Summary:

The paper proposes a new *behavior oriented parallelization (BOP)* technique, which aims to allow a program to be parallelized based on partial information about the program behaviour. In this context, behaviour oriented means, that unlike other parallelization approaches, the presented approach doesn't try to achieve a maximal degree of parallelization in the program in all possible cases, instead it tries to improve some executions, a behaviour, containing possibly input depending parallelism. At the same time however correctness and basic efficiency for other cases have to be guaranteed. The system relies therefore on user annotations or a profiling tool to suggest *possibly parallel regions (PPR)*, in which behaviour oriented parallelism might could be exploited.
Each PRR p is marked by matching markers: `BeginPRR(p)` and `EndPRR(p)`. This markers are afterwards used by the BOP system to construct a sequence of non-overlapping regions, which are candidates for parallelization. Therefore, if BOP encounters a start marker, it starts a new process, which will jumps to the end of the corresponding region and execute speculatively from there. The execution of the different processes inside the BOP is divided into a *lead* and several *spec* processes. The lead process always performs a sequential execution of the program until program exit. Especially, the lead process continues its sequential execution of the program, even after it completed its own PRR. This is called the *understudy* process, which performs actually a re-execution of the following code. If the understudy process finishes before the corresponding speculative execution, i.e. the speculative execution takes longer than the sequential one or a conflict is detected, the speculative process for the re-executed region is aborted and the understudy process commits. The concurrent speculative execution of other PRRs is continued. However, if the spec process finishes early and no conflict is detected, it is able to commit and abort the sequential understudy. Afterwards, it becomes the new lead process. For a greater number of spec processes, this is done recursively in a similar manner. Altogether, this allows the speculative execution to try to aim for a lower execution time, while having always the sequential execution as backup with a bounded amount of extra costs (time).

Issues / Open Questions:

1. For me, the approach seems to have a high memory utilization, which could limit its application on several occasions. However in the evaluation, there is no comparison provided between the memory usage of the sequential and the parallel version.

Summary T2

This week we discuss a paper about software behavior oriented parallelization (BOP). The authors introduce a software speculative system which is able to parallelize programs with unknown or only partially predictable behavior. It contains a profiler which identifies regions in the code that can be possibly parallelized (PPR) and marks them. To ensure correctness it is necessary to protect the address space which is done by three very cost effective techniques: programmable speculation, critical-path minimization and value based correctness checking.
When parallelization is applicable the PPR marker have to be set either by the automatic profiler or manually by the programmer. Code sections framed by BeginPPR(p) and EndPPR(p) can be executed in parallel where p is a unique identifier. When there are several entries to a certain code section several BeginPPR(p) markers for the same p are valid, but there must not be more than one EndPPR(p) marker.
Program execution always starts in the so called lead process. When this process encounters a BeginPPR marker it forks a speculative process and continues with the execution of the PPR instance. The forked process, called spec 1, jumps to the corresponding EndPPR marker and already starts executing the next PPR instance. This forking can also be nested, meaning that spec 1 forks a new speculative process spec 2 when it reaches a BeginPPR marker itself. When lead has finished its PPR instance and reaches EndPPR it becomes the understudy process which re-executes the following code sequentially for redundancy. This provides an upper bound for the execution time when the speculative execution gets into an endless loop or when it is slower than the sequential. When spec 1 has finished lead checks for conflicts and commits its data to spec 1 which is then the lead process. Since BOP is a process-based approach it provides strong isolation, so no synchronization is needed which saves overhead in the non-speculative process.

Questions / Opinion:

1. The authors use a memory system with page granularity. To avoid false sharing of global variables they allocate them on several pages. So in my opinion they waste a lot of free space which could result in a lack of memory

2. In 2.3.1 they say that a speculative process k fails if a page is written by lead AND the privious k-1 spec processes and read by spec k. Why isn't it "... lead OR the previous ..."

1 Summary

In this paper, the authors present a software system for behavior oriented parallelization (BOP). In this context, behavior means the set of all possible executions of a program. The main idea of this system is to combine sequential execution with speculative parallel execution of the original program. Hence, the critical path of the parallelized program is no longer than the critical path of the original one. The parallelized variant starts execution in the lead process. If the program reaches a point marked as the begin of a *possibly parallel region* (PPR), it forks a new speculative thread (here called *spec*) that begins at the end of the PPR. This way, specs are spawned recursively until a certain limit of specs is reached. When the lead thread terminates executing the PPR, it proceeds with the *understudy*, re-computing the part that the forked spec is computing. The understudy is used to guarantee correctness and counter potential performance loss if a speculation fails. Whenever a spec finishes faster than the understudy, it means a performance improvement over the sequential execution.

To preserve correctness of the program, the BOP system has to make sure that if a spec succeeds, the same results are produced as in the sequential execution, and if a spec fails, a correct rollback is performed. In this context, the authors describe *strong isolation*, which is an important property of the BOP system.

The system partitions the address space into the three disjoint groups shared, checked, and private. For each group, techniques are proposed to guarantee independence among parallel executions and to merge the results after successful speculation. A very interesting fact is, that the BOP system does not require any transactional memory, but only virtual memory provided by the OS.

BOP allows reordering of operations and is independent of the hardware's memory consistency model. BOP features a profiling analysis that detects PPRs in linear time, and allows the programmer to annotate the source code with BeginPPR and EndPPR markers to suggest PPRs to the system. Furthermore, the system provides run-time feedback to the programmer about parallelization success.

2 Questions

How can the system merge the pages of speculative threads and/or the lead process without transactional memory or explicit locking?

Software Behavior Oriented Parallelization

1.Summary

Software behavior oriented parallelization is a technique that divides memory space of a process into multiple disjoint regions in order to detect data hazards between concurrent threads in operation. In this method, user explicitly identifies parallel regions in a program using constructs BeginPPR and EndPPR. Distinguishing feature of BOP is that it uses a race between speculative parallel execution and sequential execution to ensure it is as efficient as sequential execution even in the worst case.

Parallel regions are executed using a lead process, and a spec process which is forked to start execution of next parallel region that follows the EndPPR of current region. This happens recursively so that first spec process is the lead of second spec process, and so on. Once the lead process finishes executing its parallel region it starts executing understudy region. This understudy region is nothing but the same region executed by next spec process. Whichever of the two processes finishes first is used to commit to memory, while the other process is aborted. It should be noted that changes by spec process is committed first before the lead process. If a spec process encounters exit, only that thread is aborted. Whereas an exit in lead or understudy would abort the whole program.

Conflicts are detected by checking which region of the memory an access belongs to. Different access levels are provided to lead process and spec process for each region. A modified page fault handler is used to detect which thread has caused it and modify its permissions. There is a checked region which indicate that there might be no dependence even if it looks like there is one. Private data regions indicate memory can be accessed without the fear of conflicts.

2. Open Questions

1. How conditions in which a spec thread “free” a region which is to be accessed by lead thread are handled?
2. How the finish time is independent of length of speculative execution?

Software Behavior Oriented Parallelization

Summary

This weeks paper deals with a parallelization framework which is able to exploit even incorrect/incomplete external information – given by user, compile time static analyses etc. – to speculatively parallelize the execution of a sequantial program at runtime.

The speculation in this case is not based on transitions as such but on so called *possibly parallel regions* (PPR). Those regions are considered as good candidates to be executed in parallel to the remaining control flow. That is designed as follows:

Whenever the run-time systems reaches the beginning of a PPR, it starts a new thread that speculatively skips that region and executes the code behind it. This scheme may be extended recursively for new PPRs occuring in the speculative branch, thereby gaining more and more parallelism. The original branch executes the region itself and goes on even afterwards, now starting a race with the speculative branch, acting as a kind of sequential backup. If the speculative branch finishes faster and without conflict, it overtakes the control and replaces the old branch. Otherwise, the sequential execution continues and the speculative branch is dropped.

Speaking of conflicts: The system uses a quite broad memory access conflict detection system, partitioning the memory into three areas that may dynamically interchange, but never share variables. The areas are *shared data*, a striped memory area that is protected quite similar to transactional memory, a set of *checked* values which are assumed to be privatizable at least most of the time, but cannot be shown to always be, and a set of *likely private data* that contains variables that are known to be privatizable by static analysis (but confusingly also not always...).

It follows a correctness proof of the system, a short overview of how the classifying data used to partition memory and good PPRs are found (either by combination of several analyses or manually), as well as the obligatory comparison to other work and the evaluation part.

Open Questions

- Wouldn't it be nice to also have a subset of truly private values that can be shown to need no checking at run-time at all additionally to the two sparely checked value/variable sets?

Summary Software BOP

The paper introduce a new speculation technique, which, compared to all previous papers, does not use shared memory, but separate processes. This helps with obtaining stronger isolation. The system requires the user or an analyzer to mark some regions as possibly parallel. Multiple entry points are allowed, but not multiple exit points. It is up to a runtime system to decide which parts are actually parallelized.

The actual parall system uses three differet kinds of processes, or rather process roles, lead, spec and understudy. Execution starts with one single process, the lead process. Upon encounter of a begin marker of a parallel region, a spec(ulative) process is forked, which starts execution from the corresponding end marker. After the lead process reaches the end marker, it becomes an understudy process, which executes the rest of the program seqentially. This is used to ensure that no slow down occurs when the parallel execution is slower than the sequential one due to the overhead of the system. If the understudy finishes before the speculative process, it just aborts them. When a speculative process finishes, the lead process checks for conflicts, and if none occured, it commits it changes to it. Afterwards the speculative process becomes the new lead.

The authors justify their rather expensive process based system with the benefits it provides, namely strong isolation. The authors argue that this removes synchronization overhead in the case where speculation fails. Therefore, mthe argue, their system would be benefittical when an author unfamiliar with the code base adds the PPR markers, causing a large amount of conflicts due to hidden dependencies.

Their strong protection is based on Unix processes, and differs between three types of data: Shared data is checked at page granularity; Unix read/write permissions are used to check for conflicts. For this, all processes have the data set to not readable/writable; upon write/read, the permission is set accordingly. A conflict for a speculative process S only occurs if there was a write by the lead process or one of the former speculative processes, and a read by S. This can however lead to false positives, when a process writes the same value it read. To prevent this, some variables can be marked as checked (e.g. by an analysis). For those, the runtime system will check the values at commmit time. The last type of data is private data, which never causes any conflict.

In their evaluation, the authors can achieve a speedup up to 2 times and more (at the cost of a notable increase in memory consumption), though they sometimes have to adjust the benchmarked program to prevent conflicts.

Open Questions:

- Figure 3 seems to be a bit misleading. In the text they say that the lead proces starts the understudy and waits for spec to commit. But in the figure, lead and understudy seem to be the same process.

Summary T2

This week's paper is about software behavior oriented parallelization. All of the other papers so far have been about analyzing source-code and performing the parallelization before the program is run – regardless of factors like input data, system load, etc. Although using this method, too, this week's approach can actually make use of those factors, albeit only indirectly; more like a welcome side-effect of the way the approach works.

A profiling tool marks code regions for possible parallel execution by inserting markers at the beginning and the end of those sections. *BeginPPR* branches off a code section to be executed in parallel, while the original thread is jumping to the *EndPPR* marker and continues execution from there. As soon as a parallel code section finished – meaning it reaches the *EndPPR* marker – it commits its work. I/O and system calls are simply handled sequentially as they cannot be rolled-back in general.

When branching off parallel processes, the special thing about this approach kicks in: after doing the branched-off work, the parallel thread continues with the same code which the main thread already started to execute. But while the main-thread, which started the parallel execution, switched to ignoring the markers and therefore executing sequentially, the branched off thread still allows new parallel code sections. This means that the sequential and the parallel version are running at the same time and one of them will finish first. In case of the parallel code section being the first to finish, the sequential version is aborted and the branched-off thread becomes the new main process. This applies recursively to all processes until the maximal speculation depth is reached.

This design allows for one particular benefit: if the environmental factors like input data, system load, etc. influence the program in such a way, that parallel execution is slower than sequential execution, the program loses nearly no time compared to the original sequential version as the sequential version was also running and can simply use its own work instead of the parallel processes.

Open questions:

If I understood the presented approach correctly, it concretely branches off instructions and executes those in parallel to other instructions. But this does not unroll loops. Does this mean, that only a whole loop can be executed in parallel to other instructions, but the iterations of the loop cannot execute in parallel to one another?