantics at each program point. Abstract interpretation is not generally used in the context of data dependence analysis for imperative programs (functional programs have in-depth treatment, e.g., [3]); nevertheless, there are attempts in this direction [8, 16].

Ricci [20] is an early attempt to consider dependence analysis by combining interval and bisection domains; the latter is used to maintain relations among pairs of variables. The technique is applied at the source level on some kernels. Moreover, the use of the bisection domain helped to reduce the interval width, whereas in our approach we consider the density of the elements in the interval, as well conducting the analysis on the binary level; however, we do not generate collective semantics for all possible traces; we only consider the profiling trace, which is typical for profilers.

The use of the congruence domain in abstract interpretation is described in detail in Bydge’s master thesis [4]. This paper adopts the same definitions and operations for that domain. However, the thesis does not consider program parallelisation.

Tzolovski [22] has discussed possible data dependence abstractions, which included direction and distance vectors as defined by Maydan et al. [15]. In this paper, we explore an initial practical implementation of simple dependence detection and leave more elaborate possible abstractions for future work.

SecondWrite [11] is a static binary rewriter tool. It analyses the dependence in regular loops in binary programs and work out to rewrite a parallelised binary version. SecondWrite translates the *x86* binary input to the intermediate format of the LLVM Compiler and then uses the *x86* back-end LLVM to write the output binary. They make use of the LLVM IR rich infrastructure, such as control-flow analysis, data-flow analysis, and optimisation passes to generate a parallel alternative version according to which loop can be partitioned.

## 2.2 Dynamic Analysis

Li et al. [13] introduce a profiler for the sequential and parallel program based on LLVM. They tackle the runtime overhead by parallelising the profiler. The profiler records memory accesses using signatures to achieve efficiency in space, (as an approximate representation, concept from transactional memory [21]) rather than instrument every memory access. The serial profiler has a  $190\times$  slowdown on average for NAS benchmarks and a  $191\times$  slowdown on average for Starbench programs and consumed memory up to 7.856 GB. By using lock-free parallel design, the runtime overhead reduces to  $78\times$  for NAS and  $93\times$  for Starbench. By using a signature with 100 million slots, the memory consumption reduces to 649 MB (NAS) and 1390 MB (Starbench), with accuracy less than 0.4% false-positive rate and less than 0.1% false-negative rate.

Chen et al. [5] implement a data dependence profiling tool on top of the Intel’s Open Research Compiler (ORC) to provide information about the dynamic behaviour of data dependence in programs, mainly for nested loops.

They study two approximation methods: shadow memory and sampling techniques, as a trade-off to mitigate both space and runtime overhead. The tool speculatively ignores dependence edges (between the source and the sink) that have low probability. The threshold value of this low probability can be determined by the overhead of a data mis-speculation process according to the target machine. The slowdown ranges from 16% to 167% on SPEC CPU2000 benchmarks compared to the original execution time by using a sampling rate of 0.0001 to 0.1, respectively, and with a precision ranging from 30% to 10% in missing dependence edges.

Vanka et al. [23] implement a set-based profiling approach coupled with software signatures. The key insight is that set-level tracking provides a better trade-off between accuracy and performance. At compile time, they identify the essential dependence relationships according to a specific optimisation (i.e. speculative code motion) for profiling at runtime. The profiling analysis working on sets, figuring out set’s relationships dependence, rather than working with pair-wise dependence relationships. The set-based profiler is implemented as an IR level pass in LLVM and applied to SPEC2000 benchmarks for presenting results. They achieved a slowdown  $2.97\times$  with the accuracy range from 0.15 and 0.17, measured by normalised average Euclidean distance.

Norouzi et al. [19] implement an extension of DiscoPoP data dependence profiler that uses a hybrid (static and dynamic) approach in reporting the existence of data dependence in the polyhedral loops. The static dependence analysis part excludes the detected dependent memory access instructions in the annotated area by PLUTO (an auto-parallelising compiler for polyhedral loops) and hence excludes them in the dynamic analysis. Finally, static and dynamic dependencies are merged in an appropriate way to be used later in suitable parallelisation discovery tools. It is clear that if no polyhedral loops are detected in a program, the hybrid approach turns into purely dynamic. By conducting experiments on Polybench and NAS Parallel Benchmarks suits, they achieved a median profiling-time reduction by 62% compared to DiscoPoP profiler framework.

Li et al. [12] introduce a profiling method based on repeatedly skipping memory operations in loops. They used DiscoPoP to implement the profiler. The experiments on NAS Parallel Benchmarks and Starbench parallel benchmark suite show a reduction on runtime overhead by 42.5% on average. The reduction runtime mainly comes from the data dependence building phase, where updating the shadow memory remains as the traditional way. The effect of this approach results from the existence of sequencing memory access patterns in loops (e.g. arrays), if not exists, the profiling suffers from extra runtime overhead due to the extra conditions compared to traditional one.

### 3 Proposed Method Formulation

#### 3.1 Gathering Single-Trace Semantics Dynamically

Our method differs from typical abstract interpretation in that we obtain the abstract collective semantics for a single-trace of the program, instead of all traces. Also, we rely on abstracting the collective semantics for only memory read and write operations, not all program points. Moreover, instead of generating abstract equations, and statically solve or interpret them, we rely on obtaining actual read and write addresses values at runtime from the underlying execution environment, and collect the obtained concrete values into collected abstract semantics (i.e. all possible referenced address seen for each memory instruction). In other words, we define a corresponding operation that collects the semantics. Thus, for a memory access operation  $i$  reading or writing from/to address  $a$ , we define the abstract collective semantics,  $\Sigma$  at this operation as:

$$\Sigma_i = \Sigma_i \cup \alpha(\{a\}) \quad (1)$$

Where  $\Sigma \in \mathbb{D}_A$ ; and  $\mathbb{D}_A$  is an abstract domain with partial lattice with ordering relation  $\cup$ , moreover,  $\alpha()$  is an abstraction function that abstracts the current concrete partial collective semantic set into an abstract one. We also adopt a composite abstract domain that consists of interval and congruence domains. The congruent domain is helpful to represent stride memory access patterns, and the interval domain considers the lower and upper limits of memory access address. It is worth mentioning that abstract interpretation guarantees sound analysis, where the obtained collective semantics is always a superset of the concrete collective semantics.

#### 3.2 The Interval Domain

The interval domain [7] is an abstract domain that can determine safe lower and upper limits of program variables. The abstract interval domain,  $\mathbb{D}_I$ , is defined as:

$$\mathbb{D}_I = \{[a, b]\}, \forall a \leq b \in \mathbb{Z} \quad (2)$$

The  $\cup$  and  $\cap$  operators are defined as:

$$[a, b] \cup [a', b'] = [\min(a, a'), \max(b, b')] \quad (3)$$

$$[a, b] \cap [a', b'] = \begin{cases} [\max(a, a'), \min(b, b')] & \text{if } \max(a, a') \leq \min(b, b') \\ \phi & \text{otherwise} \end{cases} \quad (4)$$

### 3.3 The Congruence Domain

The congruence domain [9] consists of abstract values denoted,  $aZ + b$ , Where  $b \in \mathbb{Z}$  and  $a \in \mathbb{N}$ . We will call  $a$  the modulo and  $b$  the remainder. A congruence relation  $(c, b)$  is defined as  $c \equiv b \pmod{a}$ . The set of all  $C$  such that  $c \in C$  and  $c \equiv b \pmod{a}$ , is  $C = \{aZ + b : \forall Z \in \mathbb{Z}\}$ .

Thus, we define the abstract congruent,  $\mathbb{D}_{CG}$ , domain as:

$$\mathbb{D}_{CG} = \{aZ + b\}, \forall a, b \in \mathbb{N} \quad (5)$$

The  $\cup$  and  $\cap$  operators are defined as:

$$(aZ + b) \cup (a'Z + b') = \gcd\{a, a', |b - b'|\}\mathbb{Z} + \min\{b, b'\} \quad (6)$$

$$(aZ + b) \cap (a'Z + b') = \begin{cases} \text{lcm}\{a, a'\}\mathbb{Z} + b'' & \text{if } b \equiv b' \pmod{\gcd\{a, a'\}} \\ \phi & \text{otherwise} \end{cases} \quad (7)$$

Where  $b'' \equiv b \pmod{a}$  and also  $b'' \equiv b' \pmod{a'}$ .

## 4 Profiling Framework

### 4.1 Pin Framework

Pin [14] is a framework for dynamic binary instrumentation framework. Similar to other frameworks, users can observe the running code, detect intensive functions and loops, monitor parameters, and modify the code while it runs. Pin framework provides an API to let users build custom tools called Pintools, which in turn dynamically instruments the compiled binary files in the user space application. By inserting an appropriate runtime analysis routine for a kind of instructions, we can understand the behaviour of a given binary program.

Our profiler inserts instrumentation code dynamically into the binary code for each memory read/write operation; which is mainly callbacks to the corresponding runtime analysis routines. The profiler can operate on three different modes. The first one performs conventional profiling, where the second one performs a comprehensive (i.e. pairwise method) profiling. Both the first and second modes can be considered as a different perspective of ground truth for the underlying data dependence, as it provides an exhaustive, accurate data dependence

results. However, it suffers from immense memory and runtime overhead. The third operation mode performs the proposed profiling technique. Our implementation focuses only on memory references, where data dependence between registers can be easily detected by convenient static analysis.

## 4.2 Conventional Profiling Technique

For each executing memory instruction, the runtime analysis records the effective memory address and corresponding instruction address as a key, value, and mode (read/write) tuple in a hash table. The hash table, thus, keeps track of the last instruction accessed that memory address.

The analysis routine can then construct a corresponding dependence arc at runtime when a memory write operation happens, marking out a dependence relation (i.e. RAW, WAR, WAW) between the current and the last instruction that accessed the same memory location, i.e. a dependence relation between the current memory instruction and the closest prior instruction(s) which depends on. This method is close to the baseline algorithms of previous work (e.g. [12]).

## 4.3 Comprehensive Profiling Technique

Conventional data dependence profiling aims to capture dependencies among executing instructions; i.e. memory references, not static instructions. Another approach, as defined by Bernstein Conditions [2], is to compare the set of all accessed memory locations for each instruction; a depending pair would have a non-empty intersection.

This analysis requires capturing the set of all data references for each static memory instruction, which results in great storage, and complexity of  $O(n^2)$  intersections, where  $n$  is the stored memory address. This analysis is close to the baseline algorithms of previous work [5, 10]. Our proposed method (described below) essentially abstracts this method, significantly reducing the storage requirements.

## 4.4 Abstract Profiling Technique

The third profiling operating mode is the proposed abstract approximate profiling. As in the previously mentioned profiling techniques, the profiler inserts a callback to the runtime analysis routine for each memory access instruction.

The input to the proposed algorithm is effectively a memory access trace. The trace can be generated dynamically by an underlying execution environment or read from an off-line trace file, that has been collected before. The trace is defined as the sequence of the tuples (inst, mode, address), where ‘inst’ refers to the memory instruction address, ‘mode’ refers to whether the instruction is read or write, and finally ‘address’ is the effective memory address accessed by the instruction.

At runtime, the analysis routine manipulates memory addresses by converting them into corresponding abstract interval and congruent domain values,

and accumulating them to the recorded (abstract) addresses, according to Eqs. 3 (referred to as  $\alpha_{\text{Interval}}$ ) and 6 (referred to as  $\alpha_{\text{Congruent}}$ ) as mentioned above. Here, each static memory instruction has its hash table entry for collecting abstract profiling data. The substantial difference from conventional profiling is that the profiler stores only the abstract set of seen memory addresses, for each memory instruction (i.e. limited number of entries related to the number of static memory instructions). Finally, we compute the intersection between those intervals/congruence values of memory instructions (read or write) according to the aforementioned Eqs. 4 and 7, indicating the potential of the existence of dependence. The intersection conditions require both interval and congruent domain to intersect.

It is worth noting that abstract profiling (as well as comprehensive profiling) cannot distinguish between WAR/RAW dependence, where ordering relations are not kept. In future work, we will consider abstracting this order by keeping distance information among instructions.

#### 4.5 Experimental Study

Figure 1 demonstrates the effectiveness of interval and congruent domain in exploring data dependence between memory references. Figure 1-(a) shows an example where statements S1 and S2 are independent, as each memory access has its separate index values (e.g. one is even, and the other is odd). The upper (not shaded) part of Table 1 shows the corresponding assembly instructions, their addresses, and interval/congruence analysis results. Apparently, there is no intersection using the congruence domain, thus no dependence, even though the intervals intersect. On the other hand, Fig. 1-(b) shows another example where S3 and S4 are independent also, as each part of the array are accessed in two different loops. The shaded part of Table 1 shows its corresponding results. It is clear that there is no intersection in the interval domain, thus no dependence, even though the congruent domain intersects. By considering both congruent and interval intersections for reporting a dependence, this can provide for the better potential to improve precision. Thus both abstract domains provide a safe approximation to the concrete domain (i.e. absence of dependence in one abstract domain is sufficient to decide the absence of dependence on the concrete domain).

**Table 1.** The output results show the effect of the different domains on analysis.

Inst. Address	Inst. Loc.	Assembly code	Interval domain	Congruence domain
f3ed7cb	S1	movsd qword ptr [rax+rdx*8], xmm0	[7e3e0, 7e420]	10 Z + 7e3e0
f3ed7e1	S2	movsd xmm1, qword ptr [rax+rdx*8]	[7e3d8, 7e418]	10 Z + 7e3d8
0c048f2	S3	movsd qword ptr [rax+rdx*8], xmm0	[7b698, 7b6b0]	8 Z + 7b698
0c04942	S4	movsd xmm2, qword ptr [rax+rdx*8]	[7b6b8, 7b6d8]	8 Z + 7b6b8

Abstract profiling may introduce false dependence arcs. Figure 2-(a) show a simple example that clarifies the causes of the false-positive relationships that

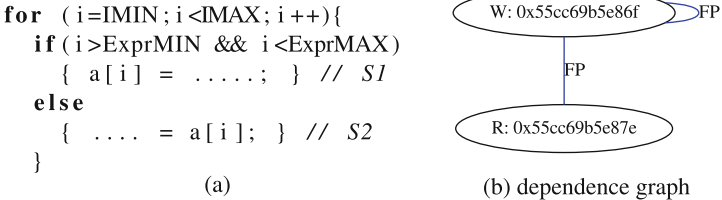


may happen. This example does not have any real data dependence relationships because of the writes at  $S1$  only access memory exclusively in the range between  $ExprMIN$  and  $ExprMAX$  of the array  $a$ , while, the reads at  $S2$  happen elsewhere. Figure 2-(b) shows the reported dependence graph contains two false-positive arcs. One arc between  $S1$  and  $S2$  because the interval of  $S1 = [ExprMIN, ExprMax]$  intersects with the interval of  $S2 = [IMIN, IMAX]$  and the related congruent values also intersect. The other arc between  $S2$  and itself representing WAW dependence.

We can enhance the precision by partitioning the memory address space into blocks (with a block size of  $2^k$  bits) and abstracting each block address space. For the previous example, the false positive eliminated if  $S1$  and  $S2$  access different blocks of memory, and hence there is no intersection between the two instruction's memory blocks. In this study, we did experiments for various memory block size, as demonstrated in the next section.

<pre> <b>for</b> (i = IMIN; i &lt; IMAX; i++)   <b>if</b> (i%2 == 0)     a[i] = ...; // S1   <b>else</b>     ... = a[i]; // S2           (a) </pre>	<pre> <b>for</b> (i = IMIN; i &lt; IMAX/2; i++){   a[i] = ...; // S3 } <b>for</b> (i = IMAX/2; i &lt; IMAX; i++){   ... = a[i]; // S4 }           (b) </pre>
---	--

**Fig. 1.** Two examples are showing the effect of the different domains on analysis.



**Fig. 2.** A simple example is showing a case of false-positive dependence arcs.

## 5 Results

We have evaluated the proposed profiling method over fifteen kernels from the polyhedral benchmark suite (PolyBench/C 3.2) and eleven programs from SPEC CPU 2006 benchmark suite to assess both the accuracy of the profiled dependencies and performance. We have conducted experiments on a machine with Intel Core I7, 16 GB memory and running Ubuntu release 18.04 (64-bit) operating system. For this study, we considered detecting two data dependence classes: RAW/WAR and WAW.

The profiling results from polyhedral benchmark suite seems pretty optimistic for our method. Results show that the method reduces execution time overhead by  $1.4\times$  on average and having a significant memory reduction  $109780\times$  on average of memory space with 100% of sensitivity and 99.05% on average of precision compared to comprehensive profiling baseline. The following part elaborates in more details SPEC profiling results, which is more challenging.

**Table 2.** Abstract analysis of SPEC2006 benchmarks.

Benchmark name	Dyn. Mem. Inst (millions)	Static. Mem. Inst	Memory Red.		Sensitivity (%) TP/(TP + FN)	Precision 1 (%) TP/(TP + FP)	Precision 2 (%) TP/(TP + FP)
			Conventional (X)	Comprehensive (X)			
401.bzip2	2,701 (29.9%)	3,068	2044.0	22032.3	100	54.1	70.4
403.gcc	653 (17.3%)	59,465	30.7	366.2	100	16.0	45.3
429.mcf	1,227 (41.2%)	743	17472.3	46907.8	100	95.9	100.0
445.gobmk	571 (22.0%)	5,744	43.0	205.9	100	17.8	80.2
456.hmmer	5,310 (38.3%)	1,132	110.8	409.3	100	39.3	83.6
458.sjeng	3,280 (20.8%)	3,269	2391.8	2489.7	100	41.1	61.4
462.libquantum	40 (17.8%)	313	58.8	201.5	100	62.0	79.6
464.h264ref	27,145 (31.7%)	10,854	856.8	1903.6	100	17.8	33.2
471.omnetpp	284 (24.6%)	2,931	168.6	390.9	100	31.1	38.4
473.astar	6,071 (28.0%)	1,698	708.5	1863.7	100	54.0	66.4
483.xalanbmk	71 (25.5%)	32,884	7.8	21.8	100	13.5	16.5

Table 2 discusses the abstract analysis profiler results. The first two columns list the benchmark programs and their dynamic read and write instruction counts (and their per cent relative to all dynamic instructions), which in turn affects the amount of runtime analysis overhead for each program. The third column lists the static memory instructions, which affects the memory overhead for the proposed abstract profiling. The fourth and fifth columns show an extreme reduction in memory usage overhead, averaging  $2172\times$  and  $6981\times$  compared to conventional and comprehensive, respectively. It is clear that our proposed profiler significantly saves memory when compared with both baseline profiling. It needs only four memory records to keep abstract data for each static memory access instruction (i.e. min and max address values for the interval domain, and  $a$  and  $b$  values for the congruent domain) compared to keeping all the executed data memory access address. This advantage enables scalability in profiling memory-intensive program, especially in low memory devices.

Finally, the last three columns report the accuracy of the proposed profiler in terms of sensitivity and precision. The sensitivity (recall) of the proposed abstract analysis achieved 100% accuracy in detecting all occurred data dependencies (as expected). The positive predictive value (precision) is calculated in the seventh and eighth columns, precision 1 and precision 2, compared to conventional and Comprehensive profiling baseline, respectively<sup>1</sup>. Results indicate that our system intends to overestimate the dependence relations; this is expected due to the underlying abstraction concept. However, the sensitivity metric is generally essential, as not missing a dependence is more important than reporting wrong dependence because it would affect the correctness of the parallelisation decision. Both sensitivity and precision are formulated by calculating  $TP$  (true

<sup>1</sup> We use two metrics as conventional profiling edges connect two dynamic instruction instances, whereas comprehensive profiling connects two static instructions; thus, ground truth is represented differently in each case.

positive; how many actual dependence pairs are detected by the proposed profiler),  $FN$  (false negative; how many missed dependence pairs), and  $FP$  (false positive, how many dependence pairs reported but are wrong). It is worth mentioning that the precision is low (61.4% on average), due to successive calls to the same functions, which leads to having some clusters of data corresponding to each call.

Figure 3 shows the slowdown for each benchmark program. We can notice the differences in runtime overhead in each benchmark program between conventional, comprehensive, and abstract profiling. Notably, the runtime overhead of abstract profiling is better than the comprehensive (10.7 $\times$ ) and the conventional (1.4 $\times$ ) profiling techniques on average, where it saves memory but needs runtime processing for every dynamic memory instruction. In future work, we will work on tackling the runtime overhead. Figure 4 shows the precision analysis and memory reduction for different memory block size. It is clear that block size is a decreasing function of precision and an increasing function of memory reduction. Block size of  $2^{12}$  bits seems to be a good design point as memory reduction is still significant (47 $\times$  average memory reduction) with 82.25% precision.

$SD^3$  [10] as related work, implemented on Pin and used a SPEC2006 to present the results compared to the pairwise method as a baseline (i.e. comprehensive). The  $SD^3$  baseline has more runtime overhead than our baseline implementation. For example, the slowdown baseline of 462.libquantum and 456.hmmer is 90 $\times$  and 210 $\times$  respectively; moreover, most of the remainder SPEC programs fail due to lack of memory resources (consumes more than 10 GB memory). On the other side, our implementation of the baseline has a slowdown 7.42 $\times$  and 71.54 $\times$ , respectively for the same two programs, and a slowdown of all tested programs 402.4 $\times$  on average. Serial  $SD^3$  has 289 $\times$  slowdown on average for the top 20 hottest loops. The parallelised version of  $SD^3$  shows a 70 $\times$  slowdown on average (8 tasks on 8 cores). The proposed profiler shows 37.4 $\times$  on average for profiling the whole program, and a memory reduction 6981 $\times$  on average.

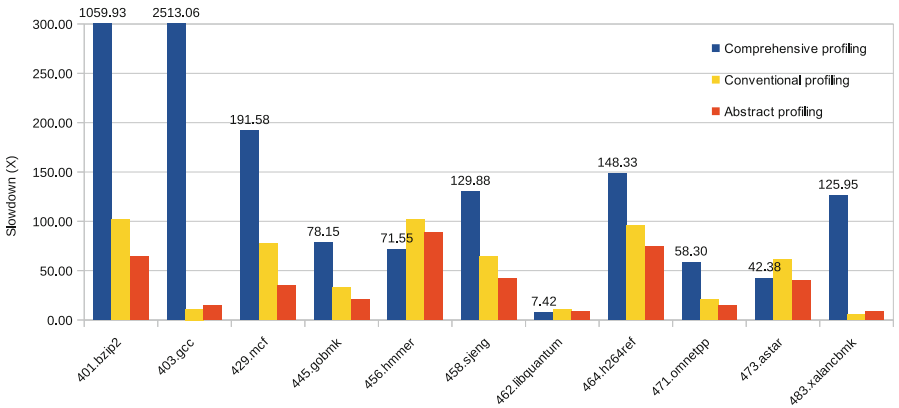


Fig. 3. Abstract profiling overhead on SPEC2006 benchmark.

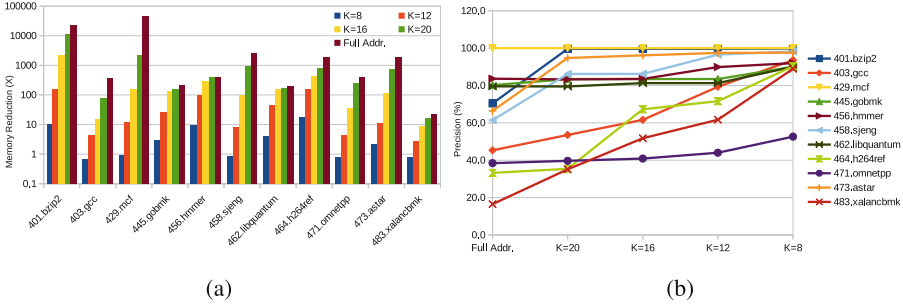


Fig. 4. Memory reduction and precision analysis for different memory block size.

Moreover, our technique applies to other approaches that intend to mitigate profiling overhead like sampling and parallelisation, which we aim to investigate in the future.

## 6 Conclusions and Future Work

This paper has considered employing a simple abstract representation based method for detecting data dependence at the binary level. This approach has combined both an interval and congruent abstract domains to detect data dependence relations, as well as abstracting partitioned memory sub address spaces. We have implemented this approach on the Pin dynamic binary instrumentation framework and conducted an initial performance study on both SPEC and polyhedral benchmark suite. Results show perfect recall rate of 100% with a significant memory reduction in comparison with the profiling baseline. Future work includes considering more elaborate abstract properties such as dependence vectors as integrating this profiling method with an automatic parallelisation framework.

## References

1. Abstract Interpretation in a Nutshell. <http://www.di.ens.fr/~7Ecouot/AI/IntroAbsInt.html>. Accessed 24 February 2018
2. Bernstein, A.J.: Analysis of programs for parallel processing. *IEEE Trans. Electron. Comput.* **5**, 757–763 (1966)
3. Bueno, F., De La Banda, M.G., Hermenegildo, M.: Effectiveness of abstract interpretation in automatic parallelization: a case study in logic programming. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **21**(2), 189–239 (1999)
4. Bygde, S.: Abstract interpretation and abstract domains. Master’s thesis, Västerås, Sweden, May 2006
5. Chen, T., Lin, J., Dai, X., Hsu, W.-C., Yew, P.-C.: Data dependence profiling for speculative optimizations. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 57–72. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-24723-4\\_5](https://doi.org/10.1007/978-3-540-24723-4_5)

6. Cousot, P.: A gentle introduction to abstract interpretation. In: The 9th International Symposium on Theoretical Aspects of Software Engineering, Nanjing, China, September 2015, pp. 1–46. <http://www.di.ens.fr/~cousot/publications.www/slides-public/PCousot-TASE-2015-AI-tutorial-4-1.pdf>
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252 (1977)
8. Cousot, P., Cousot, R.: Abstract interpretation: past, present and future. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–10 (2014)
9. Granger, P.: Static analysis of arithmetical congruences. *Int. J. Comput. Math.* **30**(3–4), 165–190 (1989)
10. Kim, M., Kim, H., Luk, C.K.: SD3: a scalable approach to dynamic data-dependence profiling. In: 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 535–546. IEEE (2010)
11. Kotha, A., Anand, K., Smithson, M., Yellareddy, G., Barua, R.: Automatic parallelization in a binary rewriter. In: 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 547–557. IEEE (2010)
12. Li, Z., Beaumont, M., Jannesari, A., Wolf, F.: Fast data-dependence profiling by skipping repeatedly executed memory operations. In: Wang, G., Zomaya, A., Perez, G.M., Li, K. (eds.) ICA3PP 2015. LNCS, vol. 9531, pp. 583–596. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-27140-8\\_40](https://doi.org/10.1007/978-3-319-27140-8_40)
13. Li, Z., Jannesari, A., Wolf, F.: An efficient data-dependence profiler for sequential and parallel programs. In: IEEE International Parallel and Distributed Processing Symposium, pp. 484–493. IEEE (2015)
14. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Conference on Programming Language Design and Implementation (2005)
15. Maydan, D., Amarsinghe, S., Lam, M.: Data dependence and data-flow analysis of arrays. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D. (eds.) LCPC 1992. LNCS, vol. 757, pp. 434–448. Springer, Heidelberg (1993). [https://doi.org/10.1007/3-540-57502-2\\_63](https://doi.org/10.1007/3-540-57502-2_63)
16. Midkiff, S.P.: Automatic Parallelization: An Overview of Fundamental Compiler Techniques. Morgan & Claypool Publishers, San Rafael (2012)
17. Moseley, T., Shye, A., Reddi, V.J., Grunwald, D., Peri, R.: Shadow profiling: hiding instrumentation costs with parallelism. In: International Symposium on Code Generation and Optimization (CGO 2007), pp. 198–208. IEEE (2007)
18. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (2015). <https://doi.org/10.1007/978-3-662-03811-6>
19. Norouzi, M., Ilias, Q., Jannesari, A., Wolf, F.: Accelerating data-dependence profiling with static hints. In: Yahyapour, R. (ed.) Euro-Par 2019. LNCS, vol. 11725, pp. 17–28. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-29400-7\\_2](https://doi.org/10.1007/978-3-030-29400-7_2)
20. Ricci, L.: Automatic loop parallelization: an abstract interpretation approach. In: Proceedings. International Conference on Parallel Computing in Electrical Engineering, pp. 112–118. IEEE (2002)
21. Sanchez, D., Yen, L., Hill, M.D., Sankaralingam, K.: Implementing signatures for transactional memory. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), pp. 123–133. IEEE (2007)

22. Tzolovski, S.: Data dependences as abstract interpretations. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 366–366. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0032756>
23. Vanka, R., Tuck, J.: Efficient and accurate data dependence profiling using software signatures. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 186–195 (2012)
24. Yu, H., Li, Z.: Multi-slicing: a compiler-supported parallel approach to data dependence profiling. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 23–33 (2012)