Summary:

The paper proposes an approach called *Fast Track*, which intends to optimize the run-time of sequential programs by supporting unsafe optimizations through speculative execution.
Unlike other approaches, we've seen so far, the presented approach does not really try to perform an automatic parallelization of the given program. Instead it uses the parallel execution to provide some kind of fall-back while allowing possibly unsafe optimizations on the given program.
In general, the execution of a so-called *fast-track region*, a code region where unsafe optimization is allowed, is done independently in parallel by two different processes: the *normal* and the *fast instance.* Together, this execution is called a *dual-track instance.*
As anticipated by the name, the fast instance executes the code of the region, which has been optimized optimistically to gain an advantage in terms of the execution time. The normal instance just executes the unchanged code. During the execution of both instances, the changes done to the memory are observed and recorded. Therefore the memory is mapped to each process as read-only at first, until the first write access is detected. Afterwards the memory is duplicated and the access recorded for conflict detection. As in other approaches, this is realized by the virtual memory handling capabilities of the hardware and the OS, such as "page-fault" handlers. After both instances are finished, the memory is compared for differences, in which case the *fast* track execution is aborted and the result from the normal used.
Obviously, the approach is safe and provides a worst case run-time close to the sequential execution, since in the worst-case, the result is indeed the sequential one. The question now is, why is this approach useful at all and yields a performance gain?
Indeed, both versions of the code region are always executed in parallel, but the execution won't be stopped, if the code region is done. Actually, the fast instance is sooner able to execute the sequential code after and outside the fast-track region. Finally, when the sequential execution of the normal instance completes, and both computations are considered equal, then there is no need to re-execute the sequential code after the region, since this has already been done by the fast-track process. When a fast-track region is encountered again, the normal process can be immediately started at this point, which yields a performance gain in terms of the execution time. If at any time, the execution of an fast instance is wrong or slower than the normal instance, the fast instance can be just aborted. Afterwards the execution can be restarted after the fast-track region with the result of the normal instance.
The approach therefore offers an interesting speed-up in the optimistic case, while providing a near as sequential run-time in the worst case.

Summary T3

The paper of this week deals with an approach called "Fast Track" and describes how speed up programs with the help of parallelism. In contrast to the papers we discussed in the last weeks this approach does not use parallelism in order to execute different parts of code by different cores or processors. Instead Fast Track enables the compiler to optimize the program more aggressively which would lead to a loss of correctness. So it uses instances running the safe original code in parallel to verify the results of the fast instance. The speedup arises due to the fact that the execution of the program can continue as soon as the fast instance has finished and does not need to wait for the normal instance. If the verification fails all further computations run by the fast instance that are based on the result of the failed one will by undone by a rollback. Beneficial applications for these technique are Memoization where one has to make a compromise between the benefit of precomputed results and overhead in loads and stores. Using Fast Track always the faster variant wins this race. Another field where Fast Track can be used is the optimization in compiler, because there are many optimizations that destroy safety but provide a high speedup.
To use Fast Track the programmer has to call the function FastTrack which returns a boolean value whether the current process is the fast or the normal instance. Based on this condition the corresponding code can be executed. At this point the dual-track instance starts and it ends at the call of the EndDualTrack function which behaves as a joint point. Nested dual-track instances are allowed, but statements containing side effects (system calls, file input / output) are forbidden.
Technically the system is based on UNIX processes each with a separate address space. To enable rollbacks changes to the memory have to be recorded. Data on the stack is protected by the compiler and data on the heap as well as global variables are protected by the OS paging support manipulating access bits and installing customized page-fault handlers that are able to record all memory accesses.

Questions / Opinion:

1. On page 2 the authors write about "late bindings", what are these?

1 Summary

In this paper, the authors present Fast Track, a system that allows to run unsafe, aggressively optimized code on a fast track of execution. To guarantee correctness, the system runs a safe version of the code on a separate process to verify the results of the fast track. Fast Track can be used directly by a compiler, but also offers a programmable interface to manually mark unsafe code.
The unsafe code is run in parallel to its safe version, where the two processes are generally spawned synchronously. If the machine only provides two cores, the processes can also be spawned asynchronously to increase the performance gain. When the normal instance terminates, it verifies the results of the fast instance. When an error appears on the fast instance or the results of the normal and the fast instance differ, then a new fast instance is started at the next dual-track instance; it is considered an error if the normal instance terminates faster than the fast instance.
Fast-track regions can be nested, however, no new processes are spawned. The system forbids abnormal branches, system calls, and I/O inside a dual-track region. The system uses paging to protect global and heap data and access maps to detect errors.
Considering a set of parameters that significantly affect the possible performance, the authors derive a model to compute potential speed ups. They claim that the parameters can be efficiently monitored at run time such that the model can be used to tune Fast Track at run time.
In the evaluation section, the authors present their hand-crafted benchmark. Their results show significant performance improvement.

2 Questions

I think this paper is missing important information about how their system is designed and how it operates. Especially the underlying techniques used by Fast Track should be explained.
Furthermore, the experimental results are highly questionable. How representative is their synthetic test? It would have been better to use different, well known tests.

Fast Track:Supporting Unsafe Optimizations with Software Speculation

1 Summary

Fast track is a parallelization technique that mainly focusses on improving the performance by exploiting unsafe optimizations done on the programs. Similar to the previous paper discussed, there is an execution of both sequential and fast track version. However, one of the differences is that sequential version is continued even though fast track version wins the race.

Programmer or compiler inserts the optimizations into the code that might be unsafe. Such portions in the code are executed in dual track. This means both optimized and unoptimized versions are executed as separate processes in parallel. Such executions are made independent of each other by replicating address space or possibly using Copy-on-Write. Variables that are changed by both processes are recorded. Global and heap data are protected using a modified page fault handler. When a fast track execution finishes, which is likely to happen before sequential execution, it starts execution of the next dual track in the code. When a sequential execution finishes, process compares its modified data with that of fast track execution. If they are different or fast track is still in execution, fast track process is aborted. And next dual track in the code is started. Sequential execution ensures correctness of the system, and is the performance of the system in worst case.

Dual track executions are highly resource intensive. Authors discuss few optimizations like enforcing a limit on number of dual track executions that can be active at the same time. There is also a limit on the amount of dynamic memory that can be allocated by a fast track process. While starting a dual track execution if only one processor is available, sequential track is started on the available processor. Fast track execution waits for a processor to be idle.

2 Questions

1. In asynchronous dual-track execution, is there any advantage in starting normal execution instead of fast track execution when there is only one processor available?

2. In section 4.1, author discusses about converting global data to heap data. I don’t understand how that helps.

Fast Track: Supporting Unsafe Optimizations with Software Speculation

Summary

This paper introduces a source code level software speculation scheme that facilitates multi-core systems to accelerate sequantial programs by running both a safe version and a *unsafely optimized* version of certain program parts. The kind of optimization is in general not fixed. As a special application told to be convenient, *memoization* is mentioned, which seems to be some sort of speculative dynamic programming. Despite the approaches discussed so far, this system also allows for improvements beyond parallelization per se.

The systen is region based where regions are bounded by a distinct entrance point and a single exit point which is dominated by it. Irregular behavior such as system calls and file operations are not covered and thus needs to reside outside of those regions. For each region there is a *normal track* and a *fast track* (the optimized version of the normal track). When reaching the entry point, the current thread forks the fast track to a separate process. In the bad case, speculation fails or is slower then the normal track. Then, the fast track is aborted/canceled and the normal execution proceeds. In the optimistic case, the fast track finishes earlier and can proceed with the execution of the program, including the starting of subsequent dual-track executions which might gain true parallelism. The normal track verifies memory effects logged by both tracks and hands the program control over to its fast track. This is said to be done by message-passing, but no concrete information about implementation details (such as protocols) is given.

As theoretical as the description of the system itself is the first part of the performance analysis. While the formulas and mutual relations between the different attributes of the system seem to be plausible (at a first glance) – but nevertheless also quite artificial given the abstraction level – the results gained from that (despite some theoretical bounds) seem to be not that expressive.

In addition to that, there follows an experimental analysis of the system whose real implementation even in this context is only roughly given. The results are – as one would expect, given the single fairly artificial benchmark program – equally informative.

Open Questions

- Has anyone in the meantime evaluated that in real world applications? For example in combination with the memoization technique mentioned in the introduction?

Fast Track Summary

The authors introduce Fast Track, a system which enables programmers to make use of arbitrary unsafe optimizations. The basic idea is the same as for BOP: Execute the unsafe (for BOP: speculative) version in parallel to the normal version. Once the normal version has finished, use its result to check whether the optimized version computed the same, correct value. If the "optimized" version didn't finish so far, or if the results don't match, its process is killed. A speedup is achieved by letting the process which run the optimized version execute the next section of the program. As with BOP, Unix processes are used for parallel execution, and the operating systems page fault handler is used to check which data is actually modified. This is done by initially removing the write permissions for all pages. Whereas in BOP, an overlap in modified pages would be a conflict, it's actually required for Fast Track. Furthermore, checking code has to be inserted, as pages being modified doesn't implies that the modifications are the same. Unlike BOP, Fast Track cannot handle stack data, which is a) startling when keeping correctness in mind and b) surprising, considering that C. Ding and K. Kelsey worked on both projects.

The authors spend some time to talk about how their system deals with resource constraints: Memory overhead is deemed negligible. For processors, they consider to different schemes of running the two tracks: In one they only start the tracks when there is a free processor for each of them. In this case, unnecessary waiting times are added. In the second scheme they execute the normal track as soon as possible, and the fast track when another processor becomes available.

To analyze the possible speedup, they give a mathematical formula describing the execution time. In their formula, 4 main parameters appear: speed of the fast track, its success rate, the overhead added by the system and the percentage of the program which is actually executed in the fast track. They use a simulation (based on their formula) and conclude that the actual speedup is highly dependant on the parameters. They claim that those parameters can be monitored at run time, which could be used to control the system, especially the depth of concurrently running sections.

Open questions:

- How could they perform their synthetic test when their system doesn't support stack data so far?
- Is their formula applicable at all when one doesn't consider equally sized loop bodies, but more irregular programs?

Summary T3

This week’s paper is about "Fast Track", an approach to implement possibly unsafe optimizations without falsifying the result. The idea is to split the program into 2 parts: a so called "fast track" and the "normal track". The fast track executes the aggressively optimized code and every time a new possibly unsafe section is launched, it spawns the safe version as a seperate process which is used to validate the result of the fast track.

The fast track does not wait for the normal track to finish before continuing. This means that later sections could use wrong results from previous sections before the normal track finishes the earlier section and validates the fast track result. In such a case the fast track is simply aborted and abandoned and the corresponding normal track becomes the new fast track. This guarantees that always the right result is used.

To not flood the system with new processes till infinity, the fast track depth - meaning how many normal tracks may run in parallel to the fast track - can be limited.

If the fast-track encounters irreversible effects like exit points or I/O, it stalls until the normal track performs this operation.

Another guarantee that they give is, that the fast-track may only allocate a certain amount of memory. If this limit is exceeded, the process is aborted. This ensures that an application does not have a sudden n-fold which might be a problem on certain systems.

Open questions:

Is there some kind of commit order for the normal tracks when it comes to I/O effects? The fast-track may not execute those but has to wait for the normal-track to execute them. But what if a normal-track B started with the input of fast-track A, this input is not yet verified by normal-track A, and B wants to execute an I/O effect. To ensure correct results, it would need to wait for normal-track A to finish. Is this what they do?

Fast Track: Supporting Unsafe Optimizations with Software Speculation

1 Summary

The paper proposes a parallelization system called Fast Track which will enable the programmer to specify so called dual-track regions which will be speculatively parallelized. Fast Track will execute a program sequentially at first, if it encounters a dual-track region it will spawn a new process which will execute the region using some other arbitrary parallelization scheme.
The original process will execute the code in the region sequentially. When the parallel track reaches the end of the dual-track region it will preserve all the memory it changed and will start executing the code after the region. When the sequential track finishes it will first check if the parallel track has left the region and kill it otherwise, in the other case the sequential track will compare the memory actions of both tracks and if they differ kill the parallel track or quit otherwise.
If the parallel track at some point reaches an exit statement or an illegal operation (such as errors or systemcalls) it will wait for the sequential track to either finish and kill the parallel track or to execute a correct exit statement. The system allows nested dual-track regions, if any nested dual-track region is encountered the code will not introduce another layer of concurrency.

2 Questions

- Where and how does the fast instance store its memory footprint when it reaches the end of a dual-track region?