Summary:

The paper proposes an new parallelization technique, which provides dynamic runtime parallelization of loops from binary single-thread programs with minimal architectural change. The realization of this technique employs Transactional Memory (TM) and a dynamic optimization framework featuring a dynamic re-compiler, which patches re-compiled code into a running program.

The proposed technique targets the parallelization of loops out of legacy binary code. Since the source code of this legacy code is not available, the first step of the approach is to identify all possible candidates of parallelizable code. This identification is done statically by building the dominator graph of the program among basic blocks to find natural loops. Notable is that loops that contain e.g. system calls are excluded, since the rollback of this loops wouldn't be possible in the context of a TM, as the effect on the operating system can't be easily reverted or controlled. During execution of the program the result of the static analysis is combined with measures of a hardware-based profiler, counting the frequency of taken branches and thereby identifying *hot* - frequently executed - loops. After a hot loop is identified, the hardware profiler determines the average number of cycles required to execute one iteration of the loop. This measurement is later used to evaluate, if the parallel version of a loop performs really better than the sequential one. The evaluation is constantly performed by the dynamic-optimization framework during the execution with the possibility to dynamically replace the parallel version of a loop with their sequential counterpart, thus trying to archive the most best possible speedup for an application.

The iterations of a parallelized loop itself are wrapped in transactions of the TM, providing a way to perform a cross-iteration dependency checking. This even leverages the need of performing a complex static pointer, data-flow and dependence analysis, which is fairly limited in detecting real cross-iteration dependency through a possible high amount of false-positives. Since the realisation of the TM itself is still complex and could lead to a high overhead, the technique features the use of a TM realized in hardware to operate as efficient as possible. This HTM thereby has to support the ordering of transactions, to ensure that all loop iterations are committed in the original sequential order of the program. But not everything can be handled by the HTM, as the values of the thread register are not covered by the TM and have to be passed to each specific iteration. For loop carried dependencies, the re-compiler generates fix-up code, which tries to "anticipate" the values of the corresponding variables optimistically. Otherwise the re-compiler just uses the value of the register, which was last provided by a thread, out of a shared array in the memory, leaving the detection of a possible conflict to the TM. Generally the conflict detection is built into the hardware cache coherency protocol and therefore happens eagerly at relative low costs, thus allowing the TM to detect conflicts soon and restart the conflicting transaction immediately. The dynamic-optimization framework monitors this behaviour concurrently and is able to revert the loop to sequential execution, if conflicts/restarts happen to often.

Open Questions:

1. The re-compiler places ATX (Abort Transaction) operations at each early-exit paths of a loop iteration. This causes the transaction to finally roll-back the memory state. But what happens next? Is the transaction retried? Is the loop execution aborted and sequentially retried? What if this early-exit is a legal break of the loop? Or does aborting the transaction here just means, that the succeeding iterations will be aborted?

Summary T1

This week we talk about a paper dealing with runtime parallelization of legacy code. It presents an approach for automatically parallelizing loops in binary code using transactional memory in order to increase performance compared with lock-based systems. The authors assume a hardware transactional memory for their system which is not yet common for many of today's processors to support this. Furthermore they use a dynamic optimization framework which identifies loops at runtime, analyzes them for parallelism, re-compiles and patches them into the running program.
When generating code for loop being parallelized a new thread is forked which starts executing at the label provided as an argument. This fork does not create stack for the new thread, it only copies the registers and sets the program counter. In parallel the forking thread itself branches to the label "btx" where it creates a new transaction and executes the loop body as the forked thread first has to update the loop-carried registers before it also branches to "btx" and executes the loop body. When an iteration has finished the transaction is committed and, in case it is not the last one, a new one created. In doing so the ordering of the transactions takes care of committing the data in the right order. When all iterations the execution continues in the original thread at the label "join".
As data dependencies in memory between two threads are handled by the transactional memory system shared data between registers need further mechanisms are used to guarantee correctness of the approach. At this point one has to distinguish between loop-carried and non-loop-carried registers as well and one has take care of conditionally-written registers. In the worst case spill-code is generated to outhouse the registers accesses to memory and benefit from the transactional memory system that can deal with this problem.
In the benchmark section of the paper the authors could achieve average speedups of 1.45 (NAS benchmark) and 1.38 (SPEC FP benchmark). Despite of the fact that they use a simulator due to the need for a HTM system I think they provide reasonably realistic benchmark result.

Questions / Opinion:

1. I do not understand why they constrain their approach to legacy binary code and do not extend it to high level code.

2. For their approach they assume the existence of a HTM system, but they do not explain how the limitation of such HTMs have an influence on their system. So for example I would be interested in how far the cache size which influences the transaction size has an effect on the speedups.

1 Summary

In this paper, the authors present a runtime parallelization technique to automatically parallelize programs. They specifically target program binaries, and focus on loop parallelization. It consists of a Hardware Transactional Memory system, a dynamic optimization framework, and their so-called re-compiler.
Their technique tries to leverage the potential parallelism by making intensive use of the TM. It requires the TM to support atomic transactions and ordered commits, since the re-compiler relies on these features to preserve the program's semantics.
The dynamic optimization framework features a control-flow analysis to identify loops and a hardware monitor that identifies frequent branches. At the program start, the control-flow analysis is performed once to detect loops; the results can be stored for future executions. As the program starts, a hardware-based profiler is launched to find *hot* loops.
After performance measurements on the sequential program, the re-compiler is spawned to parallelize the hot loop. If the parallelized version does not perform better than the sequential one, the program falls back to the old version.
When it comes to parallelizing a loop, the iterations must be distributed among several threads. Therefore, the loop can be split into *tiles* (sets of consecutive iterations). The re-compiler will introduce lightweight forks to spawn new threads. Since these threads generally do not share the same registers, control-flow through registers must be transformed; the re-compiler will simply fetch the corresponding registers to/from the TM. By exploiting the TMs commit ordering, the re-compiler ensures that the main thread sees the correct live-out registers. Since threads are spawned by a lightweight fork, they all share the same stack, what may cause problems. The re-compiler will inline function calls to depth 1. Further function calls are replaced by early exit points. If a transaction reaches such an early exit point, it aborts, and the program will fall back to its sequential variant.

2 Questions

In 5.1.2, why do they require the ETX instruction to be executed after the branch to a new iteration has been taken? Maybe in case of a rollback?
In 5.3 the authors claim that allowing function calls in parallel loop iterations is problematic. For what reason? May this cause data-races on the stack?
To me, their benchmarks seem cheated, since they omit the time taken for the one-time static analysis, and the time for rejecting inefficient parallelized loops.

Runtime Parallelization of Legacy Code on a Transactional Memory System: Summary

Paper discusses a parallelization technique focussing mainly on loop iteration parallelization. This system is built on a transactional memory system. There is also a feedback system which analyzes if it is really worth parallelizing the program.

There is a dynamic framework that is responsible for identifying parallel portions in the program, and converting it into a program suitable for parallel execution. A static loop analysis is performed to find out the candidate loops. Loop iterations are grouped together to form a unit of parallel work called tile. Authors are convinced that tile size does not affect the performance much. Loop iterations(depending on the tile size) are packed into ordered transactions. Transactions are started, and conflict identification relies on extended cache coherence protocol. If a transaction commits, cache values are propagated into caches of other cores or to the next level in the memory hierarchy. Commit orders are maintained. If a transaction aborts, corresponding cache entries are invalidated. Again, this has to wait for all older transactions to commit. A transaction may be restarted when a memory violation is identified by the cache coherence protocol.

Loop carried register dependencies are handled by adding induction code for such variables at the beginning of the loop iteration. Registers that are written conditionally are handled by converting them into memory variable, and then loading that value back to the register at the end. New instructions namely spill and fill are introduced for that reason.

If loop iterations have function calls, such function calls are inlined in order to avoid allocating separate stack frame for each such call. Dynamic re-compiler also ensures that control is returned back to the call site.

Open Questions:

1. While handling function calls in loop iterations, what happens if function is recursive or it has some static variables?

2. In section 5.3, what is the meaning of uninstrumented functions that don't return to call sites?

Runtime Parallelization of Legacy Code on a Transactional Memory System

Summary

This weeks paper discusses an attempt of using a hardware based ordered transactional memory system (TMS) with eager violation detection for automatic parallelization.

After some differentiation to related work they describe a dynamic attempt to parallelize already compiled code during execution. The work is not bounded to to a certain hardware, assuming several special purpose features as given such as a Hot Path Profiler for recognizing often executed loops. Using that combined with some program analysis, they find loops as candidates for parallelization during runtime. This is then exploited by generating parallel code for the loop, replacing it on the fly, checking whether the new version performs well and if not, either tweak some parameters of the parallel code such as how many loop iterations are combined ("tile size") into one transaction or fall back to sequential code.

The parallel code that is generated during that mechanism is structured as follows: First of all, each *tile* (rather small package of loop iterations that is executed in in a dedicated thread) is wrapped into a transaction. The transaction itself is then structured as one expects, containing suitable exit points where needed. The handling of data dependencies that are carried by registers are pushed into the TMS by spilling affected registers (detected by another analysis executed at runtime) to memory. This also solves the problem of detecting the last value written to a register, but introduces a huge runtime overhead, thus being only suitable if of only small impact. Another design decision that is equally sparsely explained is the handling of function calls. Since the systems uses fast forks, it cannot easily establish the stack of a function call inside of parallelized code which is handled by inlining of function calls, but only at first call level. In the end, all that is put between a loop prefix and suffix which handle thread forking as well as copyin and -out of shared values.

The paper ends with an evaluation that assumes that everything that is system specific and is supposed to happen during runtime has already happened. The result shows that there is a reasonable amount of potential parallelism in certain benchmark suites, but whether the described approach is truly able to exploit that remains somewhat unclear.

Open Questions

- There are several design decisions that are sparely justified, such as the restricted inlining of function calls or nested loops in induction code. Are there objective reasons beyond code complexity for this?

1 Summary T1

The paper presents a further approach to automatic loop parallelization using dynamic optimization. The most distinguishing feature compared to the previous approaches we have seen is that the authors are working on the binary level.

Their approach is dynamic: During runtime, in parallel to the actual program, a loop analyzer is run, finding loop–like instruction sequences in the program. It excludes those containing system calls, probably because those are forbidden in their transaction model. Then, those loops are profiled (also during runtime), seemingly using hardware performance counters. They then use a recompiler, to generate parallel versions of “hot” loops. The recompiler does the following: It wraps the loop body in a transactional context (provided by a hardware transactional system). Lastly, they add a number of “lightweight fork” operations, so that multiple transactions are executed in parallel. The fork is called lightweight, because it does not create a new stack. They argue that this is more benefittical than locks, as the latter would require them to pessimistically aquire a lock on loop entry and release it on exit, making the loop basically sequential again. With HTM this is not an issue, as it is optimistic by design, therefore leading to better performance when the loop iterations are actually independent. As the HTM only handles memory, they have to add code to handle registers manually. For loop-carried dependencies, induction code is extracted and added before the loop body if they are not conditional. For other dependencies, they just spill the registers to memory, so that the HTM handles them. If this causes the loop to become slower, they deoptimize the code, falling back to the sequential version. They do additional spilling to ensure that after all transactions have ended, the last written value is in the registers. For this they spill all registers to thread-local memory. After all forked threads have finished, the original thread checks which thread executed the last loop iteration, and write its spilled memory back into the registers. To ensure the correctness of this approach, they demand that the HTM commits in order.

The authors have evaluated their approach, but only on simulated hardware, as existing hardware doesn't provide their required features. This makes their performance claims somewhat debatable.

2 Open questions

- The loop analyzer excludes loops containing “computed branches”. What exactly are those?
- How is the binary patched with the parallel code?
- Where do the lightweight forks come from? Are they part of the simulated HTM?
- The authors mention that they can handle “most” loop carried code by treating it as “induction code”. What are the cases that can't be handled by it?
- How does function inlining work on the binary level?

Summary T1

This week’s paper introduces an approach to parallelize legacy assembly code without access to the source code. They focus on the parallelization of loops with as little analysis as possible to avoid big performance pitfalls. In the approach a “re-compiler” is used which generates parallel code directly during runtime.

If a loop can be parallelized, there are 2 different approaches to do so: parallelize each individual iteration or dividing the iterations into tiles. Dividing a loop into tiles means that each transaction executes multiple iterations instead of putting each iteration into their own transaction.

The parallelization happens via forking off the parallel tasks. But instead of using a heavy-weight fork – meaning to also copy the stack frame – a light-weight fork is used. This again saves a bit of overhead to increase performance.

Using this method to parallelize loops containing function calls could cause problems as they increase the stack and could allocate stack-based variables. Instead of simply running such loops sequentially, they deal with this by in-lining such function calls which apparently solves this problem.

This system is completely experimental and in fact requires a *hardware* transactional memory system. To evaluate their approach they there simulate a HTM optimistically and only using 2 cores. This optimistic approach seems a bit unrealistic and they end up with speedups of near the maximum of 2.

Open questions:

1) How does in-lining prevent the problems of function calls inside loops?
2) How realistic is their optimistic simulation? What would “real” speedups be?

Runtime Parallelization of Legacy Code on a Transactional Memory System

1 Summary

The paper proposes an automatic parallelization framework which works on binary data (i.e. software where the source is not available). The authors assume the availability of hardware transactional memory to deal with data dependencies over memory. The scope of their work is the automatic parallelization of loops on instruction level.
They implement their system in a Just-In-Time compiler which will track the execution of the target application and try to recognize loops that are suitable for parallelization. These loops are detected by a two step analysis, the first step is a static analysis which finds natural loops that do not contain system calls or dynamic branching since these would be to hard to parallelize. The second step is a dynamic analysis implemented using hardware profiling, how these two steps fit together is not described, however the authors claim that the first step may be omitted. For the loops discovered the system will then measure the average execution time per iteration.
To parallelize a loop the JIT-compiler will insert code to fork one thread for each tile (group of iterations executed by a single thread). Each tile will be executed in its own transaction. The commits or aborts of the transactions are ordered by the order they were spawned in. If the parallel version does not perform better than the sequential one the code is reverted to the sequential loop.
Their system relies on HTM for handling data dependences over memory and a technique to remove data dependences over registers. Their strategy to achieve this will generate additional loop setup code for loop carried dependencies and will also generate spillage to memory if necessary. Additionally every thread will spill its register content to memory before committing so the main thread can acquire the register state of the thread that executed the last iteration.
In the end they perform an evaluation on the NAS and SPEC benchmarks which show a speedup between 1 and 2, the average being around 1.35.

2 Questions

- Do they actually parallelize loops which are executed only once? To perform their dynamic analysis the loop has to be executed once. They could parallelize these loops by saving profiling data and continuing from that information in later executions.

- When they describe their handling of break instructions, they state that an abort action will be inserted in that path, do they also abort transactions that started after the transaction that executed the break instruction?