Exploitation of GPUs for the Parallelisation of Probably Parallel Legacy Code

Zheng Wang[1], Daniel Powell[2], Björn Franke[2], and Michael O'Boyle[2]

[1] School of Computing and Communications, Lancaster University, United Kingdom
`z.wang@lancaster.ac.uk`
[2] School of Informatics, University of Edinburgh, United Kingdom
`d.c.powell@sms.ed.ac.uk, {bfranke,mob}@inf.ed.ac.uk`

Abstract General purpose GPUs provide massive compute power, but are notoriously difficult to program. In this paper we present a complete compilation strategy to exploit GPUs for the parallelisation of sequential legacy code. Using hybrid data dependence analysis combining static and dynamic information, our compiler automatically detects suitable parallelism and generates parallel OpenCl code from sequential programs. We exploit the fact that dependence profiling provides us with parallel loop candidates that are highly likely to be genuinely parallel, but cannot be statically proven so. For the efficient GPU parallelisation of those probably parallel loop candidates, we propose a novel software speculation scheme, which ensures correctness for the unlikely, yet possible case of dynamically detected dependence violations. Our scheme operates *in place* and supports speculative read and write operations. We demonstrate the effectiveness of our approach in detecting and exploiting parallelism using sequential codes from the NAS benchmark suite. We achieve an average speedup of 3.2x, and up to 99x, over the sequential baseline. On average, this is 1.42 times faster than state-of-the-art speculation schemes and corresponds to 99% of the performance level of a manual GPU implementation developed by independent expert programmers.

Keywords: GPU, OpenCL, Parallelization, Thread Level Speculation.

1 Introduction

GPUs have become ubiquitous in a wide range of computing devices and consumer electronics appliances. They provide a powerful resource for parallel processing and can deliver great performance improvements for suitably mapped algorithms. Realising this potential, however, is challenging due to the complexity of their programming.

Auto-parallelisation technology can greatly reduce the barrier for GPU programming by automatically generating parallel code from sequential programs. However, one of the main problems is the static undecidability of the underlying data dependence problem [9]. Static analysis attempts to determine if two memory references are dependent, in which case their sequential order needs to be retrained for correctness, limiting the amount of parallelism, which can be exploited. Static analysis is necessarily conservative and despite large research efforts, frequently fails to deliver, e.g. for complex, pointer-based C code [25].

Off-line profile-guided parallelisation is a recent development, which seeks to complement static analysis with profiling information [8,7,24,30]. Using such a scheme, a program is profiled with different input data sets and dependencies are determined using dynamic memory traces. Although correctness cannot be guaranteed, given enough input data sets, the probability of correctly identifying a genuinely parallel loop increases. In this paper, we seek to exploit such probably parallel loops. In addition, we want to avoid generating potentially unsafe code or asking the user for final approval. This means, we have to rely on speculative parallelisation [19].

Current speculation schemes are designed to deal with the occasional dependence violation and, consequently, provide efficient rollback capabilities. In our case, we can rely on profiling information and we only attempt to speculatively parallelise loops, where there is almost no chance of misspeculation. We require speculation support purely as a safety net, which might not be used at all. Hence, we can afford a more expensive rollback mechanism in favour of faster checks.

In this paper we combine profile-guided parallelisation, OpenCl code generation and software thread level speculation (Sw-Tls) to exploit highly-likely parallelism on the Gpu. Our compiler uses static and profile-based dynamic dependence analysis to detect parallelism and to automatically generate parallel OpenCl code with in place dependence checking. We exploit that parallel loop candidates are "almost always" genuinely parallel, but escape static analysis.

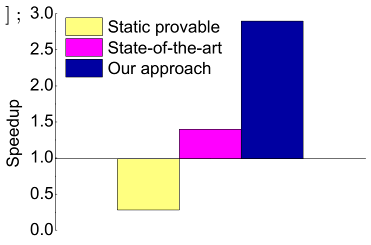
To provide safety we concurrently execute a sequential version of the program alongside our speculatively parallelised one. In the unlikely case of a dependence violation we abort parallel execution and rely on the results of the sequential program. This simple mechanism enables us to design a simple, yet efficient dependence checking mechanism for Gpus while at the same time, providing correctness for speculative parallel executions.

We have implemented our scheme using the Llvm compiler framework and have evaluated its effectiveness in detecting and exploiting parallelism in benchmarks, which are known to be manually parallelisable, but present a challenge to automatic parallelisation approaches. On an Nvidia Gpu platform, our approach achieves an average speedup of 3.2x (up to 99x), which is 1.42 times faster than its nearest competitor and delivers 99% of the performance level of a manual Gpu implementation.

2 Motivation

Consider the code fragment in figure 1 (a). This loop is extracted from the sequential version of the *BT* benchmark from the Nas benchmark suite. While conservative, static analysis fails to parallelise this loop due to the inter-procedural call to function `binvcrhs` at line 14 where an output dependence (i.e. write after write) to array `lhs` has to be assumed (inlining of `binvcrhs` would not eliminate the possible aliasing problem). Without further information, this loop would

```
void binvcrhs(double lhs[5][5],
     double c[5][5], double r[5])
{
  ...
  lhs[1][1]=lhs[1][1] - coeff*lhs[0][1];
    c[1][1] = c[1][1] - coeff*c[0][1];
  ...
}
...
void y_solve_cell() {
  ...
  for(j=1;j<grid_points[1]-1;j++){
    for(k=1;k<grid_points[2]-1;k++){
       binvcrhs(lhs[i][0][k][BB],
                lhs[i][0][k][CC],
                rhs[i][0][k]);
    }
  }
}
```



(a) source code of an example loop **(b) speedups obtained for the program**

Fig. 1. An example that static analysis fails to discover parallelism. No speedups were observed by only exploring statically provable parallelism. Profiling-based analysis, on the other hand, can provide us with additional information: no dependencies have been encountered in any trial run. By exploiting this information, we can use the GPU to execute both statically and probably parallel loops (with speculation support) and to achieve speedups rather than a slowdown. Our approach gives a speedup of 2.9x which is 2 times faster than a speedup of 1.45x given by the state-of-the-art GPU speculation scheme.

have to be executed in sequential on the Cpu (as it is too expensive to do so on the Gpu). Although we can still execute statically provable, parallel parts of the loops on the Gpu, we will have to introduce additional synchronisation and communication between the sequential Cpu and parallel Gpu computation. The additional overhead, however, could be expensive and can outweigh the benefit of parallel Gpu execution. In fact, as can be seen from figure 1 (b), doing so leads to a slowdown of 3.6x over the sequential code on a NVIDIA GTX 580 platform described in section 6.

Profile-based dependence analysis, on the other hand, provides use with the additional information that *no actual data dependence* inhibits parallelization for given sample inputs. While we still cannot prove absence of data dependences for *every possible input*, we can classify this loop as a highly-likely parallel candidate. We can then speculatively execute this loop in parallel on the Gpu with dependence violation checking together with a rollback scheme to ensure correctness if a true dependence violation is discovered at runtime. This is safe and potentially fast. As shown in figure 1 (b), a state-of-the-art Gpu speculation scheme, Paragon [21], gives a speedup of 1.45x for this particular benchmark. Though the result of using Paragon is encouraging, it can be further improved. Paragon requires a large buffer to record the speculative accessing addresses, which will be used in a separate dependence checking procedure to check the

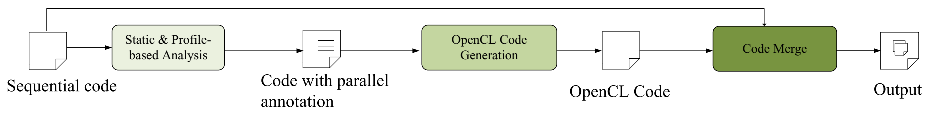


Fig. 2. Our compiler framework first uses static and profiled-based analysis to identify parallel candidates. Those parallel candidates are then translated into OpenCL kernels. Dependence checking code is added to perform dependence checking for those candidates that cannot be statically proven to be parallelizable but no dependence violation was discovered during profiling. Finally, the generated parallel OpenCL program is merged with the original sequential program as output.

potential violations of speculative accesses. This, however, can result in expensive indirect memory accessing overhead on the GPU. We would like to avoid this overhead.

As described later in this paper, our novel *in-place* dependence checking approach does not require a buffer to store the speculative accesses. It results in a speedup of 2.9x, two times faster than Paragon. With a novel dependence checking scheme, we then build a compiler framework to automatically generate parallel OpenCL code from sequential code using dependence profiling information and without user interaction, allowing us to exploit GPU parallelism for highly-likely parallel legacy code.

This example demonstrates that static analysis is overly conservative. Profiling based analysis, by contrast, opens up opportunities to exploit GPU parallelism for highly-likely parallel code.In the following three sections, we will first provide an overview of our compiler framework and then describe our parallelism detection and speculation schemes in details.

3 Overview

Our compiler uses both static and profile-driven dynamic analyses to automatically discover parallelism from sequential code and to generate parallel OpenCL code. For this, we also perform loop and array layout optimisations. At runtime, a safety net is provided for probably parallel loops that require dependence violation checking. Our prototype compiler is implemented using LLVM.

3.1 Compile Time

Figure 2 depicts our compilation framework. Our compiler uses three steps to generate parallel GPU code: parallelism detection, OpenCL code generation and code merging.

Parallelism Detection. We currently target loop-level parallelism. In particular, we use static analysis to separate definitely sequential and definitely parallel loops from other loops, which may or may not be parallel. For these possibly parallel loops we rely on dependence profiling [24,26] to extract those loops, which are *probably parallel*. We mark a loop as probably parallel if no cross-iteration dependences have been observed during any profiled execution using different data inputs. These loops are candidates for speculative parallel execution. The output of this stage is a program with OpenMp-like annotations to parallel and probably parallel loops, which include privatisable variables.

OpenCL Code Generation. The annotated program is passed to an OpenCl code generator [5], which automatically converts data-parallel loops and parallel reduction loops into OpenCl kernels. Each data-parallel loop is translated to a separate kernel using the OpenCl Apis, where each iterator of the loop is replaced by a global work-item Id. Checking code is added to speculative references, which may lead to a dependence violation in probably parallel loops. The details of our speculative checking scheme are described in section 5. Furthermore, as the currently OpencCL implementation does not support I/O operations, our approach does not speculatively parallelize any loops with I/O operations.

Code Merging. The last compilation stage merges the generated parallel OpenCl code with the original, sequential program into a single program. As such, the output program consists of both the original, safe implementation in addition to the generated OpenCl parallel code. Additional code will be automatically generated to spawn two processes to run both versions and validate results at runtime with the support a lightweight library.

3.2 Runtime

The combined use of static analysis and dependence profiling provides us with sufficient confidence that no data dependences exist in probably parallel loops, although this cannot be proven. The low expected probability of encountering any future dependences motivates us to speculatively execute such loops in parallel, without provisions for rollback to an earlier, safe state. Instead, we speed up what we expect to be the common case, i.e. parallel execution without dependence violation. In particular, we do not maintain rollback state or memory write buffers. Obviously, such as scheme will make the occurrence of a data dependence expensive to resolve, however, we do not expect this to happen frequently.

Runtime Dependence Checking. Inspired by a Cpu-based Sw-Tls scheme [16], we propose a *in place* dependence checking scheme for Gpus. Checking only needs to be applied to speculative memory references in probably parallel loops. Statically provable parallel loops do not require any runtime checking at all. For every access to a speculative variable (i.e. a variable of which a read and write access may cause an dependence violation with speculative parallel execution), our compiler automatically converts the memory reference to a speculative read/write operation. Dependences are checked *in place* and on the fly, and any

```
void binvcrhs_spec(__global double (*lhs)[5],
                   __global int (*rd_log_lhs)[5],
                   __global int (*wr_log_lhs)[5],
                   ...,
                   __global int* spec_flag,
                   __global int iter_id)
{
  ...
  _rval_0 = specLD_double(&lhs[1][1],
                          &wr_log_lhs[1][1],&rd_log_lhs[1][1],
                          iter_id,spec_flag);

  _rval_1 = specLD_double(&lhs[0][1], ...);

  //speculatively store the result to lhs[1][1]
  specST_double((_rval_0-coeff*_rval_1), &lhs[1][1],
                &wr_log_lhs[1][1], &rd_log_lhs[1][1],
                iter_id,spec_flag);
  ...
}

__kernel void y_solve_cell_L0 (...)
{
    ...
    iter_id = get_global_id(1) * get_global_size(0)
            + get_global_id(0) + init_iter_num;
    ...
    binvcrhs_spec(lhs, rd_log_lhs, wr_log_lhs,
                  lhs, rd_log_lhs, wr_log_lhs,
                  rhs, rd_log_lhs, wr_log_lhs,
                  spc_flag, iter_id);
}
```

Fig. 3. A simplified OpenCL-based code for the statically undecidable parallel loop shown in figure 1. A speculative version of the original function `binvcrhs` is generated in which every access to the speculative variable `lhs` is replaced with a speculative load/store operation.

violation will be reported to the control thread on the CPU. An example of the generated code can be found in figure 3, where reads and writes to the speculative variable `lhs` are replaced with a speculative load and store operations, respectively.

Recovery from Dependence Violations. We use competitive scheduling to deal with unexpected, but possible dependence violations. For this, we launch *both* the parallel and the original, sequential program simultaneously. Each version runs as a separated process which has its own memory space. We immediately terminate the parallel version on detection of a dependence violation. Otherwise, if no dependence violations have been observed, the version first to finish kills the slower competitor. The speculative execution will only commit if no violation is detected through all speculative execution. Maximum execution time is capped to time of sequential execution.

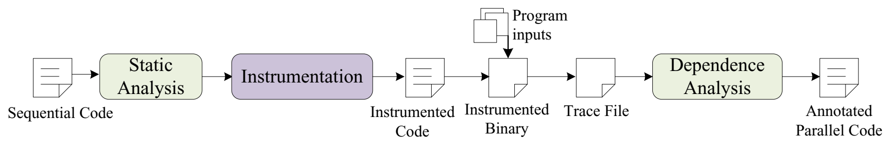


Fig. 4. The process of profile-based dependence analysis. Our compiler only uses profile-guided analysis for code regions where static analysis has bailed out.

4 Compile Time: Parallelism Detection and Code Generation

4.1 Parallelism Detection

To determine whether or not speculate we use the following hybrid approach: (i) use static analysis wherever possible and results are conclusive, (ii) use profile-guided analysis only for dependence checking where static analysis has bailed out, and (iii) identify parallel loop candidates using combined static and dynamic dependence information.

Figure 4 illustrates our hybrid static and dynamic parallelism detection approach. We use a customised memory dependence analysis path from Llvm v3.4 for static analysis. We then perform profile-guided analysis with similar capabilities as [24], but we only instrument memory operations, which previous static analysis could not resolve with certainty. The instrumented sequential application is recompiled and executed with several different inputs in sequential to generate traces of memory operations. Different program inputs are provided by the user. Each loop will be profiled once during trace collection. Loop traces are further analysed to determine if data dependences occurred during execution. Any loop that does not contain cross-iteration data dependences is then marked as *probably parallel*. Additionally, traces can be used to support static reduction recognition.

Speculative Variables. Tracking of speculative memory accesses is expensive, hence it is desirable that we only track those accesses that can potentially cause a dependence violation. Here we rely on static analysis to generate a list of variables that require speculative tracking, i.e. those which are subject to *may*-dependences. In particular, we do *not* track the accesses to read-only and thread-private variables. For the remaining speculative accesses we insert suitable wrappers, which invoke the appropriate checking functions.

4.2 Code Generation

Definitely parallel and probably parallel loops are treated similarly except probably parallel loops have references to arrays replaced with speculative loads and stores. Parallel loops are translated in a straightforward manner into kernels. A standard two-stage algorithm [3] is used to translate a parallel reduction loop. Each parallel loop is translated to a separate kernel using the OpenCl APIs where each iterator is replaced by a global work-item ID.

```
 1  double specLD_double(__global double *a, __global int *wr_log,
 2        __global int *rd_log, int iter_id, __global int *flag)
 3  {
 4    double value;
 5    atom_max(rd_log, iter_id);
 6    value = a[0];
 7    if (*wr_log > iter_id) /*Condition 1*/
 8          *flag = FAIL;
 9    return value;
10  }
11
12  double specST_double(__global double *a, __global int *wr_log,
13      __global int *rd_log, int iter_id, __global int *flag, double value)
14  {
15     atom_max(wr_log, iter_id);
16     if (*wr_log > iter_id) { /*Condition 2*/
17         *flag = FAIL;
18     }
19     a[0] = value;
20     if (*rd_log > iter_id) { /*Condition 3*/
21         *flag = FAIL;
22     }
23     return value;
24  }
```

Fig. 5. The OpenCL implementation of our speculative load and store. Dependence checking is combined with speculative loads and stores.

5 Runtime: Safe Speculative Execution

5.1 Runtime Dependence Checking

Dependence checking is combined with speculative loads and stores. Hence, we only need to check dependence violations for addresses that are actually accessed at runtime. Figure 5 shows the OpenCl implementation of speculative load and store operations. Dependence checking is performed *in place*. For each speculatively accesses address, we create a suitable entry in either a read or write log, i.e. the `rd_log` and `wr_log` variables in figure 5. The read and write logs are created on the GPU global memory, which are used to store the Id of the highest iteration that has read/written to the corresponding memory address `a` (lines 5 and 15). As OpenCl does not support barriers for Gpu threads across work groups, we use the `atom_max` operation provided by OpenCl to make sure only the *highest* iteration Id is stored in the log. The value in the log entry will be monotonically increasing[1] over time. Using the logs, we can simply determine whether a speculative load/store is successful.

5.2 Violation Detection

Speculative Load. A speculative load is successful if there have been no speculative store to the same memory location by a Gpu thread that executes a

[1] For a program with multiple Gpu kernels, the iteration Id passed to the speculative load and store functions starts from the maximum iteration number of the previous probably parallel loop. Therefore, the number is monotonically increasing for multiple speculative kernels.

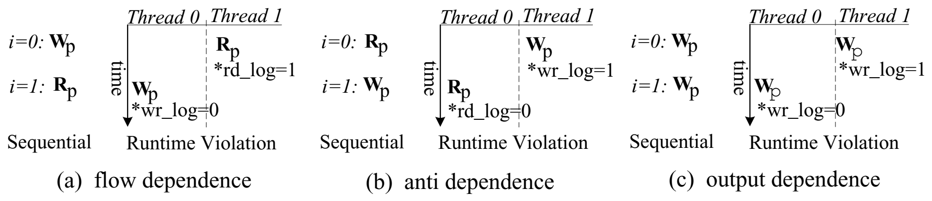


Fig. 6. Three cross-iteration dependence and the possible runtime violations due to GPU thread scheduling. All the three violations can be successful detected by our dependence checking scheme with the read (`rd_log`) and write (`wr_log`) buffers as shown in figure 5.

later loop iteration. This condition is checked in line 7. If the memory location is written in a later iteration, i.e. `(*wr_log>iter_id)`, a violation will be reported (line 8).

Speculative Store. Conversely, a speculative store is successful as long as there has been no speculative accesses (either loads or stores) to the same address by later iterations. This condition is checked in lines 16 and 18. If a later iteration attempts to write to the same location, i.e., `(*wr_log>iter_id)`, or read from it, i.e., `(*rd_log>iter_id)`, a violation is detected.

We continue the discussion of our violation detection mechanism for all possible types of runtime dependence violations. It is worth noting that our scheme is *exact* and does not report any false positives. In addition, if cross-iteration dependent accesses are executed in the correct sequential order by virtue of the GPU thread scheduler, it will correctly handle this situation and not flag any violation.

Our OPENCL code generator maps each loop iteration to an OPENCL work item to be executed by one GPU thread. Hence, no dependence violations are possible within one iteration. Though, cross-iteration dependence violations are possible due to the arbitrary order of thread scheduling on the GPU. In this case, Figure 6 enumerates all three possible cross-iteration violations. Here we show the sequential dependence of two consecutive iterations that must be respected and the potential violation due to GPU thread scheduling.

Flow Dependence. Figure 6(a) illustrates a violation of a flow dependence (i.e. read after write), where the use of `p` in iteration 1 happens before `p` is updated by thread 0, which executes iteration 0. This violation will be detected in function `specST`. It is `*rd_log=1` and `iter_id=0` and also Condition 3 (line 18) of figure 5 holds, such that a violation will be reported.

Anti Dependence. In figure 6(b), the use of `p` happens after it has been updated by the a later iteration. This causes an anti-dependence (i.e. write after read) violation, which will be captured by function `specLD`. In this case, it is `*wr_log=1` and `iter_id=0` and Condition 1 (line 7) of figure 5 holds, such that a violation will be reported.

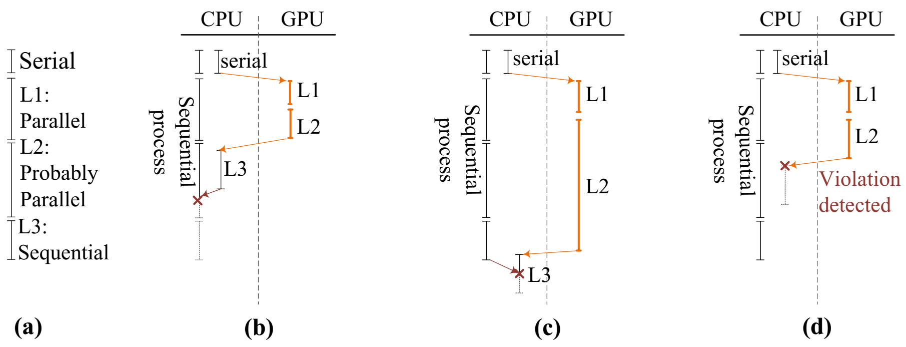


Fig. 7. Three different parallel execution scenarios for the sequential program shown in (a) : speculative execution runs faster with no conflict (b) , sequential execution runs faster (c), violation are found for speculative execution (d)

Output Dependence. Figure 6(c) is an output dependence (i.e. write after write) violation. After thread 1 has updated `p`, this memory location is overwritten by thread 0, which executes a previous iteration. In this case, it is `*wr_log=1` and `iter_id=0` and Condition 2 (line 16) in figure 5 holds, such that a violation will be reported.

5.3 Recovery from Dependence Violations

Speculative parallel executions can fail despite prior dependence profiling. We use a *competitive scheduling* scheme where we simultaneously execute a sequential version of the program alongside the parallelised program on a spare core of the host Cpu. If a dependence violation is reported, we simply abort speculative parallel execution and use the result produced by the safe, sequential run as the output of the program. Competitive scheduling caps the maximum execution time to that of the sequential program.

Figure 7 depicts our competitive scheduling scheme. This example contains three loops: a statically proven parallel loop $L1$, a probably parallel loop $L2$, and a statically proven sequential loop $L3$. In our scheme, loops $L1$ and $L2$ will be executed on the Gpu and the sequential loop $L3$ will be executed on the host Cpu. There are three possible scenarios. If the speculative version finishes first and does not observe any dependence violations, it terminates the sequential version (figure 7(b)). If the sequential version finishes first, it will abort the parallel speculative version (figure 7(c)). Finally, if the speculative version detects a dependence violation, it aborts and the sequential version will eventually finish (figure 7(d)) successfully.

5.4 Comparison to other Approaches

Our speculative checking scheme has several advantages when compared to other state-of-the-art Gpu thread level speculative schemes, e.g. Paragon [21]. Unlike

Table 1. Hardware platform

	Intel CPU	**NVIDIA GPU**
Model	Core i7	GTX 580
Core Clock	3.6 GHz	1544 MHz
Core Count	6 (12 w/HT)	512
Memory	12 GB	1.5 GB
Peak Performance	122 GFLOPS	1581 GFLOPS

Paragon, our scheme does not explicitly record addresses of speculative memory accesses. It is an integral part of the speculative accesses and perform checking on the fly. As such, our scheme does not have the indirect memory access overhead resulting from the address bookkeeping buffer, a problem which hampers Paragon's performance. Our scheme is particularly well suited for sparse data applications (e.g. using sparse matrices) where only a small number of the total index space is accessed by the program. Unlike Paragon, load and store logs (i.e `rd_log` and `wr_log`) can be re-used between multiple speculative kernels without the need for clearing them in-between. Finally, Paragon uses a naive violation detection scheme where an output dependence violation will be reported if there is more than one write to the same memory address. This naive scheme may cause false positives (i.e. a successful speculative execution is reported as violation) when an address has been updated multiple times within the same loop iteration or a write dependence is honoured. By contrast, our precise violation detection scheme is *exact* and does not suffer from this problem.

6 Experimental Setup

Platform. We evaluate our approach on a CPU-GPU mixed system with an Intel Core i7 CPU and an NVIDIA GTX 580 GPU. The system runs with a openSUSE 12.3 with Linux kernel 3.7.10. Table 1 gives detailed information of our platform.

Benchmarks. We have used the *sequential* NAS benchmark v.2.3 suite for which manually parallelised CPU and GPU implementations are available. To parallelise the code, we use a profiling-based auto-parallelisation tool to analyze data dependences and generate parallel OpenCL code. The tool parallelises loops with speculative checking, which are found to be parallelisable during profiling but cannot proven statically. For all loops that can be statically proven to be safe to parallelise, the tool parallelises them straightforward. The compiler parallelises up to three-level of a nested loops to create as many GPU threads as possible. Whenever possible, we try to avoid the CPU-GPU communications and synchronisation by running a parallel loop on the GPU. We avoid to parallelize a loop that accounts for less than 1% of the whole-program execution time unless there is a consecutive parallel or probably parallel loop candidate after it (so that we can remove a CPU-GPU synchronisation point).

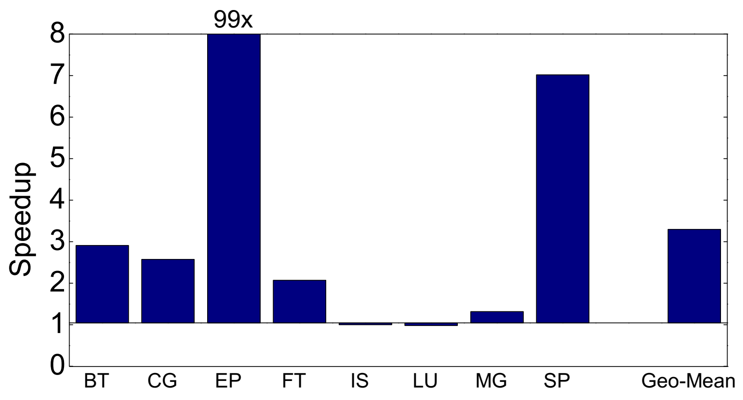


Fig. 8. Speedups over the sequential execution of our approach. We achieve on average a speedup of 3.2x and has never significantly slowed down the program over the sequential execution.

Compiler and Evaluation Runs. All programs have been compiled using GCC 4.4.7 with the -O3 option. Each experiment was repeated 5 times and the average execution time was recorded. All the benchmarks were profiled using the smallest input (class S) and evaluated with a larger input class (class A).

Comparison. Our approach is evaluated against *Paragon* [21], the closest competitor. In Paragon, probably parallel loops are discovered at program runtime by profiling those statically undecidable loops. However, we found doing so is very expensive. To provide a fair comparison, we make offload the profiling stage offline and provide Paragon with the same probably parallel code so it is speculate on exactly the same loops as our approach. We therefore only evaluate the efficiency of speculation rather than accuracy of parallelism discovery and profiling overhead. The Paragon scheme relies on OpenCL code generation. Again, we use the same OpenCL code generator to provide a fair evaluation. In addition to Paragon, we also compare our approach to two manually parallelised implementations of the NAS benchmark suite: an OPENMP version and an OPENCL implementation (SNU NPB [22]). Both versions were implemented by independent programmers. The two manual implementations provide a good estimation of the upper bound performance with the help of user assistance.

7 Experimental Results

In this section we first evaluate our approach against the sequential baseline. We then compare our approach to a scheme that only parallelises statically decidable loops on GPUS. This is followed by comparisons to a state-of-the-art GPU speculation scheme and manually parallelised implementations. Finally, we take a closer look at the limitations of static analysis and our speculation overhead, and discuss of dependence violations.

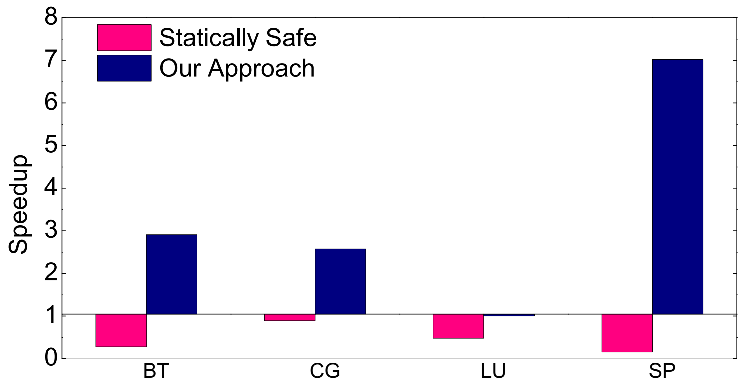


Fig. 9. Comparisons of Paragon and our in-place Gpu speculation scheme. Our scheme achieves higher speedups on more benchmarks when compared to Paragon.

7.1 Overall Results

Figure 8 shows the speedups achieved by our scheme. The performance numbers presented are speedups over the sequential execution on the Cpu. On average our scheme achieves a speedup of 3.2x. Furthermore, by co-running the original sequential program alongside the parallelised Gpu program, our scheme has never significantly slowed down the program.

As can be seen from figure 8, great performance improvement can be observed by exploiting Gpu parallelism for probably parallel loops. This is exemplified by the embarrassing parallel benchmark EP where a speedup of 99x was observed. Parallel Gpu execution can benefit for other benchmarks too. For benchmarks BT, CG and SP, we achieved a speedup of at least 2.6x and up to 7x. For benchmarks FT and MG, we only achieved modest speedups due to the available parallelism and cost of speculation. For benchmarks LU and IS, no speedups were observed on our platform. For LU, a new algorithm is required to get improved performance on the Gpu [22,5]. For IS, the parallel loop only accounts for 27% of the sequential execution and it is not worth to parallelise it on the Gpu. Nonetheless, our competitive scheduling scheme caps the execution time to the time of the sequential run if the parallel Gpu execution is not profitable.

7.2 Comparison with the Statically Safe Approach

We compare our approach to a conservative approach that only parallelises those statically proved parallel loops on the Gpu and runs the rest part in sequential on the Cpu. Obviously, no speculation is needed for such a scheme but data transfers and synchronisation are required to synchronise between the Cpu and the Gpu threads.

Figure 9 compare our approach with such a statically safe scheme. Here, some of the benchmarks are omitted because static analysis fails to discover parallelism

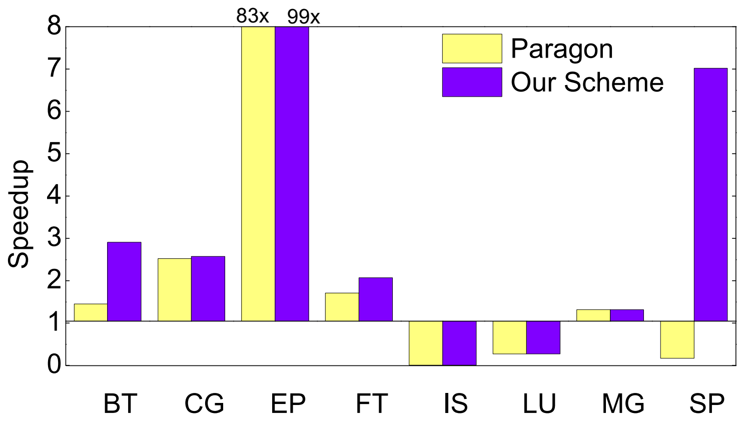


Fig. 10. Comparison of Paragon and our in-place Gpu speculation scheme. Our approach achieves higher speedups on more benchmarks when compared to Paragon.

of them. As can be seen from this figure, no speedups were observed for the conservative, safe scheme. This is due to the communication and synchronisation overhead associated with the switch between the Cpu and Gpu executions, where shared variables have to be synchronized among the two devices. This comes at the cost of expensive communications and synchronisation which outweigh the benefit of Gpu parallel executions. Our approach, by contrast, avoids this overhead by running two consecutive static and probably parallel loops on the Gpu so that we can keep the data on the Gpu and avoid the otherwise required Cpu-Gpu data transfers. Unlike the disappointing results of the static scheme, our profiled-based, Gpu speculation scheme is able to achieve speedups for all the four programs except LU where a change of algorithms is require to achieve speedups on the Gpu [22].

Overall, the static parallelisation technology is too conservative to exploit Gpu parallelism despite the abundant available parallelism for the majority benchmarks. By contrast, our approach outperforms the static parallelisation approach by a factor of 7.

7.3 Comparison with Paragon

Figure 10 compares our Gpu speculation scheme with Paragon. We factor out the performance achieved by co-running of the sequential code and focus solely on the quality of the Gpu speculation scheme. Note that we applied the same OpenCl code optimization to both approaches; therefore, the performance variations are mainly down to the difference of the speculation schemes.

This figure clearly demonstrates the advantages of our approach. As can be seen from this diagram, the overhead of Paragon can be significant for some benchmarks. For example, Paragon is not able to achieve speedups for SP while our approach gives a speedup of over 7x. For this benchmark, the indirect

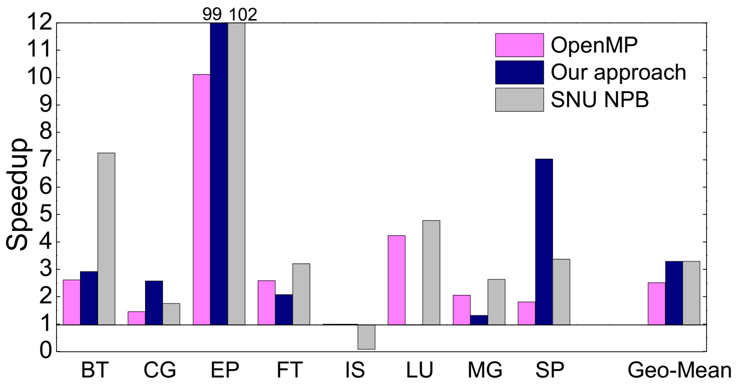


Fig. 11. Performance of the manual OpenMP and OpenCl implementation of the Nas benchmark suite and our automatically generated parallelised code

memory accessing and initialization overhead of Paragon clearly outweighs the benefit of Gpu parallel execution. Besides SP, our scheme also outperforms Paragon on benchmarks BT and FT, with a speedup up to 2 times higher. For benchmarks CG, EP and MG, speculative checking only needs to be performed on a few speculative variables and both approaches deliver similar performance. Finally, for benchmarks IS and LU, none of the two schemes achieve performance improvement due to the restriction of the program and the Gpu architecture as explained in section 7.1. Overall, our scheme outperforms Paragon by achieving higher speedups whenever it is profitable to exploit Gpu parallelism.

7.4 Comparison to Manually Parallelized Code

We also compare our approach to two manually parallelised implementations developed by independent programmers: (1) the OpenMP version of the Nas benchmark suite [1] for the Cpu and (2) SNU NPB [22], an OpenCl implementation of the Nas benchmark suite for the Gpu. The SNU NPB provides a good estimation of the up-bound performance that our Gpu speculation scheme can achieve. The results are shown in figure 11.

As can be seen from this diagram, exploiting Gpu parallelism for highly-likely parallel code can be beneficial. Example benchmarks include BT, CG, EP and SP where Gpu execution significantly outperforms the OpenMP Cpu execution by a factor up to 10. It is not supervised that a manually parallelised Gpu implementation without speculation overhead outperforms our automatic scheme, but our approach is able to achieve a level of performance close to the manual implementation. For benchmarks CG and SP, our approach even outperforms the manual Gpu implementation with advanced Gpu memory optimizations such as dynamic index reordering applied by our OpenCl code translator [5]. For benchmarks FT and MG, our approach is not as good as the OpenMP implementation. This is restricted by the programs themselves as the Cpu-Gpu communications

Table 2. Numbers of statically decidable and undecidable parallel loops of the manual OpenMP implementation

Benchmark	Manual	Statically Decidable	Statically Undecidable
BT	54	23	31
CG	19	17	2
EP	1	0	1
FT	6	0	6
IS	1	0	1
LU	29	12	17
MG	12	5	7
SP	70	41	29

is relatively high compared to computation. This can be seen from the fact that the manually parallelised Gpu code only outperforms the OpenMP Cpu code by a small margin. For benchmark LU, the algorithm in the sequential code has to be changed to a hyperplane one to achieve speedups on the Gpu [22]. This is of course out of the scope of our automatic approach. Finally, for IS, none of the three parallel versions can gain speedups because the execution time of this program is dominated by serial code.

Overall our automatic approach performs well. The average 3.2x speedup achieved by our approach is very close to the 3.3x speedup of the manually parallelised OpenCl implementation. Moreover, our approach also outperforms the OpenMp implementation on the majority of the benchmarks by exploiting Gpu parallelism.

7.5 Analysis

Limitation of Static Analysis. Table 2 shows the number of parallelised loops of the OpenMP implementation and among those how many are statically decidable and undecidable. For benchmark CG, a considerable number of the parallelised loops are statically decidable. However, for most of the programs, merely relying on static analysis is not enough to exploit program parallelism, which actually misses a significant amount of parallel opportunities. For example, for benchmarks EP, FT and IS, static analysis fails to detect any of the manually parallelised loops. Static analysis fails to explore parallelism for these three benchmarks including EP where a speedup of 90x is available. Dependence profiling information, on the other hand, can provide us with additional information, enabling us to discover those parallel opportunities. By contrast to static analysis, our hybrid static and dynamic parallelism detection scheme identifies all the parallel loops specified in OpenMP implementation. This table shows that profile-based analysis is a powerful technique that allows us to discover parallelism for highly-likely parallel legacy code.

Speculation Costs. Figure 12 shows the overhead of the speculation for each benchmark. In this diagram, the program runtime is broken down into two parts: speculation overhead and non-speculative Gpu parallel execution. The

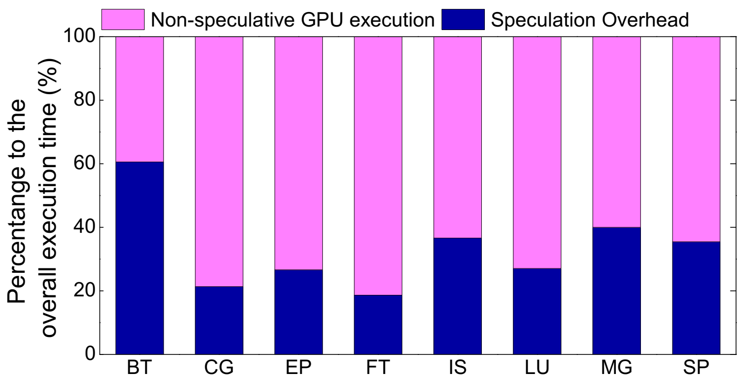


Fig. 12. Speculation overhead compared to the unsafe parallel execution without speculation on the GPU

two breakdowns are shown as the percentage to the overall program runtime. As can be seen from this diagram, the speculation overhead varies from one program to the other. Depending on the number of probably parallel loops and the frequency of speculative accesses, the overhead varies from 60% to 15% relative to the whole-program execution time. For some benchmarks, such as CG, FT and SP, the speculation overhead is relatively low, around 20%. This is because speculation only needs to be applied on a few arrays. For benchmark LU, the program execution time is dominated by the synchronisation and communication overhead due to the restriction of the program algorithm and thus the speculation overhead is not significant. For benchmarks IS, MG and BT, the overhead is more than 30% of the whole-program execution time, because of the high frequent speculative access to variables. Particularly, benchmark BT has the highest speculation overhead which accounts for 60% of the total program execution time. For this benchmark, 31 out of the 54 parallel loops cannot be statically determined and speculation has to be performed on those statically undecidable loops. Despite the speculation overhead, our approach is still able to achieve a speedup of 2.9x rather than a 3.6x slowdown of a static approach (see section 2). On average, the speculation overhead is 28% across all benchmarks.

Dependence Violation. Possibly a little surprising, in none of the above experiments dynamic dependence violations have been detected. This indicates that our profile-guided parallelisation approach correctly identifies probably parallel loops. Whilst it is easy to construct a counter example, it suggests that many loops are genuinely parallel even though static analysis is unable to prove this. In fact, we have compared the loops identified by our analysis with those parallelised in the manually derived OpenMp reference implementation of the benchmarks and confirm equivalence (subject to insertion of speculation code).

8 Related Work

Whilst specific pieces of related work have already been discussed earlier on we will provide a brief overview of TLS and profile-guided parallelisation approaches as far as relevant for this paper in the following paragraphs.

*Thread-Level Speculation (*TLS*)* Padua and Rauchwerger [20] are early pioneers of software based TLS. Their framework speculatively executes a loop as a *doall* and applies a fully parallel data dependence test to determine if it had any cross-iteration dependencies; if the test fails, then the loop is re-executed serially. There are other automatic parallelisation techniques that exploit parallelism in a speculatively execution manner [28,29], some of which require hardware support [2]. Matthew *et al.* [4] have manually parallelised the SPECINT-2000 benchmarks with TLS. Their approach relies upon the programmer to discover parallelism as well as runtime support for parallel execution. SW-TLS has been the topic of many research papers, e.g. [14,15,17,12]. All of these papers focus on individual speculation schemes, but share the assumption that dependence violations have a significant probability > 0. In fact, it is generally assumed that speculative parallel code is either generated by a traditional compiler using static analysis [31] or directly by the programmer [18]. This is different to our work, where profiling information is available and probably parallel loops have been identified for speculative execution.

Profile-Guided Parallelisation Static analyses are *fundamentally* limited by the undecidability of the underlying data flow problem [9]. This is not only of theoretical interest, but has practical implications: parallelizing compilers using static analysis are severely limited in detecting parallelism and fail to provide speedups across standard industry benchmarks representative of whole classes of real-world applications [24]. Profile-guided data flow analyses, on the other hand, have been proven to detect significantly more parallelism than their static counterparts [7,11,13], but are lacking safety, i.e. critical data dependencies can be missed. As profile-guided parallelisation has gained popularity in the academic community, several papers have investigated methods for making profile collection more efficient [8,27]. Some interactive parallelisation tools incorporate dynamic information [23], but typically this is restricted to mapping support and not used for dependence testing.

Automatic Generation of GPU Programs Some of the recent work target CUDA [10] or OPENCL [5] code generation from an already parallelized program, such as OPENMP programs. Unlike these approaches where the program parallelism needs to be identify and verify by the programmer, our compiler automatically detects parallelism from sequential code without user assistance.

Speculative Parallel Executions for GPUs The Paragon compiler [21] is the nearest work. Unlike our approach where profiling is performed off-line, Paragon uses profiling information at program runtime to determine parallelism of the statically undecidable loops by recording all the memory access. This approach, however, can incur significant overhead at program runtime. Furthermore, Paragons dependence checking scheme requires a buffer to record the memory access of speculative variables. This could lead to indirect memory accessing overhead for a separated checking process and the buffer will need to be initialized before being used. By contrast to Paragons, our in place dependence checking scheme does not have this overhead. Finally, Hayashi et al. [6] propose a scheme to automatically generated OpenCL code for Java parallel constructors and preserve precise exception semantics.

9 Conclusion

In this paper we have presented a holistic approach to exploit parallelism for highly-likely parallel legacy code on commodity GPUs. Building on prior work on profile-guided parallelization, we proposed a novel GPU-based speculation scheme to provide correctness guarantees for probably parallel loops. Our scheme discards expensive check-pointing for rollback that is not suitable for GPUs, but instead provides faster checking of dependences for speculative parallel execution regions, which are identified as probably parallel by our profile-guided analysis. Our novel approach allows dependence checking to be done in place with speculative accessing operations. We thus only need to perform checking on the addresses where speculative accesses actually take place. Our approach has been evaluated on benchmarks that are rich in parallelism, but hard to parallelize using traditional static analyses. By exploiting GPU parallel execution, we avoid the expensive overhead that otherwise would be required for serial CPU executions. We have demonstrated the effectiveness of our in-place GPU speculation scheme by comparing it to a state-of-the-art GPU-based speculation scheme. Experimental results show that our technique outperforms the state-of-the-art by a factor of 1.45. This translates to 99% of the performance of a manual OPENCL implementation without speculation overhead where the probably parallel loops have been manually verified. Our future work will explore the combination of CPU and GPU speculation schemes for auto-parallelisation on heterogeneous systems.

References

1. NAS parallel benchmarks 2.3, OpenMP C version, `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`
2. Ahn, W., Duan, Y., Torrellas, J.: Dealiaser: Alias speculation using atomic region support. In: ASPLOS 2013 (2013)
3. AMD. AMD/ATI Stream SDK, `http://www.amd.com/stream/`
4. Bridges, M., Vachharajani, N., Zhang, Y., Jablin, T., August, D.: Revisiting the sequential programming model for the multicore era. IEEE Micro 28(1) (2008)
5. Grewe, D., Wang, Z., O'Boyle, M.: Portable mapping of data parallel programs to opencl for heterogeneous systems. In: CGO 2013 (2013)
6. Hayashi, A., Grossman, M., Zhao, J., Shirako, J., Sarkar, V.: Speculative execution of parallel programs with precise exception semantics on gpus. In: LCPC 2013 (2013)
7. Ketterlin, A., Clauss, P.: Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In: MICRO 2012 (2012)
8. Kim, M., Kim, H., Luk, C.-K.: Sd3: A scalable approach to dynamic data-dependence profiling. In: MICRO 43

9. Landi, W.: Undecidability of static analysis. ACM Lett. Program. Lang. Syst. 1(4) (December 1992)
10. Lee, S., Eigenmann, R.: Openmpc: Extended openmp programming and tuning for gpus. In: SC 2010 (2010)
11. Mak, J., Faxén, K.-F., Janson, S., Mycroft, A.: Estimating and exploiting potential parallelism by source-level dependence profiling. In: EuroPar 2010 (2010)
12. Mehrara, M., Hao, J., Hsu, P.-C., Mahlke, S.: Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In: PLDI 2009 (2009)
13. Mishra, V., Aggarwal, S.K.: Partool: A feedback-directed parallelizer. In: Temam, O., Yew, P.-C., Zang, B. (eds.) APPT 2011. LNCS, vol. 6965, pp. 157–171. Springer, Heidelberg (2011)
14. Oancea, C.E., Mycroft, A.: A lightweight model for software thread-level speculation (TLS). In: PACT 2007 (2007)
15. Oancea, C.E., Mycroft, A.: Set-congruence dynamic analysis for thread-level speculation (TLS). In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 156–171. Springer, Heidelberg (2008)
16. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: SPAA 2009 (2009)
17. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: SPAA 2009 (2009)
18. Prabhu, M.K., Olukotun, K.: Using thread-level speculation to simplify manual parallelization. In: PPoPP 2003 (2003)
19. Rauchwerger, L.: Speculative parallelization of loops. Springer, Heidelberg (2011)
20. Rauchwerger, L., Padua, D.A.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. IEEE Trans. Parallel Distrib. Syst. 10(2) (1999)
21. Samadi, M., Hormati, A., Lee, J., Mahlke, S.: Paragon: Collaborative speculative loop execution on gpu and cpu. In: GPGPU 2012 (2012)
22. Seo, S., Jo, G., Lee, J.: Performance characterization of the nas parallel benchmarks in opencl. In: IISWC 2011 (2011)
23. Thies, W., Chandrasekhar, V., Amarasinghe, S.P.: A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In: MICRO 2007 (2007)
24. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine -learning based mapping. In: PLDI 2009 (2009)
25. Vandierendonck, H., Rul, S., De Bosschere, K.: The paralax infrastructure: Automatic parallelization with a helping hand. In: PACT 2010 (2010)
26. Vanka, R., Tuck, J.: Efficient and accurate data dependence profiling using software signatures. In: CGO 2012 (2012)
27. Vanka, R., Tuck, J.: Efficient and accurate data dependence profiling using software signatures. In: CGO 2012 (2012)
28. Wallace, S., Calder, B., Tullsen, D.M.: Threaded multiple path execution. In: ISCA 1998 (1998)
29. Wu, P., Kejariwal, A., Caşcaval, C.: Compiler-driven dependence profiling to guide program parallelization. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 232–248. Springer, Heidelberg (2008)
30. Yu, H., Li, Z.: Fast loop-level data dependence profiling. In: ICS 2012 (2012)
31. Zhai, A., Wang, S., Yew, P.-C., He, G.: Compiler optimizations for parallelizing general-purpose applications under thread-level speculation. In: PPoPP 2008 (2008)