Embla design notes

Karl-Filip Faxén

May 30, 2007

1 Critical path measurement

Let's turn to static critical path measurement. We will have a dependency graph between instruction addresses (or possibly instruction data items which are proxy for the addresses). We will time stamp instructions based on earliest possible execution time. The time stamp of an instruction i is

$$\max_{j \in Pred(i)} [T(j) + L(j)]$$

where $Pred(j)$ are the instructions i depends on, $T(j)$ is the time stamp of j and $L(j)$ is the latency of j (this will be 1 for most instructions but for call instructions, it will be the critical path length of the call) We need to keep track of the time stamp of each static instruction (since the dependencies as described by $Pred(j)$ are static) for each open call. Probably best done with a hash table for each call (unless we can number the instructions statically so that they are consecutive for each procedure). Since the latency is not the same for each call instruction, we need to store that with its time stamp.

Branches are treated specially, for instance by including the time stamp of the latest conditional branch in the maximum above. This is one value for each open call.

We have a stack corresponding to the open calls. Each stack frame contains:

- A hash table mapping each static instruction to its time stamp (actually, the time stamp of its most recent instance).
- The maximum time stamp seen so far (relative to the call creating this frame).
- The time stamp of the call itself, to add to the relative time stamps otherwise describing the frame. This time stamp is in the caller's time domain.
- The time stamp of the most recent conditional branch. If we observe control dependencies, we use this time to ensure no later instruction gets an earlier stamp.

Possibly, we could have absolute time stamps everywhere. The issue is rather immaterial if we also keep the time stamp of the call itself.

The trace contains guest instructions, and we assume each has latency one. System calls are an issue, definitely. They should have a latency, but we have no way of knowing what. If however system time is small compared to user time, we will not be that much off by ignoring them. But system time tends to be at least a percentage point or so, which might be a lot compared to the length of the critical path. A possible conclusion is that we should do critical path measurement at system level using Simics!

We handle branches and calls specially, but otherwise we do not care what an instruction is. We just want the dependencies impinging on it.

The trace is composed of basic blocks as a space saving measure. We then need to extract one instruction at a time from the block, find its instruction address a, look up a in the static dependence graph to find the $Pred$ set and finally compute the time stamp, also considering control dependencies. Depending on the kind of instruction, we also do the following:

- If the instruction is a conditional branch, we update the latest conditional branch time.
- If the instruction is a call, we push a new frame on to the stack, initialising
  - the conditional branch time to the start time of the activation
  - the call time to the computed time stamp
  - the local time stamps to the start time (the hash table might be left empty, with the start time as default for the lookup operation)
  - the maximum time stamp to the start time.

  The start time is either zero, if we use relative times, or the call time, if we use absolute times (as I lean towards at the moment).
- If the instruction is a return, we pop the frame off of the stack and use the maximum time (time of return) to compute the call latency.

One issue that comes up since we do static dependencies is that some instructions in the $Pred$ set may not have been executed yet. But since we have a default value that is the start time of the frame, these instructions will not make any difference as their values will be less than some other arguments. This also points towards keeping $T(j)+L(j)$ in the table (rather than just $T(j)$); since the latency of a call instruction depends on the function called, we would otherwise need two items in the table.

This discussion points at a representation of instructions in the trace as pointers to info records where the record distinguishes calls, returns, branches and everything else. We can get this information simply by looking at the jumpkind of the exit in the instruction. If there are multiple exits in the same guest instruction, I do not know what to do!

2 Register dependencies

We need to avoid dependencies on registers that are part of the calling convention, such as `ESP` and `EBP` in x86 (IA32). But we do not know for sure if `EBP` is part of the calling convention or used to hold ordinary values.

2.1 Subword memory accesses

Registers are represented using a dependency value block which has the same format as the real guest state. There is an issue with partial word access, just as there is with memory. Solutions:

- Round the offset to a block size, for instance 4 bytes, and do the analysis at that granularity. We'll loose precision for subword accesses, though (aliasing).
- Do separate analysis for each byte, looping over the bytes in a word access. Since we are only interested in existance of dependencies, we need not dwell over the number of dependencies reported.

The second alternative might be optimized. If we assume that in most cases a read to an offset d is of the same size as the last write we do not need to access the other bytes in the subword. So we need to know the size of the previous write. We also need to disable access to offset $d+1, \ldots, d+k-1$ where k is the size in bytes of the write.

Tentative solution: for each byte address, keep track of the size of the last access.

- If the size of the current access matches, just use it.
- If the current access is wider, and is
  - a read, access the rest of the info about the current access and build all of the dependencies
  - a write, disable the subsequent bytes up to the new size and change the size
- If the current access is narrower, and is
  - a read, just treat it as if the sizes had matched
  - a write, need to split the words

Then there is the problem of accesses that hit a disabled part of the word. Must find the part that describes the access.

One way to look at this problem is to view a word access as composed of a number of byte accesses and look at the set of dependence edges generated.

Independent of the implementation, we need to decide on the semantics of subword access. As long as an address is accessed in a nonoverlapping manner, nothing changes and we can handle it as before. What we want to ensure is that the reported dependencies are exactly as if we kept the info for each byte. What we can do is to keep info in per word form with a length indication.

- If we have a subword access, we simply have to clone the info to the level of granularity of the access.
- If we have a superword access (the location is already split), and
  - the access is a read (which does not clear the dependency state) we have to make the change in all subwords touched
  - the access is a write (which clears dependency state) we fuse the subwords and then do the update.

This means that we have to know which case it is, hence for each byte we record the following info:

- Whether the address is an access unit
- If it is an access unit,
  - what size of access in bytes
  - the access state (last write and reads since last write, or just last write for registers).

  If it is not an access unit,

  - where the access unit is (how many bytes back).

By design, an access unit is always naturally aligned.

There are several algorithms involved in an access. Here is the one for splitting an access unit (word) into subwords of a given size `size`:

```
void splitAccessUnit(Item *item, int size)
{
    int i,j;

    if( item->size > size ) {
        for( i=size; i<item->size; i+=size ) {
            item[i].isAccessUnit = TRUE;
            item[i].size = size;
            item[i].state = copy( item[0].state );
            for( j=1; j<size; j++ ) {
                item[i+j].accessUnitOffset = j;
            }
         }
         item->size = size;
     }
}
```

Sometimes we merge rather than split:

```
void mergeAccessUnit(Item *item, int size)
{
    int i;
    for( i=1; i<size; i++ ) {
        item[i].accessUnitOffset = i;
        item[i].isAccessUnit = FALSE;
        delete( item[i].state );
    }
}
```

Thus the algorithm for an access is somewhat like the following:

```
item = lookup( addr );
if( item->isAccessUnit == TRUE && item->size == size ) {
    return item->state; // fast path
} else {
    // determine splitting needs
    // check if the current access is naturally aligned
    if( addr%size != 0 ) {
        splitSize = 1;
    } else {
        splitSize = size;
    }
    if( item->isAccessUnit != TRUE || item->size > size || splitSize != size) {
        // we need to split
        offset = item->accessUnitOffset;
        origAddr = addr - offset;
        if( item->isAccessUnit != TRUE ) {
             // we did not hit an access unit
             item = lookup( addr - offset );
             addr = addr - offset;
        }
        do {
            splitAccessUnit( item, splitSize );
            addr = addr + item->size;
            item = lookup( addr );
        } while ( addr < origAddr+size );
    } else {
        // only correct for write access, for read, access those on the list
        mergeAccessUnit( item, size );
    }
```

Another idea is to keep everything in one place as long as the accesses are aligned. Thus all aligned accesses go in their natural item, but some items are not marked access unit. Question is, do we need to look in the subwords as well when we do not find it an access unit? Possible answer:

- If it's an acess unit, the item holds all info about it.
- Otherwise, we also need to look in
  - all subwords (they will not be access units either)
  - enclosing words until we find an access unit.

So we have a recursive structure (like a buddy system) where a block is either split or not. Suppose we have an aligned access to address a with width w, then we have to process as follows:

```
subWords(Item *item, int size)
{
    if( ! item->isAccessUnit ) {
        process( item+(size/2) );
        subWords( item, size/2 );
        subWords( item + size/2, size/2 );
    }
}

access(Item *item, int size)
{
    process( item );
    subWords( item, size );
    while( ! item->isAccessUnit ) {
        item = parent( item );
        process( item );
    }
}
```

This does not work due to the uncertainty as to what an access unit is (the item represents itself and the leftmost branch rooted at it, which does not really work). Think of this tree in a bottom-upish kind of way instead: Two buddy leaf items either have their own info (both bytes accessed) or

All accesses are managed in terms of aligned blocks. An aligned block is managed at its lowest addressed byte. If the access unit flag is set, there is no need to check subwords and no need to check superwords. The largest access unit is 8 bytes. This gives the following lay outs:

1. Two bytes, address a and $a + 1$.
2. Four bytes, addresses $a, \ldots, a + 3$.
3. Eight bytes, addresses $a, \ldots, a + 7$.

We may have info for an address divisible by eight about several accesses with different size, some of which might have been partially superceeded by subwords. So if we have such an address with is marked as an access unit, we do not know what that means. Is it a byte access unit, a word access unit, a longword access unit or a doubleword access unit?

So if we again look at the problem, but from the point of view of write and read access info, we get (for read accesses) the following: Each byte item holds a list of accesses, each with a size. Address a only holds accesses of widths that divide a. If we think about an access to an address a and width w we have the information at a and the subwords $a+1, \ldots, a+w-1$, as well as each parent of a. Address a' is a parent of address a if there is a width $w' \leq 8$ such that w' divides a' and $a = a' + w'/2$. The immediate parent of a is the largest such a'.

Suppose we define the latest writes $W(a)$ of an address a as

$$W(a) = \bigcup_{a' \in Parent(a)} \{u \mid u \in w(a) \wedge L(u) > a - a'\}$$

which implies that for an access to address a with size l we want conflicts with $W(a) \cup \ldots \cup W(a+l-1)$. Now, this can be simplified to $W(a) \cup w(a+1) \cup \ldots \cup w(a+l-1)$.

An even more abstract version is this: Let W be a set of write operations (essentially trace references with sizes). Then the last write to a is the last write w such that $A(w) \leq a < A(w) + L(w)$. Again, an access with length l conflicts with all writes which go to any of the l bytes in the access, that is all w such that $A(w) \leq a + l$ and $a < A(w) + L(w)$.

2.2 Finding the frame pointer

2.2.1 Save restore based dependence suppression

We also have save/restore of registers (including the frame pointer) accross function calls. For caller save there are not many problems, but callee save poses a problem since the called function comes to depend on the last prior definition of the register (because of the save in the callee) while subsequent uses of the register will depend on the call (due to the restore). Hence we should avoid these saves and restores when computing dependencies.

- Define a *save* to be a store of a live-in register to a slot in the current stack frame.
- Define a *restore* to be a load of a register from the place where it was most recently saved.

This gives rise to the following needs:

- Keep track of registers and their values (well, whether they have been written since the call). This boils down to looking at `Put` statements.
- Keep track of which locations are in the current stack frame. The current stack frame extends from the value of the stack pointer at or immediately after the call to a few words beyond the current value of the stack pointer (because we push the value before we update `ESP`).

- For locations in the current stack frame, keep track of certain values:
  - Which register is saved there.
  - What the last write to that register was (we only do flow dependencies for registers).
- Recognizing saves and restores which are
  - `Get` followed by `Store` where the register is not overwritten and the store is in the stack frame, and
  - `Load` followed by `Put` where the loaded value is saved (not changed since the write) and is put back in the same register as it was before the save.

Recognizing saves This is a matter of keeping track of which registers have been written since the last function call. This could be implemented by a bit vector. The indexed access forms are a complicating feature, but it can be handled. The question is whether we can use the `GetI` and `PutI` constructs to access it since they appear to have the guest state as an implicit operand. Their effect is however documented, so we can use their operands to compute what we need. The expression for the base byte touched by the access (the first one out of s) is $o + s \times ((i + b) \backslash n)$.

Ok, suppose we get zero help from the Valgrind infrastructure. Then we generate our own optimized addressing code for indexed accesses. We only use the size of the guest state to allocate a suitable guest state table which we use. Sigh.

So we need, for each byte of guest state and stack frame:

- 1 bit saying whether the byte has been updated.
- The address in the stack frame to which it has been saved.
- The last write before the call (a pointer to a trace record).

This means we do not have to check the addresses of every access but only live-in reads of every block. Actually, we can check whether a register is written since the call simply by comparing the trace record pointer to the trace record of the most recent call. Similarly, we can see that it has not been overwritten by checking that it is still before the most recent call. Well, maybe not. Suppose we have a tag per memory block (byte) which says that this is a saved copy of a specific live-in register.

Simple solution: Have a hash table called the *save lookaside buffer* containing (for a subset of stack memory locations):

- Whether the location contains a saved guest register and if so which one (offset in guest state + size).

- The last write to that guest register prior to the save (a pointer to a trace record).
- A pointer to the dependence edge to mark regular if the entry is invalidated.

We have the following implementations of common operations:

- For a write to the stack of a live-in guest register: Generate a dependence, but mark it as a potential save operation, with the marking to be invalidated if it was not a save. Allocate a save lookaside buffer entry. Fill in the guest state offset and the last write info from the guest state table in the new entry.
- For other writes: Check the save lookaside buffer and invalidate any entry found. If an invalidated entry has not been out of the stack, mark the dependence associated with the entry as a regular dependence edge (not a save edge).
- For a read: Check the save lookaside buffer. If we have a hit (search by address) and the guest state offset and size is the same in the access and in the save lookaside buffer entry, omit generating the dependence (or generate it and mark it as a potential save). Update the guest state table with the trace record pointer from the entry.

2.3 Computing register dependencies

Some of the computation of dependencies should be done at instrumentation-time. When the instrumentation routine is called, we know that it is going to be executed at least once, so we can record all dependencies that are guaranteed to be always the same. This includes the dependencies involving (guest) registers. When computing critical path length, the situation is a bit different than when computing dependencies, at least for the fully dynamic critical path computation.

Anyway, when starting instrumentation of a new block, the guest state table is initialized to all **U**. When we compute dependencies or time stamps, we check the guest state table for `Get` and `GetI` which either points at the instruction in the block that last wrote to that part of the state or has the value **U** if there is no such instruction. In that case we emit code that checks the shadow state at run-time. Well, that's not entirely true: If we are just after static dependencies, we record these in the static dependence area at instrumentation time if the value is not **U**. If we want dynamic critical path, each execution will yield different time stamps so we need to generate code looking at the instruction doing the last write and its time stamp.

2.4 The dynamic analysis

So add a payload struct to the trace records, for any extra information. Question is how `gcc` handles that? When the struct is empty? Seems to work as expected, so we can leave the payload struct in the trace record and let it be empty if we do not want any payload.

In the case of critical path measurement, the payload struct contains the earliest possible execution time, relative to the parent call (I think). So to fill in a new trace record we compute the time stamp as a suitable function of the time stamps of the instructions it depends on. This means all of the dependencies of any IR statement in the instruction, that is between the `IMark` of the current instruction and that of the next one. How do we find these dependencies? We have to look at each statement and find their embedded expressions.

NoOp Needs no work

IMark A new instruction begins; we should wrap up the story of the previous instruction (if there is one) and initialize the state to that of the new one.

Put A store to a register. We need to update the register table to show that the register was written by the current instruction. Actually, if we do not turn on optimization in Valgrind, we need only consider the `Get` and `Put` operations as generating interesting dependencies (ok, well, `Load` also).

PutI Sigh. Need to figure out what this one does. But I guess pretty much the same as for `Put` applies.

Tmp Well, this updates the temp table (if we have one) with a dependence to whatever the expression depends on. That is, the temp table maps temps to the instruction that last updated them (and that's the only update, of course). But does this mean that this instruction depends on the expression part of the `Tmp`? If IR statements are moved around between instructions we really do not know. So let's assume for the time being that we continue using an unoptimized Valgrind.

Store Depends on both of the expression and the address.

Dirty I have no idea. Does this encode syscalls? Should be handled in that case...

MFence I do not think we need to handle that. Why would we?

When it comes to expressions, there is where we really do the dependencies. If we do not use pre-instrumentation optimization in Valgrind, only the guest state and memory is live between guest instructions, so we need only check those. Actually, this makes for rather simple instrumentation: `Put`, `PutI` and `Store` update state while `Get`, `GetI` and `Load` determine the value to use for the update. If `Load` is counted as zero latency, we probably do not need to do anything special for EBP. Actually, for that it is enough if the latency is zero for loads from the stack.

A Get of ESP generates no dependence, and neither does a Get of EBP if it is marked as a frame pointer. The condition codes must also be tracked. All registers have renaming, so we only do flow (RAW) dependencies.

2.5 The static dependence calculation

In the static case, we are very close to standard Embla. The main differences are:

- Dependencies are between instruction addresses, not between lines. Computations of hidden functions still use symbols, though, so we need the line structures as well. But we also need the instruction addresses, or the addresses of the instruction structures.
- Hidden and false dependencies are not reported at all.
- We do not care about the number of dependencies, just that they are there (or not).
- We do not report flow, anti and output dependencies. As far as registers are concerned, we only ever count flow. For the stack, I'm not really sure. Maybe we should only do flow here as well? For other memory areas, all three kinds count but they are just dependencies.

We have a new design of the dependence table since we want to query it with the dependent instruction address. Thus, for each instruction it contains a list of instructions it depends on. Here are some data type definitions:

```
typedef struct _AddrList {
    Addr32            addr;
    struct _AddrList *next;
} AddrList;
typedef struct _PredItem {
    Addr32            dest;
    AddrList         *srces;
    struct _PredItem *next;
} PredItem;
```

This appears to be what we need. We then make a big array of *PredItem and index with low order bits of instruction address. An array size of 2^{20} or so would probably do nicely.

3 Instruction traces

We compute the static instruction level dependence graph using register and memory dependencies. We then use it together with an instruction trace generator to compute critical path. Hmm, this could be modularized if the critical path was measured an instruction at a time. That is, the trace generator is either an on line algorithm simulating the program's execution, or an offline algorithm just reading a trace (which could be compressed). Hmm, this brings us to compressed instruction trace representation. For a start, here are a number of alternatives:

1. One trace record for each instruction.
2. One trace record for each branch, together with a mapping of blocks to addresses. That is, each superblock (unit of instrumentation) and each part of a superblock following a conditional exit has a unique block number, and those are given in the trace.
3. A branch level mapping, but with repetitions collapsed. So some block numbers refer to sequences of blocks.

Probably, we could do quite compact representations, perhaps compact enough to keep in core. In most cases, one block would always be followed by one of two or so other blocks, in many cases with a considerable skew. So we could fill a linear buffer with a few million blocks and then stop an do statistics on it. But that would give long pauses, so it might be better to do it on line.

3.1 Simplest instruction tracer

The simplest possible design: Each time we instrument an `IMark`, we emit code to put the address of the instruction (available in the `IMark`) in a buffer. If the buffer gets full, we weep. When the instrumented execution is finished, we traverse the buffer, finding addresses that we use to look up the dependency table. Possible refinements are legio:

3.1.1 Basic blocks

Per basic block, instruction descriptors (or even instruction numbers). In particular, we might look at the following: Each time a block is instrumented we make a block descriptor containing the addresses of the instructions in the block (or addresses of their descriptors). We emit code to record the address (or serial number) of the block descriptor. When we unpack the trace, we find the block descriptors by dereferencing or indexing and loop through the instructions in the block. In fact, since blocks contain straight line code, it is enough to record the address of the first instruction, the number of instructions and their lengths. It makes sense to have a fixed block descriptor format, with type as follows:

```
typedef struct {
    Addr32        first;
    unsigned char n_ins;
    unsigned char length[7];
}
```

This struct is 12 bytes in size and encodes up to 8 instructions. We would also need some flag bits telling us if the instructions are in the Conditional, Call, Return or Other categories.

3.1.2 Recency rank

Since most of the blocks are likely to be close together, we might want to use *recency rank* to represent them. In that case, we use a small number to refer to each of the last n block numbers. In that case we need to keep track of these in a table, which is perhaps a little onerous.

3.1.3 Relative addressing

We can record the difference between a block and the previous block. This would work best with block addresses or indices, not with the addresses of block descriptors (or, that would work as well, if the block descriptors were consecutively allocated).

3.1.4 On line trace compression

Build a tree with all repetitions of blocks ever seen.

1. When a block is seen for the first time, it is entered in the root level of the table and given a block number.
2. If a block is already seen, it is in the root level and we replace it with its block number. We now look at the level following the block. Fetch next block and see if it is in that level, in that case follow the link. When a new block is not found, output the current node number and its address and add the block as a follower in the tree (make a new leaf).

This algorithm has the problem that the tree will grow very much, especially when we see loops with long trip counts. Then the tree will be of size $\sqrt{n}$ where n is the trip count of the loop.

Another approach would be based on grammars. Suppose that we build a grammar looking like:

$$\begin{array}{ll} A_1 & \rightarrow A_j \; a_k \\ & \ldots \\ A_n & \rightarrow A_i \; a_l \end{array}$$

Then we have A_0 which expands to the empty string and if the first block is a_0 make the rule $A_1 \rightarrow A_0 \; a_0$. Then we get a_1 which may or may not be equal to a_0; if it is, we do ...

No, this is no better.

If instead we think about what we would do with the entire string available, we would probably look for repetitions, for instance starting with long ones. The benefit of one repetition would be computed using the cost without and the cost with repetition sharing. Without, if the repetition is l symbols long and occurs n times, the cost is ln whereas with sharing the cost is more like $l + n$. Hence the compression factor is:

$$\frac{l+n}{ln} = \frac{1+\frac{n}{l}}{n} = \frac{1}{n} + \frac{1}{l}$$

So we simply want large n and l, and we kind of like to make them equally big.

Actually, starting with short repetitions is simpler. We make statistics about symbol pairs, then join the most common pair. We also look for repetitions of the same symbol (before looking for pairs). When we have joined a pair, we have to adjust other counts: If the pair is ab and it has n occurrences, we have to subtract 1 from the occurrence count of xa for each xab in the string and similarly for by and aby. There is a potential problem in that we want to find all occurrences of ab in order to do the substitution, but this entails a nonlinear amount of work or a large overhead in terms of space to maintain lists of occurrences. There is also a problem with $abab$ since we should replace ab with z which entails updating the occurrence counts of bz and za. But we should not do that since we in fact will not have zab or abz since both occurrences are replaced. In that case we might keep track of the number of times each pair is followed by the same pair. So we keep track of number of occurrences and number of gaps. If there are no back-to-back occurrences there will be $n - 1$ gaps. Suppose there are $m < n - 1$ gaps. Then there are $n - 1 - m$ back to back occurrences of the pair.

But we still have problems. Now we rely on gap statistics, but I am not sure we get gap statistics from just the gap statistics of the replaced pair.

We must look at all occurrences anyway. Here's why: If we join the pair ab we need to effectively change xab to xz and not all occurrences of xa are followed by b. So we *do* need to look at all occurrences. Not good.

What importance has the order in which we do the joinings? If we join ab, then what other joinings are affected? Well, there will be less occurrences of xa and by of course.

Another compression idea: Suppose we only look at non repeating sequences of symbols, with the idea that this will give us long sequences of identical (derived) symbols that can be compressed. Or we simply make do with Lempel-Ziv.

Anyway, we assume that we will find the following features in the input:

- Repetitions of the same symbol due to loops with basic block bodies or recursive procedures.
- Repetitions (or almost repetitions) of (short) symbol sequences due to loops with conditionals in their bodies or recursive procedures.
- Repetitions of (similar) patterns due to (non recursive) procedure calls.

One question to ask is: Given a symbol, what are the frequencies of strings following that symbol?

3.2 Call and return marking

We want to mark instructions in matching call/return pairs to help the path length computation. In general, it is only when we have made the return that we know if we count something as a call or not. When seeing a return, we might find that it matches zero or more calls.

What if a call is not a true call? A true call has two effects on scheduling (and thus critical path length):

1. No instruction belonging to the call can start before the call (ie before all instructions belonging to the call).
2. Instructions in the caller can start before all conditional branches in the callee have been executed.

So one effect of calls is negative and the other positive. Since non matching calls/returns tend to happen infrequently, they probably have a minor effect on the critical path length.

Tentative solution: Insert into the trace one copy of the return instruction for each stack frame popped. This means that a return can be elided, or that there are several instructions with the same address in the trace. Probably, there will be no problems. However, the basic block tracing appears to be problematic as the number of instructions in the basic block might vary. One possibility is to exclude the return, which is always the last instruction in the block, and make a special block with just that instruction. Then that block can be emitted a suitable number of times (from the code recording the return).

3.3 Trace based analysis

Most measurements of parallellism are based on traces, and variations on analyses are based on traversing the traces in different ways. Our procedure level parallelization can be defined as having the following additional dependencies in addition to the pure data dependencies:

- From a `CALL` record to all subsequent records up to and including the matching `RET` record.
- To each `RET` record from all earlier records up to and including the matching `CALL` record.

There is also a modification as to how contol dependencies are handled in that no control flow dependence edge crosses and unmatched `RET`. In practice, there is no need for a control flow dependence edge to pass a `CALL` either since the `CALL` already depends on the source of the edge and everything thereafter depends on the `CALL`.

Types of dependencies are as folows:

- Data RAW. True dependencies.

- Data WAR and WAW. Storage dependencies. Based on
  - Registers. Easy to avoid by renaming.
  - Scalar stack locations. Also possible to avoid by renaming.
  - Other memory locations.
- Contol dependencies. Comes in two flavours:
  - From a contitional branch to subsequent control dependent instructions.
  - From a conditional branch to subsequent instructions up to an unmatched `RET`.
  - From a conditional branch to any subsequent instructions.
- Structural dependencies.
  - From a `CALL` record to all subsequent records up to and including the matching `RET` record.
  - To each `RET` record from all earlier records up to and including the matching `CALL` record.

For the data dependencies (both true and storage) we can have static dependencies or dynamic dependencies. Having static dependencies entails the following transformation on the dependency graph: Let the trace have records numbered 1 to N. For each trace index i, $A(i)$ is the address of the instruction executed at place i in the trace. Then, if there is a dependence $i \rightarrow j$ in the dependence graph, then add dependencies between all pairs k, l such that $k < l$, $A(k) = A(i)$ and $A(l) = A(j)$. Does that give what we want? We want something that captures the idea that we can onlymake one version of the program so if there is a dependence betwenn two instructions at one time, we have to assume it is always there. But the main ide a is to have this rule after the mapping to NCAs. But can we do it before? Does it mean the same thing? Probably not.

A Junk

A.1 Register dependencies

Let us use the following rules:

- `ESP` is always defined at time 0.
- A register set to `ESP`+0 is always defined at time 0.
- A register set by the `LEAVE` instruction is always defined at time 0.

The trouble with this definition is that we do not (easily) know when we execute the `LEAVE` instruction (since that is part of x86 instruction set while what we see is the Valgrind IR). So what does that instruction do? It copies the `EBP` to `ESP` and then pops a new value for `EBP` off the stack, where it has been placed by a `PUSHL` in the procedure prologue. If we keep track of stack pointer value, we can see that the location `PUSHL` stores `EBP` in is new to the stack, so its deftime only depends on the deftime of the other operands of `PUSHL`. These are the stack pointer (always at deftime 0) and the old frame pointer (presumably also at deftime 0).

A simpler alternative: regard all memory access as having 0 latency, on the grounds that they might be avoided. Or at least do it for references to the stack.

Another thought is that we might not want to count latency in this way. Maybe we are only interested in whether two procedure calls can proceed in parallel. In that case we might simply count the number of guest instructions along the dependence path (to know where the calls started). We could look at

```
call foo
instr_1
...
instr_n
call bar
```

and try to figure out if the two calls are dependent. So we mark each instruction with whether it depends on the previous call. But in general, there may of course be several calls, so we could keep track of a list of calls in general. Question is, where do we start? That is, the second call depends on some of the instructions between the calls. Do they get moved to before the first call?

No, because what we do is we see if we can push the return event back towards later time.

No again. Let's simply try to assign times to all events (possibly not assuming instruction level parallelism). In the long run we want to use static dependencies, in which case we have to keep track of earliest times of instructions rather than storage locations (the dependence information must take the form of a graph per procedure where the nodes are instructions and the edges represent dependencies). So if we continue to use the Embla approach we have instruction execution events in the memory table, but we should add time information to the trace records. More fun when compacting...

A.1.1 Frame pointer identification

We can use the following rules for frame pointer calculation: If a register is a copy of the stack pointer or a restored frame pointer value, it is a frame pointer. This rule means that the frame pointer starts out frame pointer after every call using one and that it will remain frame pointer when restored. Thus we need to do the following to keep this info updated:

- During translation, keep track of `Temps` that have the same value as the stack pointer. That is, look for `Get` operations with the `ESP` index as well as copy operations. Do the same for `EBP`.

- During translation, keep track of `Store` statements with data equal to `EBP` and address falling in the stack. Mark the address as containing a saved frame pointer.

- At run time, keep track of memory locations storing the frame pointer. We need to unmark those that are overwritten, so it would be best if the info was stored together with everything else in the memory table. Otherwise we would need to handle that special case every time.

- During translation, keep track of `Load` operations targeting `EBP`. Lay out code to check if the value loaded was a saved frame pointer. For instance, by looking up the address in the memory table.

Changes needed to the code include:

- Give an additional argument to the store instrumentation routine specifying whether the stored data is a frame pointer. Also add argument to the recording routine, if there is room. Or create a special version.

- Introduce a "frame pointer tag" in the `Event` words. Is there space in there for another tag bit? I think we currently use just one, telling if the event was hidden, so one is unused.

We then suppress dependencies between instructions updating the frame or stack pointers and instructions using them. The latter include all the stack manipulating instructions.