An Efficient Data-Dependence Profiler for Sequential and Parallel Programs

Zhen Li, Ali Jannesari, and Felix Wolf
German Research School for Simulation Sciences, 52062 Aachen, Germany
Technische Universität Darmstadt, 64289 Darmstadt, Germany
{*z.li, a.jannesari*}*@grs-sim.de, wolf@cs.tu-darmstadt.de*

***Abstract*—Extracting data dependences from programs serves as the foundation of many program analysis and transformation methods, including automatic parallelization, runtime scheduling, and performance tuning. To obtain data dependences, more and more related tools are adopting profiling approaches because they can track dynamically allocated memory, pointers, and array indices. However, dependence profiling suffers from high runtime and space overhead. To lower the overhead, earlier dependence profiling techniques exploit features of the specific program analyses they are designed for. As a result, every program analysis tool in need of data-dependence information requires its own customized profiler. In this paper, we present an efficient and at the same time generic data-dependence profiler that can be used as a uniform basis for different dependence-based program analyses. Its lock-free parallel design reduces the runtime overhead to around 86× on average. Moreover, signature-based memory management adjusts space requirements to practical needs. Finally, to support analyses and tuning approaches for parallel programs such as communication pattern detection, our profiler produces detailed dependence records not only for sequential but also for multi-threaded code.**

***Keywords*-data dependence, profiling, program analysis, parallelization, parallel programming**

I. Introduction

Single-core performance is more or less stagnating. Nonetheless, to speed up a program developers can now exploit the potential of multi-core processors and make it run in parallel. However, fully utilizing this potential is often challenging, especially when the sequential version was written by someone else. Unfortunately, in many organizations the latter is more the rule than the exception [1]. Many useful tools have been proposed to assist programmers in parallelizing sequential applications and tuning their parallel versions more easily. Tools for discovering parallelism [2], [3], [4], [5], [6], [7] identify the most promising parallelization opportunities. Runtime scheduling frameworks [8], [9], [10], [11] add more parallelism to programs by dispatching code sections in a more effective way. Automatic parallelization tools [12], [13], [14] transform sequential into parallel code automatically. However, they all have in common the fact that they rely on data-dependence information to achieve their goals because data dependences can present serious obstacles to parallelization.

Data dependences can be obtained in two main ways: static and dynamic analysis. Static approaches determine data dependences without executing the program. Although they are fast and even allow fully automatic parallelization in some cases [13], [14], they lack the ability to track dynamically allocated memory, pointers, and dynamically calculated array indices. This usually makes their assessment pessimistic, limiting their practical applicability. In contrast, dynamic dependence profiling captures only those dependences that actually occur at runtime. Although dependence profiling is inherently input sensitive, the results are still useful in many situations, which is why such profiling forms the basis of many program analysis tools [2], [5], [6]. Moreover, input sensitivity can be addressed by running the target program with changing inputs and computing the union of all collected dependences.

However, a serious limitation of data-dependence profiling is high runtime overhead in terms of both time and space. The former may significantly prolong the analysis, sometimes requiring an entire night [15]. The latter may prevent the analysis completely [16]. This is because dependence profiling requires all memory accesses to be instrumented and records of all accessed memory locations to be kept. To lower the overhead, current profiling approaches limit their scope to the subset of the dependence information needed for the analysis they have been created for, sacrificing generality and, hence, discouraging reuse. Moreover, since current profilers mainly concentrate on the discovery of parallelization opportunities, they only support sequential programs, although they are also needed for parallel programs. For example, there may still be unexploited parallelism hidden inside a parallel program. Furthermore, knowledge of data communication patterns, which are nothing but cross-thread dependences, can help identify critical performance bottlenecks. Finally, debugging approaches such as data race detection can also benefit from data dependence information to improve their accuracy.

To provide a general foundation for all such analyses, we present the first generic data-dependence profiler with practical overhead, capable of supporting a broad range of dependence-based program analysis and optimization techniques—both for sequential and parallel programs. To achieve efficiency in time, the profiler is parallelized, taking advantage of lock-free design [17]. To achieve efficiency in space, the profiler leverages signatures [18], a concept borrowed from transactional memory. Both optimizations are

application-oblivious, which is why they do not restrict the profiler's scope in any way. Our profiler has the following specific features:

- It collects pair-wise data dependences of all the three types (RAW, WAR, WAW) along with runtime control-flow information
- It is efficient with respect to both time and memory (average slowdown of only 86×, average memory consumption of only 1020 MB for benchmarks from NAS and Starbench)
- It supports both sequential and parallel (i.e., multi-threaded) target programs
- It provides detailed information, including source code location, variable name, and thread ID

The remainder of the paper is organized as follows. First, we summarize related work in Section II. Then, we describe our profiling approach for sequential target programs in Section III. At this point, emphasis is given to the reduction of space overhead. In Section IV, we describe the efficient parallelization. An extension in support of multi-threaded target programs is presented in Section V. We evaluate the accuracy and performance of the full profiler design in Section VI, while we showcase several applications of our profiler in Section VII. Finally, we conclude the paper and outline future prospects in Section VIII.

II. Related Work

After purely static data-dependence analysis turned out to be too conservative in many cases, a range of predominantly dynamic approaches emerged. In previous work, their overhead was reduced either by tailoring the profiling technique to a specific analysis or by parallelizing it.

Using dependence profiling, Kremlin [2] determines the length of the critical path in a given code region. Based on this knowledge, it calculates a metric called self-parallelism, which quantifies the parallelism of the region. Instead of pair-wise dependences, Kemlin records only the length of the critical path. Alchemist [4], a tool that estimates the effectiveness of parallelizing program regions by asynchronously executing certain language constructs, profiles dependence distance instead of detailed dependences. Although these approaches profile data dependences with low overhead, the underlying profiling technique has difficulty in supporting support other program analyses.

There are also approaches that reduce the time overhead of dependence profiling through parallelization. For example, SD3 [16] exploits pipeline and data parallelism to extract data dependences from loops. At the same time, SD3 reduces the significant space overhead of tracing memory accesses by compressing strided accesses using a finite state machine. Multi-slicing [19] follows the same compression approach as SD3 to reduce the memory overhead, but leverages compiler support for parallelization. Before execution, the compiler divides the profiling job into multiple profiling tasks through a series of static analyses, including alias/edge partitioning, equivalence classification, and thinned static analysis. According to published results, the slowdown of these approaches stays close to ours when profiling the hottest 20 loops (70× on average using SD3 with 8 threads), but remains much higher when profiling whole programs (over 500× on average using multi-slicing with 8 threads).

Like SD3 and multi-slicing, we parallelize the data-dependence profiling algorithm instead of customizing it. Unlike these methods, we profile detailed data dependences and control-flow information for not only sequential but also multi-threaded programs. Furthermore, our parallelization is achieved through lock-free programming, ensuring good performance without loss of generality.

```
1  1:60 BGN loop
2  1:60 NOM {RAW 1:60|i}     {WAR 1:60|i}
3           {INIT *}
4  1:63 NOM {RAW 1:59|temp1} {RAW 1:67|temp1}
5  1:64 NOM {RAW 1:60|i}
6  1:65 NOM {RAW 1:59|temp1} {RAW 1:67|temp1}
7           {WAR 1:67|temp2} {INIT *}
8  1:66 NOM {RAW 1:59|temp1} {RAW 1:65|temp2}
9           {RAW 1:67|temp1} {INIT *}
10 1:67 NOM {RAW 1:65|temp2} {WAR 1:66|temp1}
11 1:70 NOM {RAW 1:67|temp1} {INIT *}
12 1:74 NOM {RAW 1:41|block}
13 1:74 END loop 1200
```

Figure 1. A fragment of profiled data dependences in a sequential program.

III. Data-Dependence Profiling

To explain how we reduce the space overhead via signatures, we start with the profiling approach that supports only sequential programs. Our profiler, which is implemented in C++11 based on LLVM [20], delivers the following information:

- pair-wise data dependences
- source code locations of dependences and the names of the variables involved
- runtime control-flow information

We profile detailed pair-wise data dependences because we want to support as many program analyses as possible. Control-flow information is necessary for some program analyses such as parallelism discovery and code partitioning.

A. Representation of data dependences

A sample piece of profiling data is shown in Figure 1. A data dependence is represented as a triple `<sink, type, source>`. `type` is the dependence type (RAW, WAR or WAW). Note that a special type `INIT` represents the first write operation to a memory address.

`source` and `sink` are the source code locations of the former and the latter memory accesses, respectively. `sink` is further represented as a pair

<fileID:lineID>, while source is represented as a triple <fileID:lineID|variableName>. As shown in Figure 1, data dependences with the same sink are aggregated together.

The keyword NOM (short for "NORMAL") indicates that the source line specified by aggregated sink has no control-flow information. Otherwise, BGN and END represent the entry and exit point of a control region, respectively. In Figure 1, a loop starts at source line 1:60 and ends at source line 1:74. The number following END loop shows the actual number of iterations executed, which is 1200 in this case.

B. Profiling with signatures

Traditional data-dependence profiling approaches record memory accesses using shadow memory. In shadow memory, the access history of addresses is stored in a table where the index of an address is the address itself. This approach results in a table covering the memory space from the lowest to the highest address accessed by the target program, which wastes a lot of memory. Although this problem can be partially solved by using multilevel tables, the memory overhead of shadow memory is still too high. According to previous work [16], it is often impossible to profile even small programs using shadow memory if no more than 16 GB of memory is available.

An alternative is to record memory accesses using a hash table, but this approach incurs additional time overhead since when more than one address is hashed into the same bucket, the bucket has to be searched for the address in question. Note that profiling data dependence pair-wise requires an exhaustive instrumentation on all memory accesses in the target program. The number of memory accesses in an ordinary benchmark can easily reach one billion. With all these accesses instrumented, a tiny time cost of the instrumentation function will accumulate into a huge overhead. Based on our experiments, the hash table approach is about 1.5 – 3.7× slower than our approach.

A solution to decrease the profiling overhead is to use approximate representation rather than instrument every memory access. Previous work [21] tried to ignore memory accesses in a code section when it had been executed more than 2^{32-k} times. However, when setting $k = 10$, only 33.7% of the memory accesses are covered, which can lead to significant inconsistency in profiled data dependences.

To lower the memory overhead without increasing the time overhead, we record memory accesses in signatures. A signature is a data structure that encodes an approximate representation of an unbounded set of elements with a bounded amount of state [18]. It is widely used in transactional memory systems to uncover conflicts. A signature usually supports three operations:

- Insertion: inserts a new element into the signature. The state of the signature is changed after the insertion.

```
Global signatures sig_write and sig_read

for each memory access c in the program do
    index = hash(c)
    if c is write operation then
        if sig_write[index] is empty then
            c is initialization
        end
        else
            if sig_read[index] is not empty then
                buildWAR()
            end
            buildWAW()
        end
        sig_write[index] = source line number of c
    end
    else
        if sig_write[index] is not empty then
            buildRAW()
        end
        sig_read[index] = source line number of c
    end
end
```

Algorithm 1: Algorithm for signature-based data-dependence profiling (pseudocode).

- Membership check: tests whether an element is already a member of the signature.
- Disambiguation: intersects two signatures. If an element was inserted into both of them, the resulting element must be present in the intersection.

A data dependence is similar to a conflict in transactional memory because it exists only if two or more memory operations access the same memory location in some order. Therefore, a signature is also suitable for detecting data dependences. Usually, a signature is implemented as a bloom filter [22], which is a fixed-size bit array with k different hash functions that together map an element to a number of array indices. Here, we adopt a similar idea, using a fixed-length array combined with a hash function that maps memory addresses to array indices. We use only one hash function to simplify the removal of elements because it is required by variable lifetime analysis, an optimization we implemented to lower the probability of building incorrect dependences. In variable lifetime analysis, addresses that become obsolete after deallocating the corresponding variable are removed from signatures. Also, each slot of the array is three bytes long instead of one bit so that the source line number where the memory access occurs can be stored in it. Because of the fixed length of the data structure, memory consumption can be adjusted as needed.

To detect data dependences, we apply Algorithm 1. It deploys two signatures: one for recording read operations

and one for recording write operations. When a memory access c at address x is captured, we first determine the access type (read or write). Then, we run the membership check to see if x exists in the signatures. If x already exists, we build a data dependence and change the source line number to where c occurred. Otherwise, we insert x into the signature. Note that we ignore read-after-read (RAR) dependences because in most program analyses they are not required.

With signature, we sacrifice a slight degree of accuracy of profiled dependence for profiling speed. When more than one address is hashed into the same slot, false dependences are created instead of building additional data structures to keep the addresses, saving time for maintaining the structures and searching the address from them. Signatures are implemented in fixed-size arrays so that the overhead of new/delete or malloc/free is eliminated.

A signature is an approximate representation where hash collisions can happen. A hash collision in signatures can lead to both false positives and false negatives in profiled dependences. In Section VI-A, we show that the false positive and false negative rates of profiled dependences are negligible if sufficiently large signatures are used. Nonetheless, sufficiently large is still small in comparison to shadow memory. If an estimation of the total number of memory address accesses in the target program is available, the signature size can also be estimated using formula 2 in Section VI-A. A very practical alternative is to use all the memory of the target system for profiling that remains after subtracting the memory space needed for the target program itself, which is usually more than enough to yield perfect dependences.

Finally, we merge identical dependences to reduce the runtime memory overhead and the time needed to write the dependences to disk. Based on our experience, this step is necessary to arrive at a practical solution. Merging identical dependences decreased the average output file size for NAS benchmarks from 6.1 GB to 53 KB, corresponding to an average reduction by a factor of 10^5.

IV. Parallelization

The basic idea behind the parallelization of our approach is to run the profiling algorithm in parallel on disjoint subsets of the memory accesses. To determine the dependence type (RAW, WAR, WAW) correctly, we need to preserve the temporal order of memory accesses to the same address. For this reason, a memory address is assigned to exactly one worker thread, which becomes responsible for all accesses to this address. To buffer incoming memory accesses before they are consumed, we use a separate queue for each worker thread, which can fetch data only from the queue assigned to it.

Figure 2 shows how our parallel design works. The main thread executes the program to be analyzed and collects

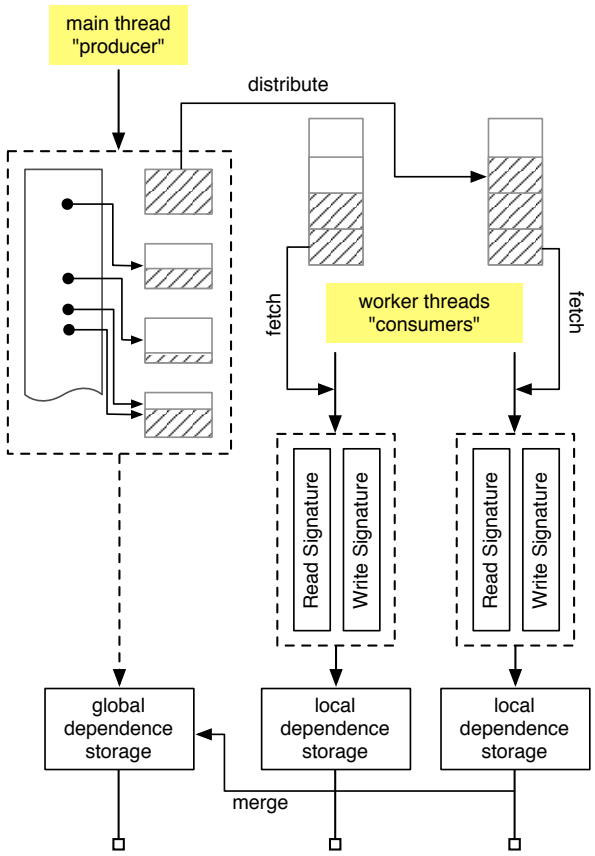

Figure 2. Architecture of a parallel data-dependence profiler for sequential programs.

memory accesses in chunks, whose size can be configured in the interest of scalability. One chunk contains only memory accesses assigned to one thread. Once a chunk is full, the main thread pushes it into the queue of the thread responsible for the accesses recorded in it. The worker threads in turn consume chunks from their queues, analyze them, and store detected data dependences in thread-local maps. Empty chunks are recycled and can be reused. The use of maps ensures that identical dependences are not stored more than once. At the end, we merge the data from all local maps into a global map. This step incurs only minor overhead since the local maps are free of duplicates. Since the major synchronization overhead comes from locking and unlocking the queues, we made the queues lock-free to lower the overhead.

A. Load balancing

In our profiler, memory accesses are distributed among worker threads using a simple modulo function:

$$worker\_ID = memory\_address \quad \% \quad W \qquad (1)$$

```
4:58|2 NOM {WAR 4:77|2|iter}
4:59|2 NOM {WAR 4:71|2|z_real}
4:64|3 NOM {RAW 3:75|0|maxiter}
   {RAW 4:58|3|iter}    {RAW 4:61|3|z_norm}
   {RAW 4:71|3|z_norm}  {RAW 4:73|3|iter}
4:69|3 NOM {RAW 4:57|3|c_real}
   {RAW 4:66|3|z2_real} {WAR 4:67|3|z_real}
4:71|2 NOM {RAW 4:69|2|z_real}
   {RAW 4:70|2|z_imag}  {WAR 4:64|2|z_norm}
4:80|1 NOM {WAW 4:80|1|green} {INIT *}
```

Figure 3. A fragment of data dependences from a parallel program captured by our profiler. Thread IDs are highlighted.

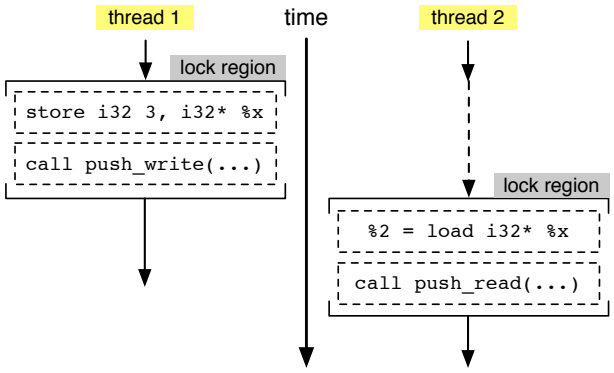

Figure 4. Instrumentation functions for memory accesses are always inserted in the same lock region.

with W being the number of worker threads. According to our experiments, this simple function achieves an even distribution of accessed memory addresses. A similar conclusion is also drawn in SD3 [16]. Although memory addresses are distributed evenly, not all of them are accessed with the same frequency. Some addresses may be accessed millions of times while others are only accessed a few times. To avoid the situation where all heavily accessed addresses are assigned to the same worker thread, we also monitor how many times an address is accessed dynamically. These access statistics are stored in a map and updated every time a memory access occurs. The access statistics are needed to ensure that the top ten most heavily accessed addresses are always evenly distributed among worker threads.

The access statistics are evaluated at regular intervals. If we notice that the distribution of heavily accessed memory addresses is out of balance, we initiate redistribution. If an address is moved to another thread, its signature state has to be moved as well. After redistribution, accesses to redistributed addresses will always be directed to the newly assigned worker thread. Redistribution rules are stored in a map and have higher priority than the modulo function.

Redistribution is costly, which is why it should not be performed too frequently. In our implementation, we check whether redistribution is needed after every 50,000 chunks. Consequently, for the benchmarks used in this paper, redistribution is performed at most 20 times when profiling a single benchmark, which is enough to have a positive impact on the time overhead.

V. Multi-Threaded Target Programs

A data dependence in a parallel program is still represented as triple `<sink, type, source>`. However, to distinguish different threads, we add thread IDs to the `sink` and `source` fields. Now, `sink` has the form `<fileID:lineID|threadID>` and `source` has the form `<fileID:lineID|threadID|variableName>`. Control-flow information is recorded in the same way as shown earlier in Section III. Figure 3 shows a fragment of dependences captured in a parallel program.

A. Modified parallelization strategy

In a sequential program, the temporal order of memory accesses is automatically preserved. Thanks to this property, we can easily ensure that our parallel profiler produces the same data dependences as the serial version—provided we push a memory access into chunk immediately after encountering it. However, parallel programs do not have this property. In a multi-threaded environment it is not guaranteed that the push operation is always executed immediately after the memory access, resulting in incorrect data dependences.

To solve this problem, we need to make a memory access and its corresponding push operation atomic. Thus, we require that accesses to the same address from multiple threads are protected by locks, and we insert the push operation into the same lock region, as shown in Figure 4. So far we support only parallel programming languages where locking/unlocking primitives have to be written explicitly in the source code. However, programing languages with implicit synchronization can be easily supported by automatically discovering implicit synchronization patterns. [23]

B. Data races

We generally do not know whether the cross-thread dependences we report are enforced or not, that is, whether they will be reproduced when the program is run again. In this sense, they can also be regarded as incidental *happens-before* relationships. Unless a data race is acceptable, a correct program would always enforce such dependences. An example of an acceptable data race is the concurrent update of a flag indicating whether a parallel search was successful. Since acceptable data races are rare, it is usually desirable to know whether a dependence is enforced or not. One way of detecting unenforced dependences is to run the

Table I
FALSE POSITIVE AND FALSE NEGATIVE RATES OF PROFILED DEPENDENCES FOR STARBENCH.

Program	LOC	# addresses	# accesses	# dependences	# slots = 1.0E+6		# slots = 1.0E+7		# slots = 1.0E+8	
					FPR	FNR	FPR	FNR	FPR	FNR
c-ray	620	1.1E+6	1.9E+9	574	19.95	1.34	6.00	0.24	0.00	0.00
kmeans	603	6.9E+5	1.9E+9	281	5.46	0.75	0.21	0.00	0.00	0.00
md5	661	2.6E+5	4.8E+8	859	3.08	0.15	0.03	0.00	0.00	0.00
ray-rot	1425	4.0E+5	9.8E+8	862	11.82	1.64	1.19	0.00	0.00	0.00
rgbyuv	483	6.3E+6	2.1E+8	155	47.67	15.74	4.44	1.90	0.21	0.05
rotate	871	3.1E+6	3.7E+8	278	55.92	15.68	4.50	3.02	0.00	0.00
rot-cc	1122	6.3E+6	4.9E+8	372	63.15	19.52	24.08	2.04	0.39	0.00
streamcluster	860	8.6E+3	1.2E+7	780	1.55	0.42	0.84	0.12	0.55	0.06
tinyjpeg	1922	4.2E+2	2.3E+7	1711	17.37	0.54	4.23	0.11	0.02	0.00
bodytrack	3614	4.4E+6	4.8E+9	3422	25.07	3.56	3.69	0.32	0.75	0.30
h264dec	42 822	8.7E+5	3.6E+8	31 138	18.10	0.23	2.63	0.03	1.95	0.01
average	—	—	—	—	24.47	5.42	4.71	0.71	0.35	0.04

program more than once and hope that a different thread schedule will reverse the order and expose the race. Because this can be a successful strategy for finding races, reporting potentially irreproducible dependences is also valuable from a correctness perspective.

However, there are also cases where we can actually prove the occurrence of a data race even after a single run. The situation where the atomicity of access occurrence and reporting is violated can only happen if there are no synchronization mechanisms in place to keep the two accesses to memory location mutually exclusive. For this reason, the reported dependence may show the reverse of the actual execution order. To catch such cases, we acquire the timestamp of every memory access and pass it to the corresponding push operation as a parameter. Whenever a worker thread fetches memory accesses from its queue it usually expects increasing timestamps. A violation of this condition indicates that the memory accesses were pushed in a different order from the one in which they occurred. In this case, we mark the dependence accordingly. Moreover, whenever we see such a reversal, we can conclude that the memory accesses were not guaranteed to be mutually exclusive. Although mutual exclusion does not necessarily enforce a particular access order, its absence definitely exposes a potential data race.

VI. EVALUATION

We conducted a range of experiments to evaluate both the accuracy of the profiled dependences and the performance of our implementation. Test cases are the NAS Parallel Benchmarks 3.3.1 [24] (NAS), a suite of programs derived from real-world computational fluid-dynamics applications, and the Starbench parallel benchmark suite [25] (Starbench), which covers programs from diverse domains, including image processing, information security, machine learning and so on. Whenever possible, we tried different inputs to compensate for the input sensitivity of dynamic dependence profiling.

A. Accuracy of profiled dependences

We first evaluate the accuracy of the profiled data dependences since we build upon the idea of a signature as an approximate representation of memory accesses. As it is described in Section III-B, the membership check of this approximate representation can deliver false positives, which further lead to false positives and false negatives in dependences.

To measure the false positive rate (FPR) and the false negative rate (FNR) of the profiled dependences, we implemented a "perfect signature", in which hash collisions are guaranteed not to happen. Essentially, the perfect signature is a table where each memory address has its own entry, so that false positives are never produced. We use the perfect signature as the baseline to quantify the FPR and the FNR of the dependences delivered by our profiler.

Table I shows the results for Starbench. Three groups of FPR and FNR are shown under three different signature sizes in terms of the total number of slots. When using 1.0E+6 slots, the average FPR and FNR are 24.47% and 5.42%, respectively. The values are significantly reduced to 4.71% and 0.71% when the signature size is increased to 1.0E+7. Finally, hardly any incorrect dependences appear when the signature has 1.0E+8 slots as the average value of both FPR and FNR are lower than 0.4%. In our implementation, each slot is four bytes. Thus, 1.0E+8 slots consume only 382 MB of memory, which is adequate for any ordinary PC.

c-ray, *rgbyuv*, *rotate*, *rot-cc* and *bodytrack* have higher FPR and FNR than other programs because they access a large number of different addresses. This observation matches the theory of predicting the false positive rate of a signature. Assume that we use a hash function that selects each array slot with equal probability. Let m be the number of slots in the array. Then, the estimated false positive rate (P_{fp}), i.e., the probability that a certain slot is *used* after inserting n elements is:

$$P_{fp} = 1 - (1 - \frac{1}{m})^n. \quad (2)$$

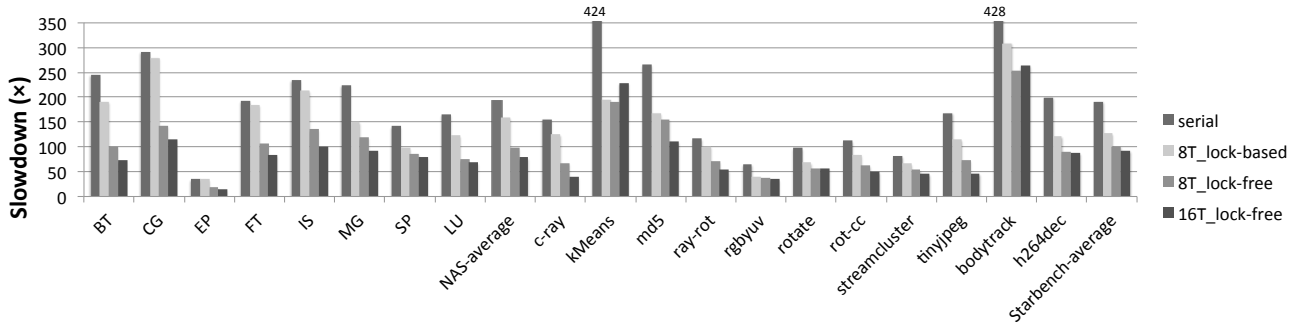

Figure 5. Slowdowns caused by the data-dependence profiler for sequential NAS and Starbench benchmarks.

Clearly, P_{fp} is inversely proportional to m and proportional to n. In our case, m is the size of signature and n is the number of addresses.

B. Performance

We conducted our performance experiments on a server with 2 x 8-core Intel Xeon E5-2650 2 GHz processors with 32 GB memory, running Ubuntu 12.04 (64-bit server edition). All the test programs were compiled with option `-g -O2` using Clang 3.3. For NAS, we used the input set W; for Starbench, we used the reference input set.

1) Time overhead: First, we examine the time overhead of our profiler. The number of threads for profiling is set to 8 and 16. The slowdown figures are average values of three executions compared with the execution time of uninstrumented runs. The negligible time spent in the instrumentation is not included in the overhead. For NAS and Starbench, instrumentation was always done in two seconds.

The slowdown of our profiler when profiling sequential programs is shown in Figure 5. The average slowdowns for the two benchmark suites ("NAS-average" and "Starbench-average") are also included. As the figure shows, our serial profiler has a 190× slowdown on average for NAS benchmarks and a 191× slowdown on average for Starbench programs. The overhead is not surprising since we perform an exhaustive profiling for the whole program.

When using 8 threads, our lock-free parallel profiler gives a 97× slowdown on average for NAS benchmarks and a 101× slowdown on average for Starbench programs. After increasing the number of threads to 16, the average slowdown is only 78× for NAS benchmarks, and 93× for Starbench programs. Compared to the serial profiler, our lock-free parallel profiler achieves a 2.4× and a 2.1× speedup using 16 threads on NAS and Starbench benchmark suites, respectively.

Our profiler may seems slightly slower than SD3, which has a 70× slowdown on average using eight threads [16]. However, the slowdown of SD3 is measured by profiling the hottest 20 loops from each benchmark. When profiling data dependences all across the target program, multi-slicing, which is essentially SD3 but parallelized in a different way, has a more than 500× slowdown on average using eight threads [19].

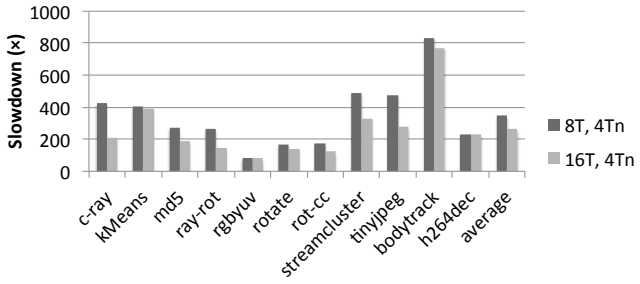

Figure 6. Slowdown of the profiler for parallel Starbench programs (pthread version, T = thread for profiling, T_n = thread for benchmark).

The speedup is not linear for two reasons. Firstly, data-dependence profiling always has imbalanced workload due to uneven accesses, as we discussed in Section IV-A. In this case, simply introducing more worker threads does not help balance the workload. Similar behavior is also observed in related work [19]. Profiling performance is affected by this problem on five benchmarks: *kMeans*, *rgbyuv*, *rotate*, *bodytrack* and *h264dec*.

Secondly, determining detailed data dependence types (RAW, WAR, WAW) requires retaining the temporal order of memory accesses to the same address, which means such accesses have to be processed sequentially. Obviously, determining only a binary value (whether a dependence exists or not) instead of detailed types would allow a more balanced workload and lead to better performance. Moreover, the performance of the profiler can be further improved by performing set-based profiling, which tells whether a data dependence exists between two code sections instead of two statements. However, all these optimization will decrease the generality of the profiler, which is contrary to our purpose.

Figure 5 also shows the slowdown of our lock-based profiler when eight threads are used. Compared to the lock-

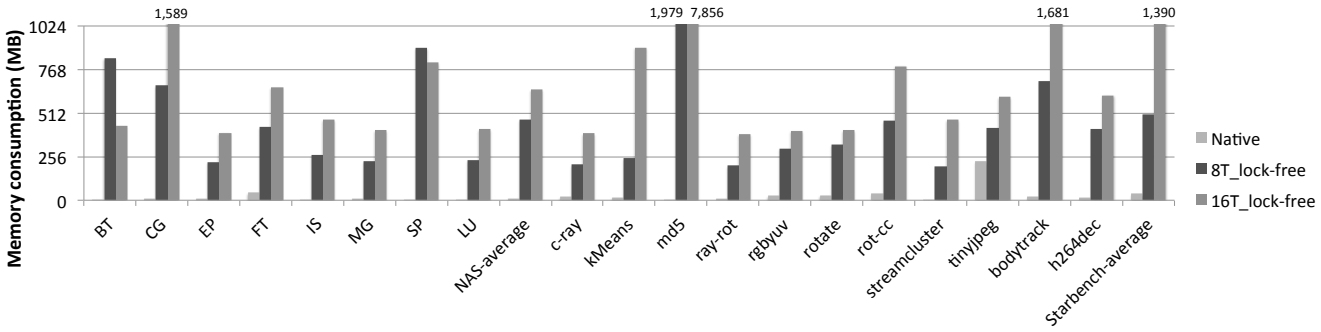

Figure 7. Memory consumption of the profiler for sequential NAS and Starbench benchmarks.

based version, our lock-free version gives a 1.6× speedup on average for NAS benchmarks, and a 1.3× speedup on average for Starbench programs. A faster lock-free implementation that only allocates memory but never de-allocates will further boost performance, but increase the memory overhead significantly.

When profiling multi-threaded code, our profiler has a higher time overhead because more contentions are introduced. Native execution time of the parallel benchmarks is calculated by accumulating the time spent in each thread.

Slowdowns of our profiler for parallel Starbench programs (pthread version, 4 threads) are shown in Figure 6. We only tested Starbench because our profiler currently requires parallel programs with explicit locking/unlocking primitives. Using eight threads for profiling, the average slowdown of our profiler for Starbench is 346×, and further decreases to 261× when 16 threads are used for profiling. Again, *kMeans*, *rgbyuv*, *rotate*, *bodytrack* and *h264dec* do not scale well because of their imbalanced memory access pattern.

2) Memory consumption: We measure memory consumption using the "maximum resident set size" value provided by `/usr/bin/time` with the verbose (`-v`) option. Figure 7 shows the results when 6.25E+6 signature slots are used in each thread, which aggregated to 1.0E+8 slots in total of 16 threads. This configuration leads to a memory consumption of 191 MB and 382 MB by the signatures for 8 threads and 16 threads, respectively.

When using 8 threads, our profiler consumes 473 MB of memory on average for NAS benchmarks and 505 MB of memory on average for Starbench programs. After increasing the number of threads to 16, the average memory consumption is increased to 649 MB and 1390 MB for NAS and Starbench programs, respectively. The worst case happens when 16 threads are used to profile *md5*, which consumes about 7.6 GB of memory. Although this may exceed the memory capacity configured in a three-year-old PC, it is still adequate for up-to-date machines, not to mention servers that are usually configured with 16 GB memory or more.

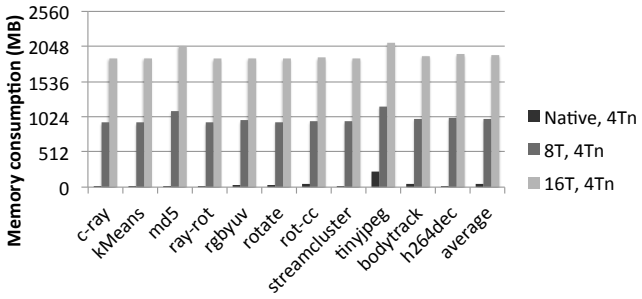

Figure 8. Memory consumption of the profiler for parallel Starbench programs (pthread version, T = thread for profiling, T_n = thread for benchmark).

Memory consumption of our profiler for parallel Starbench programs (pthread version, 4 threads) is shown in Figure 8. Our profiler consumes 995 MB and 1920 MB memory on average using 8 and 16 threads for profiling, respectively. The consumption is higher than profiling sequential benchmarks (505 MB and 1390 MB) because of the different implementation of lock-free queues, additional data structures to record thread interleaving events, and extended representation of data dependency. However, the consumption is still moderate for an ordinary PC.

VII. Application

In this section, we demonstrate two applications of our profiler, showcasing its ability to support a broad variety of program analyses. First, we apply our profiler to discover potential parallelism in loops of sequential programs. After that, we examine how our profiler can be used to detect communication patterns in multi-threaded code. If not stated, we always use signatures big enough to produce dependences without false positives and false negatives.

A. Discovering potential loop parallelism

To show how our profiler supports dependence-based program analysis and further evaluates the quality of the data dependences it delivers, we fed DiscoPoP [3], a tool

Table II
DETECTION OF PARALLELIZABLE LOOPS IN NAS BENCHMARKS.

Program	# OMP	# identified (DP)	# identified (sig)	# missed (sig)
BT	30	30	30	0
SP	34	34	34	0
LU	33	33	33	0
IS	11	8	8	0
EP	1	1	1	0
CG	16	9	9	0
MG	14	14	14	0
FT	8	7	7	0
Overall	147	136	136	0

to detect potential parallelism in sequential programs, with the output of our profiler. With its own data-dependence profiling component (no wrong dependences, equivalent to a perfect signature), DiscoPoP was able to identify 92.5% of the loops in the sequential versions of the NAS benchmarks that appear parallelized in the parallel ones. With sufficiently large signatures, we expect that our dependences will lead to the same results.

Table II shows the results of the experiment. The column "# OMP" shows the number of annotated loops in the OpenMP versions of the benchmarks. The two columns labeled with "# identified" list how many annotated loops were identified as parallelizable by DiscoPoP (DP) and our profiler (sig), respectively. As shown in Table II, as many as 92.5% (136/147) of the annotated loops were identified based on our profiled dependences, and the loops identified in each benchmarks are exactly the same as those identified by DiscoPoP.

This application proves that our profiler can support parallelism discovery techniques, and the quality of the profiled dependences are high enough to support such techniques when using sufficiently large signatures. With the help of our profiler, parallelism discovery techniques can share a common base where they can benefit from both high-quality dependence and efficiency.

B. Detecting communication patterns

The performance of parallel applications very often depends on efficient communication. This is as true for message passing as it is for communication via shared variables. Knowing the communication pattern of a shared-memory kernel can therefore be important to discover performance bottlenecks such as true sharing or to support software-hardware co-design. In shared-memory programming, communication often follows the pattern of producer and consumer. The producer thread writes a variable, after which the consumer thread reads the written value. The read happens before the next write occurs. Such a pattern can be represented as a matrix, showing the communication intensity between producer and consumer threads.

Producer-consumer behavior describes a read-after-write relation between memory operations, which can be easily derived from the RAW dependences produced by our profiler. With detailed information such as thread IDs available, we can generate the communication matrix directly from the output of our profiler. Figure 9 shows the communication pattern of *splash2x.water-spatial* [26] that we computed. The ticks of the vertical and horizontal axes represent producer and consumer threads, respectively. The darker the square the stronger communication between the the two threads. Compared to an earlier analysis [27], we identified exactly the same communication pattern.

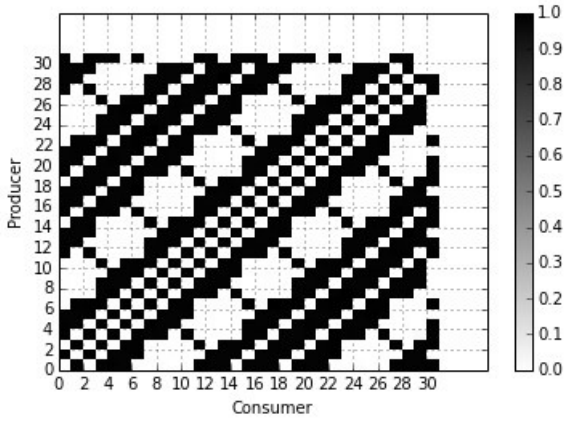

Figure 9. Communication pattern of *splash2x.water-spatial* derived from the output of our profiler.

Previous approaches [27], [28] that characterize communication patterns are usually built on top of simulators, which can easily have a slowdown of more than a factor of 1,000× if in-order processing is required (and it is required to produce communication patterns as producers and consumers need to be distinguished). With the help of our profiler, the same communication patterns can be obtained more efficiently since our profiler has only a 261× slowdown on average when profiling multithreaded Starbench benchmarks.

VIII. CONCLUSION

Many program analysis and tuning tools are built on top of data-dependence profilers. To keep the profiling overhead within reasonable limits, the underlying profilers are usually customized so that only the information needed for a specific analysis or tool is collected. This solution leads to a dissatisfactory situation: every time a new analysis tool is constructed, existing profilers cannot be reused. Creating a new one is not only expensive and inconvenient, but it also makes the final analyses or tools hard to compare since they are based on different profiling techniques.

To enable reuse without having to accept compromises in terms of efficiency, we developed a parallel and lock-free data-dependence profiler that can serve as a uniform basis for different dependence-based analyses. While its time

and space overhead stays within practical limits, our profiler also supports multi-threaded code. In this way, it supports not only date-dependence analyses for multi-threaded code, but also tuning and debugging approaches where the necessary information can be derived from dependences. While performing an exhaustive dependence search with 16 profiling threads, our lock-free parallel design limited the average slowdown to 78× and 93× for NAS and Starbench, respectively. Using a signature with 10^8 slots, the memory consumption did not exceed 649 MB (NAS) and 1390 MB (Starbench), while producing less than 0.4% false positives and less than 0.1% false negatives.

An integrated program-analysis framework with APIs to retrieve dependence information is already in development. The framework reorganizes profiled data into multiple representations, including dynamic execution tree, call tree, dependence graph, loop table, etc., and a dependence-based program analysis can be implemented as a plugin. Our plan is to release an open-source version of our tools in the near future.

ACKNOWLEDGMENTS

We thank our students Tuan Dung Nguyen and Wolfram M. Gottschlich for their invaluable programming support. We also express our gratitude to Michael Beaumont and Janet Carter-Sigglow for their very helpful feedback on language issues.

REFERENCES

[1] R. E. Johnson, "Software development is program transformation," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. ACM, 2010, pp. 177–180.

[2] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. ACM, 2011, pp. 458–469.

[3] Z. Li, A. Jannesari, and F. Wolf, "Discovery of potential parallelism in sequential programs," in *Proceedings of the 42nd International Conference on Parallel Processing*, ser. PSTI '13. IEEE Computer Society, 2013, pp. 1004–1013.

[4] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. IEEE Computer Society, 2009, pp. 47–58.

[5] M. Kim, H. Kim, and C.-K. Luk, "Prospector: Discovering parallelism via dynamic data-dependence profiling," in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, ser. HOTPAR '10, 2010.

[6] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 45. IEEE Computer Society, 2012, pp. 437–448.

[7] S. Rul, H. Vandierendonck, and K. De Bosschere, "Function level parallelism driven by data dependencies," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 55–62, Mar. 2007.

[8] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. IEEE Computer Society, 2005, pp. 105–118.

[9] D. I. August, J. Huang, S. R. Beard, N. P. Johnson, and T. B. Jablin, "Automatically exploiting cross-invocation parallelism using runtime information," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '13. IEEE Computer Society, 2013, pp. 1–11.

[10] R. Govindarajan and J. Anantpur, "Runtime dependence computation and execution of loops on heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '13. IEEE Computer Society, 2013, pp. 1–10.

[11] J. M. Ye and T. Chen, "Exploring potential parallelism of sequential programs with superblock reordering," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, ser. HPCC '12. IEEE Computer Society, 2012, pp. 9–16.

[12] S.-I. Lee, T. Johnson, and R. Eigenmann, "Cetus - an extensible compiler infrastructure for source-to-source transformation," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, L. Rauchwerger, Ed. Springer Berlin Heidelberg, 2004, vol. 2958, pp. 539–553.

[13] M. Amini, O. Goubier, S. Guelton, J. O. Mcmahon, F. Xavier Pasquier, G. Péan, and P. Villalon, "Par4All: From convex array regions to heterogeneous computing," in *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques*, ser. IMPACT 2012, 2012.

[14] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly - performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[15] S. Rul, H. Vandierendonck, and K. De Bosschere, "A profile-based tool for finding pipeline parallelism in sequential programs," *Parallel Comput.*, vol. 36, no. 9, pp. 531–551, Sep. 2010.

[16] M. Kim, H. Kim, and C.-K. Luk, "SD3: A scalable approach to dynamic data-dependence profiling," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 43. IEEE Computer Society, 2010, pp. 535–546.

[17] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, May 2007.

[18] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing signatures for transactional memory," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. IEEE Computer Society, 2007, pp. 123–133.

[19] H. Yu and Z. Li, "Multi-slicing: A compiler-supported parallel approach to data dependence profiling," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. ACM, 2012, pp. 23–33.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2nd International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[21] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with LLVM compiler," in *Proceedings of the Second International Conference on Runtime Verification*, ser. RV'11. Springer-Verlag, 2012, pp. 110–114.

[22] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[23] A. Jannesari and W. F. Tichy, "Library-independent data race detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, p. 1, 2013.

[24] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks," *The International Journal of Supercomputer Applications*, 1991.

[25] M. Andersch, B. Juurlink, and C. C. Chi, "A benchmark suite for evaluating parallel programming models," in *Proceedings 24th Workshop on Parallel Systems and Algorithms*, ser. PARS '11, 2011, pp. 7–17.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. ACM, 1995, pp. 24–36.

[27] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of Splash-2 and Parsec," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, ser. IISWC '09. IEEE Computer Society, 2009, pp. 86–97.

[28] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das, "Towards a communication characterization methodology for parallel applications," in *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, ser. HPCA '97. IEEE Computer Society, 1997, pp. 310–.