The Architecture of an Optimistic CPU: The WarpEngine.

John G. Cleary†, Murray Pearson†, Husam Kinawi§

†Dept. of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand.
{jcleary, mpearson}@waikato.ac.nz

§Dept. of Computer Science, University of Calgary, Calgary, Alberta T2N 1N4, Canada.
kinawi@cpsc.ucalgary.ca

Abstract

The architecture for a shared memory CPU is described. The CPU allows for parallelism down to the level of single instructions and is tolerant of memory latency. All executable instructions and memory accesses are time stamped. The TimeWarp algorithm is used for managing synchronisation. This algorithm is optimistic and requires that all computations can be rolled back. The basic functions required for implementing the control and memory system used by TimeWarp are described. The memory model presented to the programmer is a single linear address space modified by a single thread of comtrol. Thus, at the software level there is no need for explicit synchronising actions when accessing memory. The physical implementation, however, is multiple CPUs with their own caches and local memory with each CPU simultaneously executing multiple threads of control.

1. Introduction

Computer designers currently face a very interesting set of challenges. The steady increase in the number of transistors on a chip and the speed at which a chip can be clocked has continued its inexorable progress. In 1994, chips with 3×10^6 transistors and clock speeds of hundreds of MHz are in routine production and there are no short-term barriers to these numbers continuing to double every three years into the foreseeable future [5]. However, off-chip and memory speeds have not increased to nearly the same extent [12]. This has lead to a situation where system performance is constrained by the latency of memory fetches and off-chip interactions. One response to this has been to achieve additional performance through the use of parallelism extracted at two levels.

Within a single chip, pipelining and superscalar execution techniques are used. These techniques have lead to increasingly complex and baroque CPU architectures where increases in performance require changes to the architecture. The second approach is to use many CPUs in a single system. Initially, such systems were configured with disjoint memory where communication between CPUs required explicit calls in software to send messages. More recently, shared memory systems, where the communication is effectively handled by the memory system, have become commercially available [3, 17, 22]. In these systems the user is presented with a single shared linear address space which can be seen by all CPUs. The underlying hardware implementation of such systems can vary markedly, from systems with single shared buses [3] to asynchronous systems where shared memory pages are communicated by hardware messages [1, 11, 22]. Shared memory systems are significantly harder to build than distributed memory systems because of the need to deal in hardware with remote memory accesses and cache coherence. However, they have the great advantage that they remove from the programmer the burden of partitioning data-structures to spread them across multiple disjoint address spaces.

Programming (and achieving good performance) on shared memory systems remains a significantly harder task than programming on sequential processors. This is because two significant tasks are still left to the programmer. These are ensuring that access to shared memory locations is synchronised (by locks, semaphores etc.) and decomposing the code into minimally communicating units that are to execute in parallel.

This paper presents an architecture that addresses these problems. The virtual view seen by the programmer is a single address space operated on by a single thread of control. The intended physical implementation is multiple CPUs with their own local memory operating on many parallel threads of control. Parallelism is provided at three levels. At the first level, within a single CPU, individual instructions can be executed by functional units that execute in parallel, synchronised by a data-flow protocol. The number of functional units per CPU can be scaled to fit hardware performance parameters rather than redesigning the architecture. A second level of parallelism organises instructions into blocks (of up to 16 instructions). A single CPU may simultaneously execute many blocks. The third level of parallelism has multiple CPUs executing different blocks of instructions.

Synchronisation is achieved (at the hardware level) by assigning a unique “time stamp” to each execution instance of a block. Reads and writes to memory refer to a time stamp as well as to an address. The overall co-ordination of these time stamps uses the TimeWarp algorithm which is introduced in section 2. Section 3 then describes how TimeWarp can be integrated into a memory system and how time stamps are generated for the execution blocks. The rest of the paper concentrates on the internal architecture of the CPU including its instruction set. The paper concludes with order of magnitude estimates of performance and a discussion of future work.

2. TimeWarp

TimeWarp is an algorithm for distributing and parallelising discrete event simulations. The algorithm was originally proposed by Jefferson in 1982 [7, 14, 15, 24].

Parallelising discrete event simulations is a difficult problem because of the need to synchronise the simulation times of different simulation objects without violating the principle of causality [8]. The TimeWarp algorithm solves this by executing events optimistically and later, if necessary, rolling-back (undoing) events that should not in fact have been executed. A simulation executed using TimeWarp is usually decomposed as a series of objects which communicate by passing messages. In simulation terms, these messages can be thought of as scheduling events which are specified to occur at a particular receiving object at a particular time.

To rollback the execution of an event two things must be accomplished: restore the state of the system; and undo the effect of any messages sent by the event. This latter is accomplished by sending anti-messages following the earlier messages. When an anti-message is received, it can cause two effects. If the original message had not as yet been executed (it is queued somewhere waiting to be scheduled) then it is annihilated and no further action is taken. If the original message had been executed then it will be rolledback (with possible transmission of further anti-messages), the antimessage and message will “annihilate” and execution will resume as if the original message (and anti-message) had never been received.

A common technique for restoring the state of objects on a rollback is to take a copy of the object's state each time it receives a message (much more efficient techniques are possible but do not concern us here). Then when a rollback occurs, the appropriate previous state can be copied back into the object. One side effect of this is that each object builds up a queue of old state copies. It is hence necessary to eventually garbage collect these old state copies (this is referred to as “fossil” collection). This can only be done when it is known that a state copy will never be required for a rollback. This is done by periodically computing a value called the GVT (Global Virtual Time) which is equal to the minimum time of any currently active object or message between objects. It is guaranteed by the TimeWarp mechanism that no object will rollback to a virtual time prior to GVT, and hence it is possible to fossil collect all state copies with time stamps less than GVT. GVT is also used to “commit” other interactions with the outside world. For example, a request to print cannot be honoured until GVT passes the time that the request was made.

Results about the performance of TimeWarp are surveyed in [8]. Good performance by TimeWarp relies on a number of factors. First, the cascades of anti-messages induced by rollbacks must damp out quickly and not continue building. The overheads required to restore state on a rollback also cannot be too expensive. In practice, this means that the cost of recording state changes during forward execution must be low. Given all of this, TimeWarp is capable of parallelising and speeding up problems that are otherwise intractable. Fujimoto [8] concludes that TimeWarp is the only algorithm known which is capable of robustly speeding up a wide range of discrete event simulation problems but that this is contingent on good implementations with low overheads.

It has been recognised ever since the algorithm was originally proposed that TimeWarp is not just an algorithm for parallelising discrete event simulations but is also a general purpose technique for synchronising parallel computation. For example, it can be used as an undo mechanism in editors [23], consistent checkpointing for fault tolerance, incremental recovery in data bases, selective undo in groupware tools [21], or debugging for parallel and distributed programs [4]. In this context it can be seen as a generalisation of optimistic commit protocols. It has also been described as an algorithm for synchronising AND-parallel Prolog [20]. Fujimoto has proposed a Virtual Time Machine [6, 9] for use with discrete event simulation systems. Fujimoto’s work is seminal for the current WarpEngine design.

3. Time-stamped Memory

The first component of the WarpEngine that we will describe is the time stamped memory subsystem for a single CPU. The memory subsystem receives from the CPU read and write request messages and sends back reply messages in response to the reads.

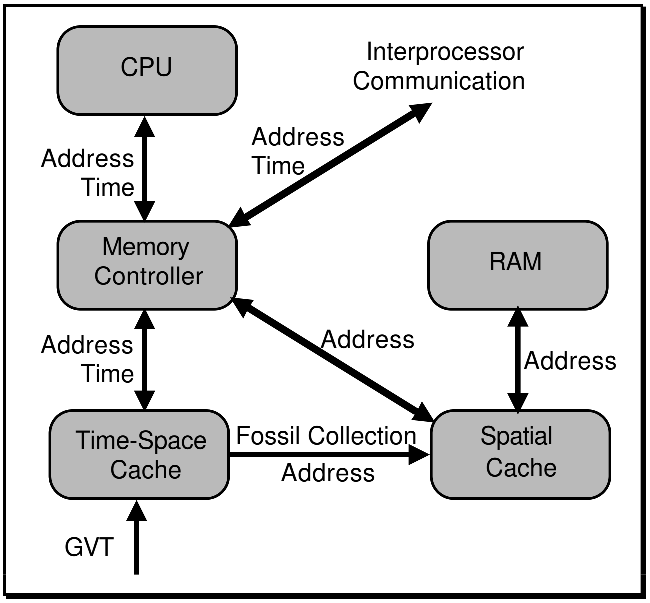


Figure 3.1. Components of the Memory Subsystem.

Each read is accompanied by an address, a time stamp and a tag (which is used by the CPU for identifying which read request is being satisfied when the memory subsystem finally replies). We will see in the next section how the time stamps are generated, but for now it is sufficient that the time stamps impose a single linear temporal order relating all the reads and writes ie., a memory access issued before another in program order would be time stamped with an earlier time. Similarly, a write is accompanied by an address, a time stamp and a value. The value returned for a read is the value from the last write prior to the read (in time stamp order) for that address. Note that we are assuming an optimistic system here, so that writes may be physically generated in a different order from their time stamps.

As a consequence of using TimeWarp, the CPU may issue an "anti-write", this specifies a time stamp and address. Its effect is to delete the original write. As we will see below it may require some read requests to be re-satisfied.

3.1. Implementation of the Time-stamped Memory

A naive implementation of such a scheme would clearly not be sufficiently efficient. The scheme we propose has a three level memory system. At the level closest to the CPU there is a fully associative memory cache, the "time-space" cache. This contains triples of the form: address, time stamp and value for all recent writes. As well it contains a record of all reads in the form of address, time stamp and tag triples (we will see why this is necessary shortly). Only read traces with time stamps greater than GVT need to be stored; traces prior to GVT are fossil collected during GVT computation.

The next two levels of the memory hierarchy are the standard ones of a cache, which we call a spatial cache, and local dynamic RAM. At these levels, memory is referenced only by address; time stamps are not involved. Values stored in this "timeless" part of the system are committed and are guaranteed never to be rolledback.

Four operations are possible on the memory subsystem: read, write, anti-write, and fossil collection as a result of an advance in GVT. Figure 3.1 shows a diagram of the memory subsystem's major components and the interactions between them.

A read will generate two parallel accesses one to the time-space cache and another to the spatial cache. In the time-space cache the read's trace will be recorded. As well, all recorded writes with the same address will be matched and the one which is most recent but prior to the time on the read will be fetched. There are two possible results to this operation. There may be a memory tuple which matches the read in which case this is returned to the CPU. But if there is no match then a check is made to see if the spatial cache returned a result. If it did then that is used as the result. If there is still no value available then the read request is passed up the (standard) memory hierarchy. If there is no local record of the address then action is needed to obtain a copy of the appropriate memory page from whatever remote CPU it is located (or to generate a memory reference error).

A write is sent only to the time-space cache where its trace is recorded. As well, all recorded read requests (for the same address) with time stamps greater than the time stamp of the write but less than the time stamp of the next recorded write to the same address are retrieved. Each of these is sent back as a reply to the CPU using the value just written. When these replies arrive at the CPU they will cause a rollback and re-execution of code in the CPU.

An anti-write functions similarly to a write. It first locates the original write entry at the specified time and address and deletes it. It then finds all recorded reads later than the deleted write and before the next recorded write to the same address. For each such read request, a reply is generated to the CPU giving the value of the earlier write (or an invalid value if there is none).

The fossil collection process deletes both read requests and write entries. All traces of read request traces earlier than the current value of GVT can be deleted immediately. Each write entry earlier than GVT is retrieved and its address noted. If it is later than all other writes to this address (which are earlier than GVT) then the value can be written directly to the spatial cache. All the entries less than GVT at that address can then be deleted. We anticipate that this process of fossil collection will run “in the background” concurrently with the requests from the CPU.

3.2. Inter-processor Communication

Being a shared memory system, all inter-processor communication is via the memory subsystem. The use of time stamps eliminates the need for standard memory consistency protocols since it ensures sequentially consistent memory accesses [16, 19]. Assuming some directory technique is in place for recording which addresses are to be shared between processors, then sharing memory is very simple. Whenever a write (or anti-write) is done to a shared address then the address, time stamp, value triple is sent to all machines sharing this address. There is no need to wait until the write arrives at all its destinations or to explicitly invalidate remote entries. When a (anti-)write arrives at a remote machine it is treated the same way as a (anti-)write from the local CPU which may cause the generation of multiple read replies to the CPU and consequent rollback. Furthermore, values fossil collected to the spatial caches will be automatically coherent without the need for any cache coherency protocols.

Because synchronisation is implicitly handled by the time stamping there is no need to implement any explicit locking or atomic operations between processors.

3.3. Size of the Time-space Cache

Clearly the time-space cache will be critical to the performance of the system and will present an implementation challenge; it will be significantly more complex than a standard fully associative cache. It does have one redeeming feature which is that its effectiveness does not depend on its size. The size will be determined by the number of read and write entries that have to be stored in it rather than in the spatial cache. There can only be a limited number of read/write requests from the local CPU or remotely from other CPUs on each clock cycle (in the performance estimates below we assume at most one cache operation per clock cycle). So the number of entries required will be some constant multiplied by the time gap between the highest time stamped entry and the current value of GVT. This spread will be determined by the spread in different time stamps which are actively being executed by different CPUs and the lag in computing the current value of GVT. Getting a good estimate for this spread and the size of the cache will require more detailed simulations than have been possible to this point. Initial estimates indicate that it will be dominated by the GVT lag which is determined by the diameter of the communications graph for the multiple-CPUs. Given at most one read or write per clock period and diameters of tens or hundreds of clocks the time-space cache may be effective with a thousand or so entries. In the single CPU case a few tens of entries are probably sufficient.

The size of the spatial cache and its structure is determined in the same way as conventional processors [13] and hence is not discussed in this paper.

4. Code Blocks and Time stamps

Within the WarpEngine CPU instructions are organised into blocks. Conditional execution selects (or rejects) a complete block for execution, that is, individual instructions cannot be conditionally executed. Also, all instructions within a block can (potentially) be executed in parallel. Figure 4.1a gives an example of a code block. A block roughly corresponds to a basic block in classical CPUs. A block has a fixed maximum number of fixed-width instructions (in the current incarnation of the instruction set it is 16 instructions).

During execution each code block is dynamically allocated a time stamp when it is scheduled for execution. This time stamp is used for all read and write requests issued from within the block. Time stamps are assigned so that they mimic sequential execution. The way that this is done is to associate with each block an interval of time and to allow it to schedule only a small, fixed number of “child” blocks (the current incarnation of the instruction set allows at most four children). The actual time stamp associated with a block is the earliest time point in the interval. The intervals of the children are disjoint and contained within the parents interval. In this manner, all the time intervals form a nested hierarchy or tree. A sequential execution of a program would then consist of a pre-order traversal of this tree of blocks, where the parent is always executed before its children.

Figure 4.1 shows a simple example of this process. The code to be executed is a simple loop that counts the number of zeroes in an array. This code splits into two blocks, one that does the loop execution and test for zero, and the other which increments the counter. The figure shows part of the tree generated for a particular set of values in the array.

Ultimately the time stamps have to be represented concretely as numbers. Space precludes us from going into details here but a number of different schemes are currently being investigated. One of the simplest is to allow the root of the block tree to use the range from 0 to 2^{32}-1. Each node then would have a range from i to j and would allocate the time stamp i to itself and four (approximately) equal ranges $i+1..i_1$, $i_1+1..i_2$, $i_2+1..i_3$, $i_3+1..j$ to each of the four possible children. This scheme is somewhat naive as with deeply nested and unbalanced trees precision will be rapidly lost. The use of a floating point representation together with occasional collection and re-scaling of the intervals gives a good compromise solution.

```
for I:= 1 to N do
        if V[I]=0 then C:=C+1;
```

(a) High Level Code

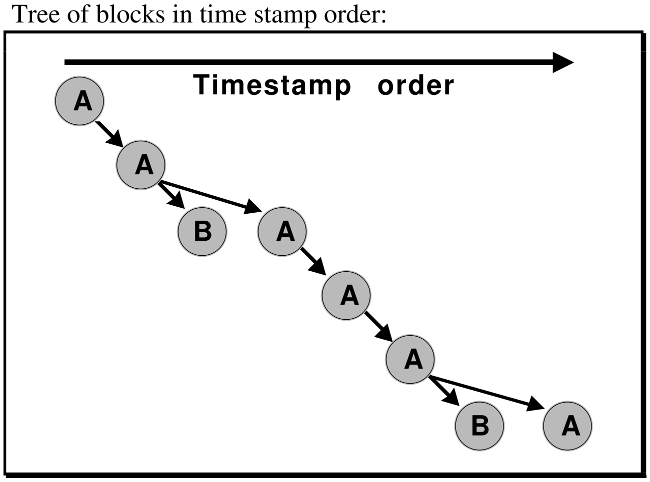
```
Block A:
        I := I+1;
        if V[I]=0 then schedule child 1 as block B;
        if I<N then schedule child 2 as block A;

Block B:
        C:=C+1;
```

(b) Blocks generated for Code shown in (a)

Example execution:

	1	2	3	4	5	…
V	42	0	1	2	0	…

Tree of blocks in time stamp order:



c) Trace of execution of blocks using the example data

Figure 4.1. Example of Execution of Blocks

The loop example above also illustrates the parallelism between blocks that is possible. In conventional parallel architectures it is not possible to parallelize the loop as written as there is a causal dependency between the successive iterations of the loop because of the counter C. However, in an array with few zeroes, C will seldom be incremented and the dependency will be very "weak". In the WarpEngine all the blocks can potentially be run in parallel. This parallelism will be limited only by rollbacks caused by increments to C rolling back later iterations which use C. The other limitation is the sequential order in which the successive blocks are dispatched. A more aggressive approach is to generate the successive iterations in a tree.

Whenever a block is scheduled for execution it can be executed either on the local (generating) processor or sent to another processor for execution. The time stamps on the blocks give some clues about how to do this. For example one strategy is to try and execute blocks with low time stamps as soon as possible. This would imply that as soon as a processor starts running ahead it should obtain low time stamped blocks from the rest of the system. This approach is probably too naive as rapid migration of blocks will cause poor cache coherence and consequent poor overall performance. In the end it may be that the software needs to provide hints or directives about when to dispatch blocks to other processors.

5. GVT Calculation

GVT is defined as the minimum time stamp of any object in the system, so calculation of GVT requires access to all time stamped objects in the system. This includes both blocks which are currently executing and memory requests currently being sent by CPUs or being transmitted between memories. Various GVT algorithms have been discussed in the literature [2, 18]. The important point here is that the calculation can be done in a time proportional to the diameter of the communication network between CPUs and that it requires both the CPUs and memory controllers to transmit their current minimum times.

GVT will be continually updated, probably by a separate communications channel running through the various CPUs. Once computed it then needs to be transmitted back to all the memory controllers (so that they can fossil collect the time-space cache) and to the CPUs so that they can delete blocks which cannot possibly be rolledback and re-executed.

6. Instructions

In this section a particular instruction set is described as an example of the WarpEngine architecture. A number of decisions about sizes (32-bit words, 16 instructions per block and so on) have been made for the sake of a concrete example. Clearly many variants of these basic ideas are possible. We have chosen for our example an architecture that closely follows that of a simple load store RISC

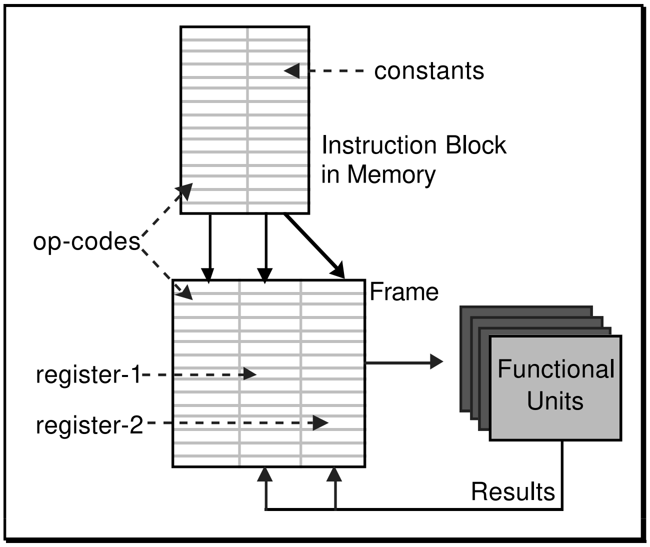


Figure 6.1. Flow of Instructions in CPU.

architecture using 32-bit words. This makes it easier to compare with modern designs.

6.1. Instruction Execution

We start with a description of the structure of a block and the path followed by an instruction during its execution. In an attempt to wring out as much parallelism as possible the individual instructions within a block execute in parallel. Rather than using time stamps to control this parallelism (which would require issuing a separate time stamp for every instruction) a dataflow mechanism is used. Instructions specify where their results will go (rather than where their arguments will come from). Figure 6.1 illustrates how this mechanism works.

In memory each instruction is represented by two 32 bit words. The first, the I-word contains the op-code and related information. The second, the C-word, contains a single literal or constant value, its use is explained below.

An instruction goes through a number of stages in its execution. After a block is initially dispatched it may be queued or migrated to another processor. Eventually it is "initiated". At this point the block is allocated a "frame". A frame resides in the CPU and consists of 16 slots (recall that there are 16 instructions in a block) with one instruction loaded into each slot. A slot contains three 32 bit fields: an op-code field and two "registers". The op-code is loaded directly from the instructions I-word. It specifies the instruction to be executed as well as other information. Potentially all the instructions in one block are loaded into the frame in parallel. (It is assumed that one such load can complete every clock cycle in the performance estimates below. So if a cache line were 16 words long the whole cache line would be loaded in parallel).

The registers in a slot can be loaded either from the C-word of an instruction or as a result of another instruction. Two bits in the I-word specify how the C-word is to be used. It can be either stored in register-1 or register-2 of the corresponding slot or ignored. Because some instructions require both their register slots to be filled with constants a C-word can also be directed to register-2 of the adjacent slot. There are thus only 16 constants for 32 register slots in a block. However, this seems in practice to be a very weak constraint.

Once both registers in a slot have been filled with valid values then the instruction can be "fired". That is, transferred to a functional unit for execution. The 5-bit op-code acts as an address and specifies which functional unit the instruction and its accompanying registers are to be routed to. The functional unit then receives the contents of both the registers as well the residual 25 bits (32 bits less two for C-word routing and 5 for the functional unit address - we refer to this as the F-field). Using these three pieces of information the functional unit computes anywhere between 0 and 5 results. The F-field specifies a register in the frame for each result. The functional unit then sends each result back and fills in the corresponding slot which will enable that instruction in turn to be fired completing the cycle.

6.2. Individual Instructions

Figure 6.2 shows an example instruction layout at each stage for an ADD instruction which adds 5 to a value 7 received from another instruction .

Within the I-word the value CC=01 specifies that the constant is to be stored in the register-1 of the current slot. Following that two fields dest1 and dest2 specify where the results of the instruction are to be stored. There are 32 possible registers in the current frame so 5-bits suffices. The first destination specifies where the actual sum truncated to 32 bits is to be stored, the second destination specifies where the overflow from the addition is to be stored. (The second value can be used to test for exception and overflow conditions). The 25 bits in the F-field allow up to 5 destinations to be specified. Three of these are used for the sum and the other two for the overflows. The convention is used that a destination of 0 means do not store that result. So three potential destinations are unused in this instruction.

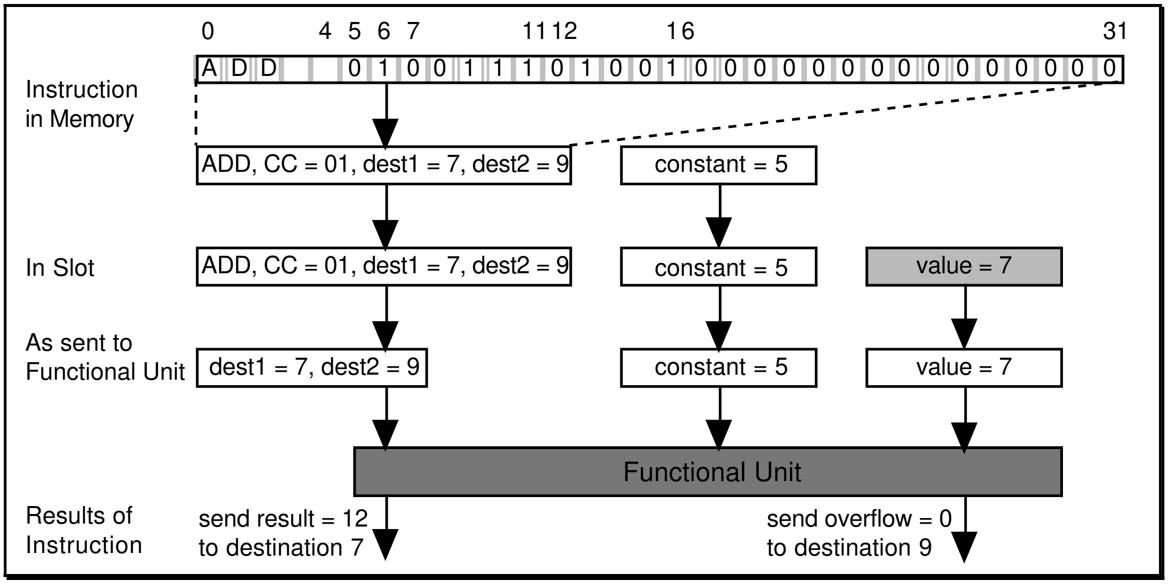


Figure 6.2. Instruction Format During Execution Stages.

All instructions (except the no-operation) take two parameters. All arithmetic and logical instructions (add, subtract, divide, shift, and, or, xor) use the F-field to specify (up to) 5 destinations.

There are three memory instructions which communicate with the memory subsystem. The load instruction adds the two values in its registers and uses the sum as the address to read from memory at the time of the current block. As with the arithmetic instructions the F-field is used to specify up to 5 destinations for the value returned from the read. The store instruction takes a value and an address and stores the value at that address at the time of the current block. As there is no need for any destination fields the F-field is used as a signed offset to the address.

There is one rather more exotic memory instruction. This allows a parent to store values directly into the register slots of its children. It was included because hand compilation of code showed that this process was very awkward with just a load and store and that it was frequently required. The instruction takes one value (it ignores its other register, the only instruction which does so) and stores it into a specified destination (5 bits) for one of the 4 possible children of this block (2 bits). (Thus 18 bits of the F-field are ignored).

There is a single instruction for comparing values. It takes its two input values and compares them. Three results can be generated from the one instruction. Each result is specified in the F-field by a 5 bit destination and 3 bits for the comparison operator ($<,\leq,>,\geq,\neq,=,0,1$). The results are 0 if the comparison fails and 1 if it succeeds.

There are three control instructions which use the results of such comparisons. The jump instruction specifies a block to be executed. It takes as values the address of the block and the result of a comparison operation (or a constant). If the comparison failed (the value is 0) then the block is not executed otherwise the block is dispatched for later execution. The F-field uses 2 bits to specify which child the block is to be and uses 23 bits as an offset to the block address.

There are two other control instructions, a halt and a trap, which are used to raise error conditions. They take as values a literal to indicate the cause of the error and a comparison flag which says whether to raise the error. For example, the trap could be used to raise an error if an arithmetic instruction overflowed.

6.3. Execution Rollback

Rollback of instructions is initiated when the memory subsystem returns a revised value from a read. It will have already returned some value and as a result of a (anti-) write the read may be resatisfied. What will appear to happen from the point of view of a slot is that a register that already has a valid value stored in it will receive a new value. If the slot has fired previously then it can deal with this by firing again.

One significant optimisation is to detect when the new value is the same as the old one, then the instruction does not need to be fired again. In some cases this optimisation may prevent a cascade of unnecessary recomputation. For correctness of the TimeWarp algorithm it is essential that the memory subsystem detect when the same value is stored into memory and does not do any memory replies as a consequence. However, from the point of view of instruction execution this is an optional optimisation.

7. The Hardware Structure of the WarpEngine CPU

An initial investigation is currently being carried out to evaluate the proposed architecture of the CPU. A number of issues that could influence the success of the CPU have been identified and are discussed in this section.

Success in implementing the Warp Engine will be heavily dependent on an efficient implementation of communication paths between the frames and the functional units and also the return communication paths from functional units back to the registers. Successful implementation is likely to be a compromise between the number of functional units and the complexity of the logic necessary to steer instructions and data between frames and functional units. At one extreme a complete set of functional units would be assigned to each slot of every frame. This would mean that a simple bus could be used to implement the communications paths between the slot and its set of functional units. This approach is very inefficient as only one of the functional units connected to a slot would be active at a time. This approach introduces redundancy as only one of the functional units connected to a slot would be active at a time.

At the other extreme, functional units would be shared between all slots of all frames of the CPU. While this would minimise the number of functional units necessary, the steering logic would be extremely complex if satisfactory performance is to be obtained. In addition a frame address would be required to be sent with an instruction so that its results can be routed back to the correct frame and the results pathway would also be more complex.

In the case that more than one slot can access a particular functional unit it is necessary to have a mechanism in place to ensure that instructions and data are not lost because a functional unit is not ready to receive them. One solution to this problem is to prevent the transfer from taking place until an appropriate functional unit becomes available. This solution is possibly unfair in that it cannot be guaranteed that all slots that can potentially communicate with a particular functional unit have equal priority. Alternatively, a buffer could be placed on the input stages of each of the functional units. This approach may however cause higher instruction execution latencies in the case where the buffers are empty.

There is a special case during execution that requires some additional hardware. This occurs when an instruction has been dispatched from its slot and is currently executing in a functional unit. If at this point, one of the registers (in the slot for the active instruction) receives a new value a difficulty arises. If the instruction is dispatched again then it may be executing simultaneously in two functional units (which may have different latencies). If these complete out of dispatch order then the wrong (rolled back) value will be stored in the result register.

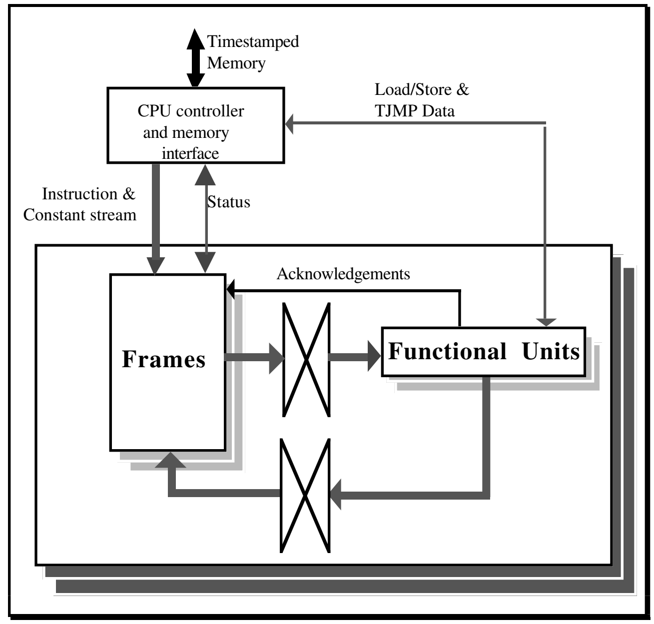


Figure 7.1 Block Diagram of CPU.

To deal with this an acknowledgment mechanism is required to allow the functional units to notify the slot that dispatched the instruction that it has finished and the results have been returned (see Figure 7.1). Only when the current execution has been acknowledged can any new re-execution be started.

As can be seen the final organisation of the CPU is far from being finalised. While the next section on *estimated performance* may give some insight into the number of frames required in each CPU, no estimates have yet been obtained for the numbers and types of functional units and the overall organisation. It is hoped that an initial functional simulation will give insight into these details.

8. Estimated Performance

As yet, it is not possible to give any detailed performance estimates. However, order of magnitude estimates give a good taste for requirements of each of the components of the architecture. The basic assumption we make in this analysis is that the different parts of the architecture are balanced, that is, there is no single bottle-neck to system performance.

The start of the analysis assumes that one memory operation can be completed per clock-cycle. This seems a reasonable assumption on the grounds that while it might be possible to multi-port the time-space cache this would be very complex and it might well be better to split the CPU into two before doing this.

Using a 95% on-chip cache hit rate this gives one external memory operation every 20 clock cycles. It seems likely that this difference will be consumed by the ratio between the differences in speed between the internal cache and external memory [13].

Using the estimates from [13] approximately 20% of instructions will be references to memory. So for instruction execution to keep up, 5 instructions must complete and be dispatched per clock. This requires the switch to transmit 5 instructions per clock. Each of these instructions requires between 0 and 2 data values. Examination of hand generated code gives an average of about 1.5 values per instruction, so the result switch must return on average 7.5 results per clock.

Examination of the contents of hand coded blocks shows that they average about 10 instructions each. At 5 instructions per clock this implies that one block must be started every 2 clock cycles. This is possible if all the instructions in a block are loaded in one clock and then all the constants are loaded in the next.

Operation	Number completing per cycle	Cycles to complete
Memory read/write	1.0	
External memory	0.05	
Instructions	5.0	
Results	7.5	
Code block	0.5	
Minimum Latency		3.0
Average latency		5.0
Number slots for blocks		> 8.0

Table 8.1. Summary of Performance Estimates.

Causal links between instructions limit the average number of instructions that can be executed in parallel to roughly 3 (that is the critical path through a block is 3 to 4 instructions long). This implies that the number of blocks active at any one time will be $5/3$ times the average latency of instructions. Some idea of this latency can be gained by looking at a few representative instructions. Every instruction has to be dispatched, will take at least one clock to execute and then will take one clock for the results to be returned. That is, the minimum latency will be 3 clocks. A memory read that comes directly out of cache will likely require one extra clock to compute the address. Assuming 5% of references will require 20 clocks (for an off-chip fetch) an average memory reference will have a 4.9 clock latency. Using an average of 5 clocks for an overall instruction latency gives 8 to 9 $(= 5/3 \times 5)$blocks currently active. More space needs to be allocated for blocks than this as the lag in computing GVT means that some blocks, although they are inactive and will not be rolledback, still cannot be released until GVT advances. Table 8.1 summarises these performance estimates.

9. Summary and Future Work

So what can be said about an architecture like this where a great many details are still to be refined and where it is far from clear what the final performance or size of the system will be? The first claim that we would like to make is that to a first approximation the WarpEngine is feasible. Clearly some parts of the design are complex but it is more a question of by what date we could build such a system rather than whether it could be built at all. The second claim we would like to make is that it presents a scalable and conceptually simple approach to extracting parallelism from programs. The parallelism in the WarpEngine is limited only by the dynamic causal links between instructions. It is possible to envisage more efficient mechanisms but not ones that extract more parallelism. The other major advantage of the WarpEngine approach is that the burden of synchronising shared memory is completely removed. So at least from the point of view of achieving correct code the programmer is faced with the same problem as on a sequential computer. One implication of this is that it may be feasible to parallelise (and more daringly speed up) dusty deck code.

Given this there are still ways in which the approach could fail. First, the conceptual simplicity could vanish in the details of implementation. For example, it is unclear how best to implement key elements including the switch and the time-space cache. Second, the burden of control parallelism still remains to some extent. It is unclear how smart compilers will need to be to get good performance and the extent to which this burden will be forced back onto the programmer. The programmer must shoulder at least some of the burden as there are cases where the best algorithms are different on sequential and on parallel computers.

The final proof of this approach will be in demonstrating that the complete system from user code through compilers to execution can deliver good performance. The author's have had sufficient experience with the frustrations of extracting parallelism from real problems to realise that this is no small project. We look forward to the challenges of constructing the simulators and compilers (and eventually hardware) to prove these ideas.

Acknowledgments

Husam Kinawi would like to acknowledge support by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Agarwal, A., Chaiken, D., D'Souza, G. , Johnson, K., Kranz, D., Kubiatowicz, J., Kiyoshi Kurihara, K., Lim, B-H. , Maa, G., Nussbaum, D., Parkin, M. and Yeung, D., (1991) "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," Proceedings of The Workshop on Scalable Shared Memory Multiprocessors, Kluwer Academic Publishers.

[2] Bellenot, S., (1990) "Global Virtual Time Algorithms," Proceedings of the 1990 SCS Multi-conference on Distributed Simulation, Vol. 22(2), pp. 122 - 130, January.

[3] Catanzaro, B., (1994) "Multiprocessor System Architecture," Sun Microsystems Inc., Mt. View, CA, SunSoft Press, Prentice Hall Title.

[4] Choi, J. D., Miller, B. P. and Netzer, R. H. B., (1988) "Techniques for Debugging Parallel Programs with Flowback Analysis," Dept. of Computer Science, Univ. of Wisconsin Tech. Report No. 786, later in ACM Transactions on Programming Languages and Systems.

[5] Courtois, B., (1993) "CAD and Testing of ICs and Systems - Where are we going?" Techniques of Informatics and Microelectronics for Computer Architecture, Grenoble, France, November.

[6] Fujimoto, R. M., (1989) "The Virtual Time Machine," Dept. of Computer Science, Univ. of Utah Tech. Report UUCS-88-019.

[7] Fujimoto, R. M., (1990a) "Time Warp on a Shared Memory Multiprocessor," Trans. of the SCS 6(3), pp. 211-239, July.

[8] Fujimoto, R. M., (1990b) "Parallel Discrete Event Simulation," Comm. of the ACM, Vol. 33, No. 10, pp. 30 - 53, October.

[9] Fujimoto, R. M., Tsai, J., Gopalakrishnan, G., (1992) "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp," IEEE Transactions on Computers, Vol. 41, No. 1, pp. 68 - 82, January.

[10] Ghosh, K. and Fujimoto, R. M., (1991) "Parallel Discrete Event Simulation Using Space-Time Memory," International Conference on Parallel Processing, pp. III-201 - III-208.

[11] Hagersten, E., Haridi, S. and Warren, D., (1990) "The Cache-Coherent Protocol of the Data Diffusion Machine," Cache and Interconnect Architectures in Multiprocessors.

[12] Hennessy, J. L. and Jouppi, N. P., (1991) "Computer Technology and Architecture: An Evolving Interaction", IEEE Computer 24(9), pp. 18 - 29, September.

[13] Hennessy, J. L. and Patterson, D. A., (1990) "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers.

[14] Jefferson, D. R., (1985) "Virtual Time," ACM Transactions on Programming Languages and Systems, 7(3), pp. 404 - 425, July.

[15] Jefferson, D. R., (1989) "Virtual Time II: Storage Management in Distributed Simulation," Proceedings of the Annual Symposium on Principles of Distributed Computing, pp. 75 - 89, August.

[16] Lamport, L., (1979) "How to make a multi-processor computer that correctly executes multiprocess programs," IEEE Transactions on Computers, Vol. C-28, No. 9, September.

[17] Lenoski, D., Laudon, J., Stevens, L., Joe, T., Nakahira, D., Gupta, A. and Hennessy J. L., (1992) "The DASH Prototype: Implementation and Performance," Proc. of 19th Annual International Symposium on Computer Architecture, pp. 92 - 103, May.

[18] Lin, Y. and Lazowska, D., (1990) "Determining the Global Virtual Time in a Distributed Simulation," Proceedings of the International Conference on Parallel Processing III, pp. 201 - 209.

[19] Mosberger, D., (1993) "Memory Consistency Models," Operating Systems Review, Vol. 17, No. 1, January. Also in, Dept. of Comp. Sci., Univ. of Arizona Tech. Report 93/11.

[20] Olthof, I. and Cleary, J. G., (accepted 1994) "The Design of an Optimistic AND-parallel Prolog," Journal of Logic Programming.

[21] Prakash, A. and Knister, M. J., (1992) "Undoing Actions in Collaborative Work," ACM Conf. on Computer Supported Cooperative Work: Sharing Perspectives, Toronto, pp. 273 - 280, November.

[22] Rothnie, J., (1992) "Overview of the KSR1 Computer System," Kendall Square Research Report TR9202001, March.

[23] Thimbleby, H., (1990) "User Interface Design," ACM Press, pp. 261 - 286.

[24] Unger, B. W., Cleary, J., Dewar, A. and Xiao, Z., (1990) "A Multi-Lingual Optimistic Distributed Simulator," Transactions of the Society for Computer Simulation, Vol. 7, No. 2, pp. 121 - 152, June.