Double Inspection for Run-Time Loop Parallelization

Michael Philippsen[1,⋆], Nikolai Tillmann[2], and Daniel Brinkers[1]

[1] University of Erlangen-Nuremberg, Computer Science Dept., Programming Systems Group, Erlangen, Germany
{philippsen,daniel.brinkers}@cs.fau.de
[2] Microsoft Research, One Microsoft Way, Redmond, WA, USA
nikolait@microsoft.com

Abstract. The Inspector/Executor is well-known for parallelizing loops with irregular access patterns that cannot be analyzed statically. The downsides of existing inspectors are that it is hard to amortize their high run-time overheads by actually executing the loop in parallel, that they can only be applied to loops with dependencies that do not change during their execution and that they are often specifically designed for array codes and are in general not applicable in object oriented just-in-time compilation.

In this paper we present an inspector that inspects a loop *twice* to detect if it is fully parallelizable. It works for arbitrary memory access patterns, is conservative as it notices if changing data dependencies would cause errors in a potential parallel execution, and most importantly, as it is designed for current multicore architectures it is fast – despite of its double inspection effort: it pays off at its first use.

On benchmarks we can amortize the inspection overhead and outperform the sequential version from 2 or 3 cores onward.

1 Introduction

Just-in-time compiled object-oriented script languages like JavaScript [3] are getting important and heavily used in practice[1] and their use is no longer restricted to small, short-running or interactive applications. But they are not well-suited for the multicore future as, in general, they do not offer any means to express parallelism. Sequential JavaScript code runs in every web browser and will therefore face the fate of every sequential code in the foreseeable multicore future. But due to the dynamic typing and due to the fact that such codes in general are not the typical regular array-based codes known from scientific programs, well-known results from automatic parallelization of high-performance codes are hardly applicable. Hence, unless/until there is some way to express parallelism

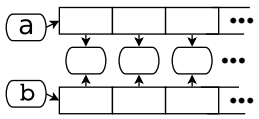
⋆ For this work the author has spent time at Microsoft Research during his sabbatical.

[1] As of March 2011, www.tiobe.com reports that about 35% of today's code is written in dynamically typed languages and that in the last five years about half of the most used programming languages have been scripting languages.

S. Rajopadhye and M. Mills Strout (Eds.): LCPC 2011, LNCS 7146, pp. 46–60, 2013.

in those languages, run-time parallelization is the only viable way to go – especially since the dynamic typing renders even fewer loops amenable to static parallelization than usual. Consider the following JavaScript example:

```
var a = [{v:1}, {v:1}, {v:1} ...];
var b = [a[0],  a[1],  a[2] ...];
for(i = 0; i < 10.000; i++){
   a[i].f = 2 * b[i].v;
}
```



Two complications in this (pseudo-)code cause traditional parallelizers to fail. The first problem is that `a[]` and `b[]` are arrays of references to objects that reside on the heap instead of arrays of primitive types. As `a[i]` and `b[j]` can refer to the same object parallelizers can no longer work on array indices but they must consider general memory addresses and data dependences that forbid parallel execution which are hard to detect by means of alias analysis. The second problem is that the code writes to the field `f` of the `a`-objects. Although there might be some objects that already have this field; for others the field is created on demand at the moment of the assignment.

In addition to work on privatization and on detection of reductions [12], most work on run-time parallelization is based on the inspector/executor idea of Saltz et al. [13]. Their idea is that in a first loop, an inspector performs a dry-run of the loop and looks at all array indices that the loop will touch – without actually executing the operations of the given loop. If certain addresses are touched by more than one iteration and not just for reading, then there is a data dependence that requires that these iterations are performed in their original order, i.e., not in parallel to each other. If there are no such dependencies, then the loop is fully parallelizable; otherwise the loop is partially parallelizable in so-called wave fronts. All those iterations of the original loop are scheduled to a single wave front that can be executed in parallel to each other as they do not have any cross-iteration dependencies to each other. Dependencies only exist to iterations that are scheduled to other wave fronts and that are hence separated by at least one synchronization barrier.

The general problem is that the dry-run takes time as the inspector must evaluate all addresses and must keep track of each of these addresses in a book-keeping data structure to detect potential dependencies. The longer this inspection takes, the more parallelism must be found so that the total time needed for both the inspector plus the subsequent executor (that executes all the loop's iterations in wave fronts) is still smaller than a pure sequential execution of the original loop. The runtime of the inspector is thus the most important aspect and the crucial stumbling block of the whole idea.

There are four responses to this challenge. The first is optimistic/speculative, i.e., to execute the loop in parallel and check whether data dependencies occurred. In that case some form of roll-back purges the wrong effects and restarts with the original sequential loop. (Check the literature for thread level speculation [14] or transactional memories [5] for solutions.) Note that the optimistic approach cannot avoid the book-keeping cost for detecting dependencies.

Book-keeping must still be fast. The second response is an amortization argument: the inspector/executor approach is only applied for loops with compute intensive bodies that do much work with few array elements. (In some early papers [6] the authors added a few (and at that times slow) trigonometric functions for the mere purpose of making their benchmarks look nice.) The third response is a second-degree amortization argument. If the data dependencies are not changed by the loop itself, then the inspector can be run once and its result can later be used for many parallel executions of the loop at hand. This works well, if for example the parallelized loop is buried in an outer loop. Unfortunately, such a so-called schedule reuse [10] often does not work in dynamically typed script languages or on real-world problems [7], since in general no static analysis can assure that the data dependencies will not change. Finally, the fourth countermeasure is to perform the inspection itself in parallel to save time. However, when two parallel inspectors try to register in a book-keeping data structure that a single address is being touched, this will – at least conceptually – require slow synchronization or critical sections to guard the tracking data structure. Thus, it is essential to avoid such synchronization wherever possible. This is even more difficult on modern multicores as their memory systems are typically not sequentially consistent and cores might see changes at different times. And even with the synchronization demands solved, the individual effort for registering every single address must be kept tiny and without branches to keep the processor pipelines busy.

The following section presents such an inspector that detects if a loop can be fully executed in parallel. The main idea is to inspect the loop twice, but with only a single synchronization barrier and memory fence in-between. Our inspector's book-keeping effort for registering every single address is tiny – just a few machine instructions with only one conditional branching instruction. It amortizes easily and its complexity is in the order of the complexity of the loop to be parallelized (instead of being in the order of the size of the main memory). We show-case the performance of double inspection in Sec. 3 before we have a quick look at the related work in Sec. 4 and conclude.

2 Double Inspection

2.1 Basic Idea – Overly Conservative

The main idea of the double inspection is to inspect the loop in two phases, each of which is done in parallel by a set of inspector threads. To ease understanding we discuss a simplified and overly conservative version of the double inspection approach first. This version cannot parallelize loops with loop-independent dependencies.[2] The two subsections that follow will remove this restriction and they will refine and optimize the idea.

[2] A loop-independent dependence is a data-flow dependence that already exists in the loop body even if the loop control structure is taken away.

We assume that the inspector is deterministic so that each phase will observe the same sequence of memory accesses. For a formal view on our inspector, we start with a set of definitions. Let $U(I)$ be the set of all memory addresses from which iteration I reads. (Note that we talk about addresses and not array indices.) $D(I)$ is the set of all memory addresses to which iteration I writes. $W(A)$ is the set of iterations that write to the memory address A.

Foreach iteration I of the loop do (maybe in parallel):

(1): If $\exists\ u \in U(I)$ with $W(u) \neq \emptyset$ then there is a flow or anti dependence. (This is overly conservative as we will discuss in Sec. 2.2.)

(2): If $\exists\ d \in D(I)$ with $|W(d)| \neq 1$ then there is an output dependence.

Note that it would be possible to find all dependencies of one iteration I by iterating through the corresponding W set for every address in $U(I)$ and $D(I)$.

We now reduce $W(A)$ to an arbitrarily selected $W'(A)$. Every $W'(A)$ holds only one arbitrary element of $W(A)$. We can use a similar algorithm with these sets. Step (1) stays unmodified, as we only check whether the set is empty and do not care about the elements or their number. Step (2) has to change slightly. As the size of the set $W'(A)$ is known to be one, we now check whether the element in $W'(d)$ is I:

Step (2'): If $\exists\ d \in D(I)$ with $W'(d) \neq I$ then there is an output dependence.

With the modified sets W' we are no longer able to find all dependencies of an iteration I because the information about some dependency may be discarded. But we are still able to detect, *if* there are any dependencies. If $W(d)$ has more than one element and $W'(d)$ consists of the element I, Step (2') will not detect the dependence for iteration I. But it will of course detect the dependence for all other elements in $W(d)$.

We represent all sets W' by means of an array `used`. Each `used[A]` in this array holds one element of the set $W(A)$. An empty set is represented by the special value null. Parallel inspector threads fill the array `used` in the first loop of the double inspection, the **pre-registration loop** and register all memory addresses to which the inspected loop writes. To do so, there is a copy of the original loop in which each of the writes is replaced by a macro that performs the book-keeping. After the writes are gone, dead code elimination purges almost everything expect for the macros (and for the relavent control structures).

In the running example, with thread-specific values for `lwb` and `upb`, the pre-registration loop of the inspector would therefore be:

```
for(i = lwb; i < upb; i++){
   preregister(hash(&a[i].f), i);  //only for writes
}
```

Note that instead of the real addresses, we use hashed versions to make sure that the book-keeping array stay small. The smaller it is, the more likely are false positives. In garbage-collected languages that may move around objects the "hashed" value needs to survive the collector's activities. Our current prototype ignores that and uses the hash function `(ptr<<3)&0xffff`.

The pseudo-code of the inspector's pre-register macro is:

```
int used[]; //shared array, used by all inspector threads
void preregister(int addr_hsh, int iteration){
   used[addr_hsh] = iteration;
}
```

Note, that at one point of time several inspector threads might write to the same slot of the book-keeping array `used`. This is fine and one of the main design principles of double inspection, as our only requirement is that the machine architecture makes sure that one of the threads will win, i.e., when all inspector threads have finished, one uncorrupted iteration number will be in the books. It is irrelevant that write operations will be buffered in the store buffer or in the caches of the individual cores. We only have to make sure that there is a single synchronization barrier with a memory fence after this first inspection pass.

After this first pass, the inspector threads perform their second pass over all the iterations. This time, in a **checking loop**, we check both for the reading and writing memory accesses whether they cause dependencies and hence render the original loop as not fully parallelizable.

Here is the checking loop for the running example (again, everything is removed except for the relevant control structures):

```
for(i = lwb; i < upb; i++){
   read(hash(&b[i].v), i)
   write(hash(&a[i].f), i);
}
```

The macros for checking are shown below.

```
void write(int addr_hsh, int iteration){
   if(used[addr_hsh] != iteration) alert(); //alert, if someone else wrote
}
void read(int addr_hsh, int iteration){
  if(used[addr_hsh]) alert(); //alert, if ever written
}
```

Let's look at the `write` first. If an inspector thread is the only one that has pre-registered that an iteration has written to a certain address, it will find in the second pass that the iteration number is still in the books, and everything is fine. If however, there are more than one iterations that write to an a address, at least one of the inspectors will detect a difference either because the thread itself has registered two iteration numbers or because its iteration number has been overwritten by another inspector thread. The asynchronous pre-registrations can happen without synchronization and in any order, since *at least one* of the inspector threads will find a different value in the second pass. Hence, output dependencies (= two writes to the same address by different iterations) will be detected – the `alert` will flag the loop as not fully parallelizable.

In the same way, the inspector that checks an address for reading will flag a dependence if that address has been pre-registered (i.e., it is written to) at all.

This basic double inspection is overly conservative, since even loops that only have loop-independent dependencies will be flagged as not fully parallelizable. For example, even if a loop iteration just reads a value and updates it without any interference from other iterations, this basic double inspector signals a

dependence as the updated address has already been pre-registered when the checking loop tests the read access and finds some non-null entry in the books.

In inspector/executor systems it is crucial that the inspector and the executor follow the same control paths. Being conservative helps, because data value dependent execution paths will be recognized.[3]

Let's make the inspector more aggressive now. Sec. 2.3 will avoid to re-initialize the book-keeping array for every inspection.

2.2 First Improvement: Tolerate Flow Dependences

The naive approach to extend the basic version to ignore loop-independent dependencies is to replace the null-check in the `read`-macro. This "improved" version of the macro would only flag a dependence if some *other* iteration had written to an address that is read in the current iteration.

```
void read(int addr_hsh, int iteration){
   if(used[addr_hsh] && (used[addr_hsh] != iteration)) alert();
}
```

Unfortunately, this is too simplistic. The problem are dependencies that are a result of the computations in the loop itself. Consider the following example:

```
int A[4] = {0,1,2,3}
for(int i=0; i<3; ++i){
   A[i] = i+1;
   A[A[i]] = i+1;
}
```

In this example, just registering and checking the addresses of `A[i]` and `A[A[i]]` will not detect any conflicts as `A[A[i]]` stays equal to `A[i]` because the inspector is a dry-run and does not execute the computations/assignments. The data dependences that prevent full parallelization in this example are however a result of these modifications. A flow dependence that reads a value and later contributes to an overwriting of it, is ok as the assignment is in a way the last thing that happens to that memory address. In contrast, an anti-dependence that writes a value first and later (re-)reads it (or something it depends on), might change dependencies. Thus, inspectors that are based on a dry-run can ignore loop-independent flow dependences but they have to be conservative and signal a potential threat to parallelizability upon a loop-independent anti-dependence.

The insight for an efficient implementation is that if there is a loop-carried dependence for a certain memory address, then the checking-loop will find a mismatch of iteration numbers at some point. If there are only loop-independent dependencies for that address, then *only one* inspector thread reads and modifies the corresponding slot of the book-keeping array. As no synchronization and memory consistency measures are needed to guard against interfering inspector threads in the checking loop, we can use a bit of this slot to detect the nature of a loop-independent dependence. Upon a write, this so-called flow bit is set. When

[3] We apply a pre-test to stay away from loops with conditional branches that depend on loop-external side effects.

a read finds the flow bit already set, there is an anti-dependence and hence a threat to full parallelizability. If the read precedes the write, the read finds the flow bit still un-set and the check remains quiet.

It is crucial to implement this efficiently. To do so, we (for now) use the least significant bit of the `used` integer values. The pre-registration registers `2*iteration` which shifts all bits to the left and lets the least significant flow bit un-set. In the checking loop, `write` ignores the flow bit in the comparison, but then sets it by assigning `2*iteration+1`. The comparison in `read` is sensitive to the flow bit. If the loop body has written to an address before, the `read`-macro will flag a parallelizability problem. Writes after reads are ok.

We expect the `iteration` numbers to be non-negative, fitting into 31 bits. A guard just before the inspector loop checks the lower and upper bounds to ensure that no overflows occur in the actual inspector.

```
void preregister(int addr_hsh, int iteration){
   used[addr_hsh] = 2 * iteration;
}
void write(int addr_hsh, int iteration){
   if((used[addr_hsh] | 1) != (2  * iteration + 1)) alert();
   used[addr_hsh] = iteration * 2 + 1;
}
void read(int addr_hsh, int iteration){
  if(used[addr_hsh] && (used[addr_hsh] != iteration * 2)) alert();
}
```

On our benchmarks it has improved the speed of book-keeping by about 2% on average that `write` does not compare `used[addr_hsh]&(~1)` to `2*iteration`. The reason is that the compiler can keep `2*iteration+1` in one iteration variable when unrolling the checking loop, whereas the bitmasking code would need two iteration variables.

2.3 Second Improvement: Avoid Re-initialization

Up to now the inspectors have to re-initialize the book-keeping array `used` for every loop that has to be inspected. To avoid re-initialization, we turn `used` into an array of 64 bit values. The upper 32 bits are used for a generation number plus the extra bit needed to tolerate flow dependencies. The generation counter `base` is incremented once for every loop to be inspected. Only after `MAX_INT/2` inspections, that is almost never, the `used`-array needs to be re-initialized.

```
long used[]; //shared array, used by all inspector threads
int base = 0;
void preregister(int addr_hsh, int iteration){
    used[addr_hsh] = (base<<33) + iteration;
}
void write(int addr_hsh, int iteration){
   if(((int32*)&used[addr_hsh])[0] != iteration) alert();
   ((int32*)&used[addr_hsh])[1] = 2 * base + 1;
}
void read(int addr_hsh, int iteration){
   if(((used[addr_hsh] - (base<<33)) ^ iteration) > 0) alert();
}
```

An optimization insight is that `write` does *not* have to check whether the value is from the correct generation, since *it has to be* since each value a `write` checks has been written by a corresponding `preregister`. Hence `write` only compares the lower 32 bit to `iteration` which automatically also ignores the flow bit. After the comparison the flow bit is set in the upper `int`.

Our x86-64 assembler code is fast and only needs five instructions, namely `lea` and `mov` to compute the hash value and to access the booked value, a `cmp`, a (short) conditional jump `jne`, and another `mov` for the assignment. A register holds `2*base+1` as it is fixed for the whole inspection.

A straightforward implementation of `read` would first check whether the value in the books is from the current generation and only then look at the iteration numbers. That would result in two conditional jumps per `read`. On current pipeline architectures with branch prediction this turns out to be too slow. It is crucial to avoid the jump on a generation mismatch. To do so, there is the `xor` in the `read`-operation. We first subtract the current `base`-value from the value in the book in the upper `int` (`base<<33` is a pre-computed fixed value that is kept in a register). If the value in the books originated from an earlier version, then the result is negative, which is ok, as there is no dependence. If the value is from the same generation but from a different iteration, then the result of the `xor` is positive. The value is also positive, if after subtracting `base` and `xor`-ing away the iteration number, only the flow bit remains switched on. Hence, with a simple subtraction and an `xor`, we merge the generation test, the iteration test, and the flow test into a single conditional jump. Moreover, in the case of x86-64, the result of the comparison to 0 is implicitly remembered in status bits after the `xor` operation, so that no explicit machine instruction is needed to perform the comparison. Thus, the `read`-macro also needs just five machine instructions (`lea`, `mov`, `sub`, `xor`, and `jg`).

As an alternative to generation numbers one could use two sets of book-keeping arrays. While one set is being used for inspection the other is cleaned. However, since current multicore processors do not yet have an abundance of cores, we need all of them so that inspection plus execution performs faster than the sequential loop.

3 Benchmarks

For the measurements we have used a single Intel Core i5 760 chip (4 cores) running at 2.80 GHz with 8 MB shared cache and 8 GB of RAM (DDR-1333). The machine runs Windows 7 x64. We have used the Visual Studio 2010 compiler.[4]

We choose four fully parallelizable benchmarks to test our inspector's performance. The loops under consideration are tiny, i.e., 3–5 lines of code long. We extracted them from the SunSpider JavaScript benchmarks, where the loops

[4] We have also performed measurements on an Intel Xeon X7560 "Nehalem-Ex" chip (8 cores + hyperthreading) running at 2.27 GHz with 24 MB shared cache and 512 GB of RAM (DDR3-1333). This machine runs Linux 2.6.32. We have used the GNU g++ 4.4.3 compiler. We found the same overall behavior and only show i5 times.

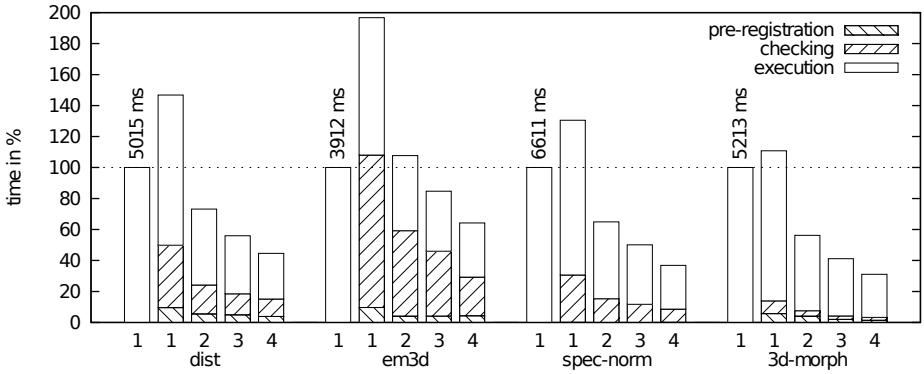


Fig. 1. Sequential performance and parallel performance with double inspection. Numbers show the runtimes of the parallelizable loop without framework setup on 1-4 cores.

have significant runtime which we determined with the SPUR tracing JIT [1]. We have implemented them in C/C++ since this made it easier to interact with our C-based inspection framework. As can be seen in Fig. 1, the pre-registration loop, the checking-loop, and the executor scale nicely and from two cores on outperform the sequential execution on a single core. (On em3d, we need 3 cores to see a speedup.) So even if the benchmark loops are only executed just once, double inspection and parallel execution are faster than executing the loops sequentially. The cost of the inspection is amortized immediately, i.e., without schedule and schedule reuse or speculation.

Dist calculates the distance between pairs of 2D points. What rules out static parallelization are the indirections: another array holds pointers to the 2D points that need to be used in the distance computation. This is similar to traversing a linked list of objects and applying a side-effect free method to each of the objects. Although this is parallelizable, there is the underlying container implementation whose links and pointers are only known at runtime. In Dist there are four read accesses and one write access per iteration. This explains the relation of the time spent in the pre-registration versus the checking loop.

Em3d propagates electro magnetic waves that are stored in a bipartite graph with many indirections. In our benchmarks we have a used an input graph that can be processed in parallel. Computing the indirections, i.e., the addresses of the data that is needed at run time is costly. While the sequential execution only needs to do this computation once, we (currently) re-compute the information in both inspector loops and in the executor. Hence, two cores are not sufficient to compensate for the extra cost – we only see speedups from three cores onward. The checking loop is so much slower here, since there are 20 reads for each write.

Spec-norm calculates the spectrum norm of a parameterized matrix with a matrix-vector product as the hot loop. We have included this benchmark to demonstrate that our technique is even applicable and achieves good performance on typical numeric problems. The loop in this benchmarks has 4865 reads for one write. Due to the normalization to the sequential execution time,

the cost of pre-registration cannot be seen in the graph. Execution takes longer than inspection, because many computations are irrelevant for the dry-run.

3d-morph is a 3d graphics algorithm that does (slow) trigonometric computations in its hot loop. This is another worst-case benchmark as a static analysis could have detected the parallelizability. In contrast to Spec-norm, there are only write operations. Hence pre-registration and checking take about the same time.

4 Related Work

A number of parallel inspectors have been developed in the past that use a critical section to guard the book-keeping data structures, for example [9,16]. To check whether the availability of a highly optimized atomic machine instruction on today's multicore machines (like xchg, cmpxchg, ...) helps those types of inspectors, we have designed and optimized such a basic inspector that also only uses a tiny number of machine instructions for the book-keeping. Whenever this basic inspector sees a read or a write of an address, it upgrades the state of a corresponding slot in a `used`-array by conceptually calling the inlined `read`- or `write`-macro given below in pseudo-code.[5] Again, instead of the real addresses, we use hashed versions to make sure that the book-keeping array remains small.

```
const int NO_ACCESS = 0;
const int READ      = 1;
const int WRITE     = 2;
int used[]; //shared array, used by all inspector threads
void read(int addr_hsh) {
   if(xchg(&used[addr_hsh], READ) == WRITE)) alert();
}
void write(int addr_hsh) {
   if(xchg(&used[addr_hsh], WRITE) != NO_ACCESS)) alert();
}
```

This inspector is also very conservative as it even signals a dependence if an address is touched more than once from the same iteration and not just for reading. An extended version of this inspector that funnels the iteration number into the slots of the `used`-array to avoid an alert if an iteration reads and later writes to the same address was significantly slower.

Again, we added a generation counter to avoid re-initialization cost and found a way to do the `read`- and `write`-macros with just a single conditional jump. Both macros are very light-weight, except for the atomic `xchg` instruction, which dominates execution time due to the necessary synchronization. We feel that it would be hard to construct a faster inspector that relies on a synchronous access to a book-keeping data structure. Hence, this `xchg`-algorithm is the most competitive representative of this line of related work.

[5] An `xchg` machine instruction (InterlockedExchange on Windows) atomically writes the second argument into the address given by the first argument. It also returns the value that has been at that address before the assignment.

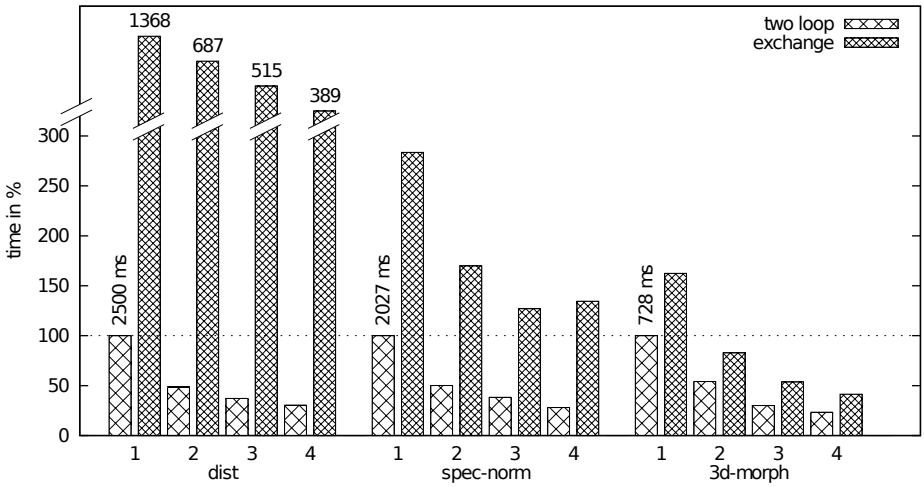


Fig. 2. Comparison to `xchg`-times. Times of the inspector only, taken on 1-4 cores. The runtime of the executor is not shown as it would be the same regardless of the type of inspector used.

```
int base = 0; //shared
void read(int addr_hsh) {
   if(xchg(&used[addr_hsh], READ) == base + WRITE)) alert();
}
void write(int addr_hsh) {
   if(xchg(&used[addr_hsh], WRITE) > base + NO_ACCESS)) alert();
}
```

These results shown in Fig. 2 demonstrate that even on today's multicore architectures such types of inspectors are (still?) not fast enough and that parallel inspectors need to be lock free, as our double inspection is. The benchmark em3d is missing in Fig. 2, since the `xchg`-algorithm is overly conservative, detects a loop independent flow dependence, and hence signals that the loop cannot be parallelized. On spec-norm the `xchg`-algorithm does not scale well (even on the 8 core chip). The bottleneck is the bus traffic needed to implement the locking.

Other inspectors try to actually compute wave fronts and deal with partially parallelizable loops, e.g. [11]. Leung and Zahorjan [8] have introduced the ideas of sectioning and bootstrapping and have demonstrated that these ideas help in speeding up parallel inspectors that compute wave fronts. But our double inspection deliberately stays away from computing wave fronts and just decides whether a loop is fully parallelizable or not. The reason is, that all the known wave front computing algorithms spend too much time in their book-keeping.

For a comparison, we have used sectioning to parallelize the wave front algorithm by Yang et al. [15]. This algorithm firstly is straightforward to parallelize, secondly requires fewer book-keeping data than other wave front algorithms, and finally can (in contrast to others) also handle all types of dependencies. We have

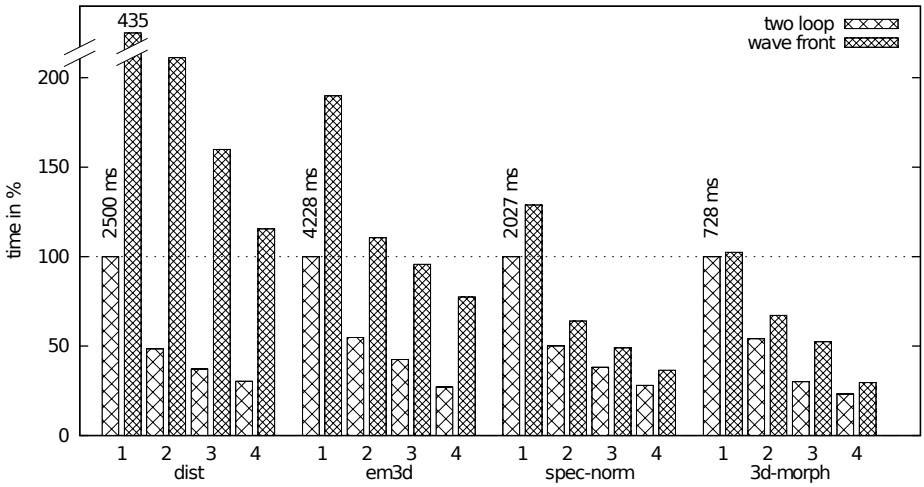


Fig. 3. Comparison to wave front algorithm. Times from taken on 1-4 cores.

optimized their algorithm by removing time consuming array dereferencing and indirection. This also allowed to avoid re-initialization of the book-keeping data structures. In addition, we have fixed a bug. The appendix gives the resulting pseudo-code of the fastest wave front algorithm we know.

Fig. 3 also shows that even with our optimizations, computing wave fronts is slow on those of our benchmarks that have indirect and more read accesses.[6] In addition to a slower inspection, execution suffers from extra synchronization barriers as the resulting wave fronts of all inspectors are concatenated. Even on fully parallelizable loops there is at least one synchronization barrier per thread. The result is that the time needed to compute the wave fronts cannot (easily) be amortized by parallel execution without schedule reuse or large loop bodies. Hence, our double inspection approach only checks for full parallelizability which can be done much faster and is universally applicable, even in dynamically typed codes that are hard to analyze for applicability of schedule reuse.

Other inspectors have used two phases before, e.g., [2,11]. In their first phase, the threads usually work on a private data structure. The second phase then merges the results from the first phase. Unfortunately, the merging phase is usually bounded by the sizes of the arrays, instead of being bounded by the complexity of the loop to be parallelized. For non-array codes that use heap objects, the merging phase would have a complexity that scales with the size of the main memory/the total address space. Moreover, the number p of threads often

[6] We are showing the best case for the wave front algorithm here. The code in the appendix can fail on certain loop-independent dependencies and is not as generally applicable as our double loop inspector. Moreover, for spec-norm and 3d-morph the access pattern is the best case for the wave front algorithm, as in contrast to the first two benchmarks, it causes almost no cache misses.

comes in as an additional factor (sometimes just as log p). Our double inspection is different because both its loops scale with the complexity of the original loop. The number of threads only affects the cost of the single synchronization barrier and does not come in as a multiplicative factor. Moreover, our book-keeping data structure does not need to be re-initialized between inspector runs. And it does not need to be scanned in its entirety in a merging phase.

Although our inspector is presented in this paper for use with a subsequent executor, it can also be used in speculative parallelism, as suggested in [4,12]. Since the book-keeping effort of our double inspection is tiny, it can easily be added to the speculative parallel execution without affecting its performance too much. To implement it, the speculative execution can either be piggy-backed to the pre-registration loop or to the checking loop of our inspector.

5 Conclusion

In this paper we have presented a fast parallel inspector that detects loops that can be executed in parallel. The novel idea of this inspector is that parallel threads inspect segments of the loop asynchronously. And instead of a costly merging phase, we have those threads inspect the loops again after a single synchronization barrier. The highly optimized implementation for the double inspection loops on current multicore architectures makes it possible to amortize the cost of inspection immediately – there is no longer any need for schedule reuse. On benchmarks we can amortize the inspection overhead and outperform the sequential version from 2 or 3 cores onward.

As a by-product, we also suggested optimizations for other types of inspectors that are known from the related work.

Future work should study the runtime overheads in a just-in-time engine. While one core could start executing a loop sequentially, the double inspection could be applied to the tail/the majority of the loop iterations. And if the tail turns out to be parallelizable, the JIT would switch. This could hide the inspection overhead. It will also be interesting to see if the technique can be extended to perform task-level parallelization of function calls or other code regions.

Acknowledgements. We thank Wolfram Schulte and Sebastian Burckhardt for many discussions and insights on contemporary multicore architectures, and especially Erez Petrank for an earlier investigation of the problem space.

Appendix: A Parallel Wave Front Inspector without any Shared Data Structures

With sectioning, we split the iteration space of a given loop into sections, each of which is handled by a single inspector thread that uses the algorithm of [15] with our enhancement and the bug fix. The resulting wave fronts of all threads

are concatenated for execution. Fully parallelizable loops that are inspected and executed with n threads need n synchronization barriers.

The arrays `def` and `use` store in which wave front a value has been written/read before.[7] The wave front of an iteration is 1 plus the maximum of all the wave fronts of all the addresses that the iteration touches. This maximum is computed by the `read` and `write` macros and is stored in `current_wf` after all addresses are touched. Before all the addresses that the iteration touches are checked again, `wf` is updated to keep the wave front of the current iteration.

A second pass through all the addresses updates `use` and `def` to reflect that the current iteration reads/writes them. The original algorithm sets the corresponding slot of the `use` array for every read to the current iteration number. This is a bug and can lead to wrong results. This update must happen, *iff* the wave front number corresponding to the `use` array entry is smaller than the wave front number of the current iteration, because when referring to the latest iteration that reads this address, the new execution order must be considered.

Per read and per write, this algorithm needs about twice as many machine instructions than our double inspection. And since each of the inspector threads only works on its private data structures there is no need to synchronize. The downside is that itis far from optimal to concatenate all the thread-local wave fronts. The executer pays for this.

```
int use[];        //private per inspector thread
int def[];        //private
int wf[];         //private
int base_wf;      //of inspection
int current_wf;   //of iteration
int max_wf;       //of inspection
void begin_iteration() {
    current_wf = base_wf;
}
void read(int addr_hsh) {
    current_wf = max(current_wf, def[addr_hsh]);
}
void write(int addr_hsh) {
    current_wf = max(current_wf, def[addr_hsh], use[addr_hsh]);
}
void between(int iteration){
    wf[iteration] = ++current_wf;
    max_wf = max(max_wf, current_wf);
}
void read_update(int addr_hsh){
     use[addr_hsh] = max(use[addr_hsh], current_wf);
}
void write_update(int addr_hsh){
    def[addr_hsh] = current_wf;
}
```

Due to lack of space, we cannot show an extension of this algorithm that reuses its book-keeping data structures instead of re-initializing it for each inspection.

[7] The original algorithm stored the iterations numbers in those arrays and used the `wf` array to look up the wave fronts for the iterations. We avoid this indirection.

Again, a generation number can be funneled into the values that are written to the arrays.

References

1. Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., Venter, H.: SPUR: a trace-based JIT compiler for CIL. In: Proc. OOPSLA 2010, ACM Intl. Conf. Object-Oriented Programming, Systems, Languages, and Applications, Reno, NV, pp. 708–725 (October 2010)
2. Chen, D.K., Torellas, J., Yew, P.C.: An efficient algorithm for the run-time parallelization of DOACROSS loops. In: Proc. ACM/IEEE Conf. Supercomp., Washington, DC, pp. 518–527 (November 1994)
3. Eich, B.: JavaScript at ten years. In: ACM SIGPLAN Intl. Conf. Functional Programming, keynote. Tallinn, Estonia (September 2005), `http://www.mozilla.org/js/language/ICFP-Keynote.ppt`
4. Gupta, M., Nim, R.: Techniques for speculative run-time parallelization of loops. In: Proc. ACM/IEEE Conf. Supercomp., Melbourne, Australia, pp. 1–12 (July 1998)
5. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proc. OOPSLA 2003, ACM Intl. Conf. Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, CA, pp. 388–402 (October 2003)
6. Kao, S.H., Yang, C.T., Tseng, S.S.: Run-time parallelization for loops. In: Proc. HICSS 1996, Hawaii Intl. Conf. System Sciences, Wailea, HI, vol. 1, pp. 233–242 (January 1996)
7. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. Comm. ACM 52(9), 89–97 (2009)
8. Leung, S.T., Zahorjan, J.: Improving the performance of runtime parallelization. In: Prof. PPoPP 1993, ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, San Diego, CA, pp. 83–91 (May 1993)
9. Midkiff, S.P., Padua, D.A.: Compiler algorithms for synchronization. IEEE Trans. Comput. 36(12), 1485–1495 (1987)
10. Ponnusamy, R., Saltz, J., Choudhary, A.: Runtime compilation techniques for data partitioning and communication schedule reuse. In: Proc. ACM/IEEE Conf. Supercomp., Portland, OR, pp. 361–370 (November 1993)
11. Rauchwerger, L., Amato, N.M., Padua, D.A.: A scalable method for run-time loop parallelization. Intl. J. Parallel Programming 26(6), 537–576 (1995)
12. Rauchwerger, L., Padua, D.A.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. IEEE Trans. Parallel and Distrib. Systems 10(2), 160–180 (1999)
13. Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Trans. Comput. 40(5), 603–612 (1991)
14. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: Proc. Intl. Symp. Computer Architecture, Vancouver, Canada, pp. 1–12 (June 2000)
15. Yang, C.T., Tseng, S.S., Kao, S.H., Hsieh, M.H., Jiang, M.F.: Run-time parallelization for partially parallel loops. In: Proc. Intl. Conf. Parallel and Distrib. Systems, Seoul, South Korea, pp. 308–313 (December 1997)
16. Zhu, C.Q., Yew, P.C.: A scheme to enforce data dependence on large multiprocessor systems. IEEE Trans. Softw. Eng. 13(6), 726–739 (1987)