Identifying Speculation Points

While thinking about the role of dynamic compilation in support of speculation, I was trying to come up with a generic view of speculation points within a program. I had assumed that speculation points would naturally be branches, but a small amount of thought shows this not to be the case.

This was obvious when I realised that **any instruction in a serial program can be a speculation point.**

Take the case of method (subroutine) speculation where the call is executed non speculatively and the code following the method, the continuation, is executed speculatively. What if we inlined the method? This should make no difference to the opportunity for speculation but we would now be dealing with a totally linear piece of code.

The answer is that, in any linear sequence of code, we can choose to jump ahead and speculate on any sequence of code beyond the current program counter. All we need to do is identify the speculation point S and the continuation point C. Consider the following sequence:

```
        Inst1
S       Inst2
        Inst3
        Inst4
C       Inst5
        Inst6
```

Assuming we are executing Inst1 and we come upon the speculation point at Inst2, we continue to execute Inst2 .. Inst 4 but start a speculative thread to execute the instructions beginning at C (Inst5) onwards. When the non-speculative thread reaches and has executed the instruction before C (Inst4), it is necessary to commit any non-speculative state and determine if the speculative thread can be made non-speculative and continue or whether it needs to be squashed and restarted. These are, of course, the normal rules that are used for both loop and method speculation.

Because HLL programs are structured, the S & C points are usually associated with control points in the program which reflect that structure and indicate points where speculation might be worthwhile but there is nothing fundamental about these.

Method Call

Here the speculation point is the call and the continuation point is the instruction immediately following.

```
        Inst1
S       BSR sub
C       Inst3
        Inst4
```

Of course, in this case, the subroutine body exists elsewhere, but if we consider it as logically inlined then we can see that it conforms to the general picture.

Recursion, both serial and parallel, is just a particular case of method speculation.

A Loop

Consider the following ‘do while’ loop. To make it simple, assume that the loop termination is determined by the single Tst instruction. Now consider applying the previous rules. We need an additional rule that a speculative thread gives rise to another (more) speculative thread. This is again a standard technique.

If we enter the loop from Inst1 we will execute Body1 & Body2 non-speculatively and start a speculative thread at the C point Tst. If the condition is true, we will jump back to execute a speculative copy of the loop body. However, because we encounter a speculation point S, we create a further speculative thread starting at C. This will continue until the loop termination condition is false when the speculative thread will continue to execute Inst6 etc..

```
        Inst1
S       Body1
        Body2
C       Tst
        Bcc (Body1)
        Inst6
```

Of course, in practice, we need to cope with any loop control variables. If, for example, the loop is controlled by a simple incremented counter immediately before Tst then this could be the continuation point. More complex loops might require transformation to make this simple model work sensibly. (exercise for the reader!).

We also need to deal with the situation where the non-speculative thread invalidates the speculative thread which follows it. This requires the well known rule that the squashing of a speculative thread must invalidate any further speculative threads which it created.

ILP Speculation

We ought to be able to express speculation, using this approach, down to the level where a thread is a single instruction. In that case we might hope that this becomes equivalent to speculative ILP as executed in superscalar processors.

```
        Inst1
S       Inst2
CS      Inst3
CS      Inst4
C       Inst5
        Inst6
```

This specifies that, on reaching Inst2 we immediately spawn Inst3, Inst4 & Inst5 speculatively in parallel.

The squashing rules must again deal with the multiple speculative threads but this corresponds to the normal mechanisms which exist in superscalar processors.

Nested Speculation

In the above ILP example (and all previous) we have assumed that the C corresponding to an S is the one that follows in sequence. However, this does not have to be the case. Consider the following:

```
        Inst1
S       Inst2
S       Inst3
C       Inst4
        Inst5
C       Inst6
```

The only sensible interpretation would seem to be that the S & C points are nested in a block structured way. On reaching Inst2 we would start a speculative thread at Inst6. The non-speculative thread, on reaching Inst3 would start a further speculative thread at Inst4.

We need to cope with the situation where the execution of Inst4 and Inst5 is squashed before Inst 5 is reached. In that case we must still examine the effect of the commit of the non-speculative part of the inner thread on that which started at Inst6. However, this is really no different from the previous cases of multiple speculative threads. The only additional rule we need to add is that inner threads are less speculative than outer ones.

Data Dependence

We suggested above that we can choose any instruction as a spawn point and any subsequent instruction as a continuation point. Does this make practical sense?

Consider firstly a simple linear sequence of instructions with no branches. What might determine how we would choose such points? Clearly there is the issue of granularity. Depending on hardware considerations we may choose smaller or larger groups of instructions. In the ILP case we may choose to speculate on the execution of single instructions.

The viability of this is determined in an ILP processor by the data dependences. There is no point in scheduling speculative execution of and instruction if its inputs depend on a previous instruction which is still being executed. In a superscalar processor, a combination of compiler analysis, instruction re-ordering and hardware interlocks are used to control both instruction issue and mis-speculation.

In general it is going to be a combination of granularity and data dependence which will determine the choice of sensible points in a more general scheme. In ILP we need to consider dependences caused by register usage. In a larger grain scheme intended for multi-threaded multi-processors we probably want to avoid dependences in a speculative thread on registers in a different core. This is a good reason why method boundaries are probably suitable points to consider; the register dependences are limited and can probably be eliminated in the thread spawning mechanism.

The data dependences via memory are usually either difficult or impossible to determine by static analysis depending on the nature of the program/ language. It is clear that some analysis of the probability of store carried data dependences are important in making choices of speculation points. However, it is important to realise that we have a wide choice of where these might be which is not directly related to the program structure.

More Complex Instruction Sequences

The above discussion assumed a linear sequence of instructions without branches. If we have branches, does this limit our choices?

Method Calls

Consider first the case of method calls. We have seen how, considering the call sequence as linear code, normal method speculation corresponds to choosing the speculation point as the call instruction and the continuation point as the immediately following one.

However, if we were to consider the code as logically extending into the method body, would it make any sense to choose a continuation point somewhere within this method body? Assume, to simplify the discussion, that the body itself contains no branches. We could consider the following:

```
        Inst1
S       BSR sub
        Inst3
        Inst4
        …

Sub     Sub1
C       Sub2
        Sub3
        Rts
```

Assume that the parameters are passed to the call on the stack (i.e. not using registers) at the point the call occurs. The return link would usually be in a register although it could in principle also be on the stack. If, when we started a speculative thread at point C, we forwarded the stack (and frame?) pointer and the return link to the thread, the continuation could access all the input data to the call. It might be necessary to set up a separate stack to hold data generated by the speculative thread, but this again is a detail. We could, of course, also pass parameters in registers as long as we forwarded them.

Assuming the continuation code was largely independent of the first part of the method, there is no reason why it should not continue. It could generate the method result, perform the return and continue (speculatively) with the code following the call. When the non-speculative thread reaches the continuation point, it may have produced values which invalidate the speculation. These could be in registers, on the stack or global. To avoid the register situation, we might want to compile the method body appropriately. The speculative thread would be squashed if necessary, but if not, it would be made non-speculative and allowed to continue.

It is not suggested that this approach would make sense in most circumstances, However, it does indicate that the principle is generally applicable and that it is only data dependences which determine correct behaviour.

An 'if' Statement

By avoiding branches in the code we have considered so far. We have always ensured that the speculative thread is executing code that will be required and will produce wanted results if it is not invalidated by data dependences.

However, if we are allowed to choose any instructions as speculation and continuation points this may not always be the case. Consider the following 'if' statement code:

```
        Inst1
S       Inst2
        Inst3
        Bcc (Inst8)
        Inst5
C       Inst6
        Inst7
        Inst8
```

Here we have chosen to begin a speculative thread before a test which controls execution of a block of code (Inst5..Inst7). however, we have chosen our continuation point within the block which may or may not get executed.

The situation here is slightly different. If the test fails, when executed by the non-speculative thread, the thread will never reach the continuation point. However, it is clear that the fact that the control of the non-speculative thread is about to progress beyond the continuation point is an indication that the non-speculative thread should commit and squash the speculative one. We could handle this as the fact that the program counter should be the address before the continuation point at the time the non-speculative thread commits. If the actual value were added to the data that the speculative thread checks at the commitment point this would handle the situation correctly.

One could continue to examine more complex sequences of branch instructions to analyse whether the principles work with general 'spaghetti' code and arbitrary spawn and continuation points. It is believed, but not proved, that they will.

Summary

It has been observed that thread level speculation can be achieved from serial code by choosing any instruction as a spawn point and any subsequent instruction as a continuation point. It has been shown that the common forms of TLS, namely method speculation and loop body speculation are particular cases of this principle where the points have been chosen to correspond with points in high level languages where the structure suggests that speculation may be profitable. It has further been shown that ILP speculation can be viewed in the same framework and that speculation can readily be nested with correct interpretation of the markings.

At execution time the following steps and principles must be followed:

- When control reaches a speculation point, the current thread should continue and a new speculative thread started at the continuation point.
- When a thread reaches and completes the instruction before the continuation point, if it is non-speculative it must commit its calculated values. Otherwise it must wait to become non-speculative.
- When a thread commits, the thread that it created will either be squashed or become non-speculative depending on whether the committing thread has made any of its input values invalid.
- Any squashed thread must also squash threads which it created.
- If the speculation is nested, the speculation and continuation points must be considered in a block structured manner. Outer speculations must be considered more speculative than inner ones and therefore squashed if they fail.
- If the continuation point is chosen within a method call (relative to the speculation point) the parameters and control variables (sp, fp & lr) need to be forwarded to the speculative thread.
- To handle situations where a continuation point is chosen within a section of code which would not have been executed in a normal serial schedule, it is necessary to modify the execution rules to squash the resulting speculative thread if the control point of a non-speculative thread is about to transfer beyond the continuation point.

Conclusions

Most TLS proposals concentrate on either method level or loop level speculative parallelism. This analysis has shown that there is nothing fundamental concerning this choice of control structure division. In principle, we can view the code at single instruction level and choose any instruction as a speculation point and any subsequent instruction as a continuation point. To make this principle work in all circumstances, it is necessary to add slightly to the normal rules of speculative execution which have been used previously.

By using this analysis, it may be possible to be more flexible in the generation of speculative parallelism. For example, compiler analysis might indicate that there are sections of a loop body which need to be executed serially before a speculative version of the next loop body is generated. The techniques described here show how this should be achievable by direct selection of sections of the loop body.