

Dynamic Scheduling in RISC Architectures

A. Bolychevsky, C. R. Jesshope and V. B. Muchnick

Department of Electronic and Electrical Engineering
University of Surrey
Guildford GU2 5XH, UK

Abstract

Multithreaded processors support a number of execution contexts and switch contexts rapidly in order to tolerate highly latent events such as external memory references. Existing multithreaded architectures are implicitly based on the assumption that latency tolerance requires massive parallelism, which must be found from diverse contexts. We have carried out a quantitative analysis of the efficiency of multithreaded execution as a function of the number of threads for two important classes of memory systems: conventional off-chip memory and symmetric networks. The results of these analyses show that there are fundamental reasons for the efficiency to grow very rapidly with the number of threads. This in turn implies that the original goal of latency tolerance can be achieved with only a limited number of threads that can typically be drawn from the same referential context and do not therefore require the heavyweight hardware solutions of conventional multithreading. A novel dynamically scheduled RISC architecture is presented based on this new understanding of the problem.

1 Introduction

There are many constraints on computer architecture; technological limitations and instruction set compatibility being two of the most tight. In this paper we propose and justify some future directions in RISC based processor design which provide solutions to a fundamental problem encountered in a wide range of computer systems; that of statically scheduling concurrent operations in order to avoid high-latency and non-deterministic events. Our solution to this problem, namely to remove the necessity of a static schedule, is not unique. However, the manner in which this is achieved and the impact that it has, due to the very small overhead required, are we believe quite novel.

Let us first consider why scheduling, whether dynamic or static, is such an issue in today's computer systems. Even if we talk of a sequential or uni-processor systems there is always concurrency to exploit. All processors these days are pipelined and many support multiple issue of instructions either in a programmed manner (VLIW) or an implicit manner (Superscalar). Of course this also implies that compiler writers, even when compiling sequential code, must analyse dependencies and extract concurrency in order to exploit these features. Ideally a compiler writer should know the exact behaviour of every instruction executed, in order to produce the perfect schedule, i.e. one which overlaps sufficient independent instructions to overcome high latency operations, for example, when performing a memory fetch or computing some result etc. Unfortunately this is not the case, indeed technological considerations, such as divergence between memory and processor performance (on and off-chip performance), mean that this is unlikely ever to be the case again. Non-determinism in the execution of computer instructions occurs in many areas:

- in cache based memory systems, where a cache miss will mean the difference between waiting just a few cycles or many cycles for a memory reference to be satisfied,
- branches in control, which must be predicted if instructions are to be prefetched and decoded, or
- network based shared-memory systems found in parallel computers, where accesses have a very high latency aggravated by a significant dispersion due to contention in the network.

Any misprediction in these areas will destroy a static schedule and may have severe consequences on overall performance. Dataflow research [12] was thought by many to provide the solution to the scheduling problem but the solution it provides is far too general, not only does it support a dynamic schedule but also dynamic parallelism, which is not required in compiling most imperative programming languages. Moreover, the overheads are high and the resulting processor designs tend to require deep pipelines which in turn make the solution inapplicable to the majority of installed codes, from which only modest levels of parallelism may be extracted. More recently, many other techniques have been proposed which enhance the conventional RISC based approach to processor design. These solutions either attempt to predict the non-determinism in the areas of cache accesses and branching or try to mitigate against the effects of misprediction. Branch prediction is a technique which has been used for some time in existing microprocessors but which is still being refined [9]. Cache prefetching and data streaming [1, 6, 10, 11, 15, 19] attempt to ensure that the required data is pre-loaded into either the cache memory or a dedicated stream buffer prior to a memory fetch being issued. Yet another approach taken is the lockup-free cache [18, 21]. Often however, these techniques

introduce further speculation, such as that involved with prefetching [8], which can, as has been demonstrated [16], have an even more detrimental effect on performance in the event of a misprediction.

It seems clear (to us) that computer architects are looking in the wrong direction down the arrow of time and instead of designing computers that try to prejudge a program's data accesses or branches they should simply look at tolerating the latencies involved. This paper demonstrates how this may be achieved as well as showing that it need not require massive concurrency as is often thought to be the case.

The alternative approach is multi-threading, which is far from being a recent development as Burton Smith's pioneering work on the Delencor HEP demonstrates [20]. It does, though, seem to have come of age [2] recently. However, the most up-to-date work on multi-threading [3, 13] still takes the view that a thread is a lightweight process complete with minimal context, such as stack and registers. Our own approach is that a thread is just a program counter. In both cases, non-deterministic events which are mispredicted, such as branches and loads, will cause a new thread (program counter) to be executed. In our view it seems strange that many contexts should be thought of as a good basis for non-deterministic thread interleaving, as this will play havoc with any locality that may exist within a single context, with consequent loss of performance due to cache misses and memory bandwidth limitations. Indeed the authors can find no justification for this approach whatsoever and demonstrate in this paper that relatively few threads are required in order to obtain good performance from a multithreaded processor and that such a small number of threads may easily be derived from within a single context.

In order to differentiate this approach from current thinking we will refer to it as micro-threading. Of course with the expectation that such threads will be rather small, possibly just a few instructions, it is imperative that the overheads for fork, join and synchronisation are extremely low. In section 2 we outline our approach to designing a modified RISC processor, which supports dynamic scheduling through micro-threading. We will argue that for single threaded code this approach need be no less efficient than the conventional single threaded RISC processor it replaces. Moreover, we will demonstrate how very simple techniques enable us to introduce micro-threading with little or no overhead in terms of instructions executed. In section 3 we develop an analytical model for both conventional and network based memory systems which confirms some previously simulated results on network based systems [4]. This model demonstrates how few threads are required to sustain a high proportion of maximum throughput from such systems. This analysis justifies the approach we have described in section 2.

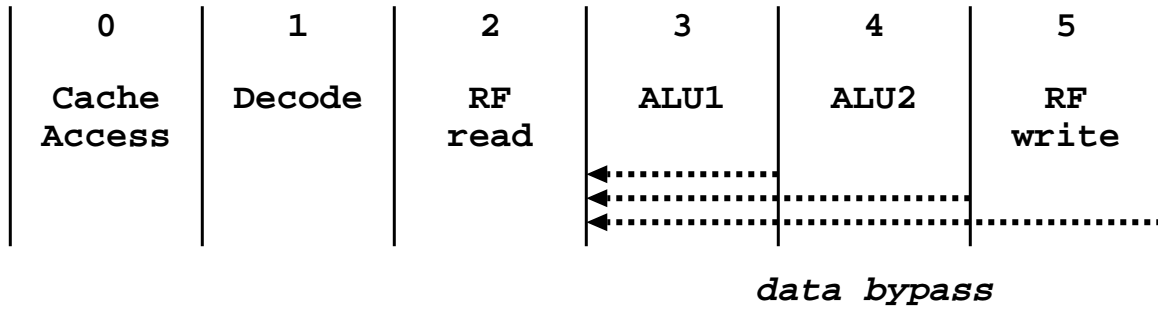


Figure 1: RISC integer pipeline timing

2 A dynamically scheduled RISC-based architecture

Our goal in this work has been to design micro-threading primitives that might be added to any modern RISC based processor. The resulting design should be completely compatible with existing compiled codes, which should execute with no loss of efficiency. Moreover, this should require as few additions as possible to the instruction set of whatever RISC design it is based. The techniques proposed will provide very rapid context switching of lightweight threads (micro-threads) which share a single set of registers and the stack. Our motivation is to support the efficient execution of data-parallel languages [5]. However, due to the economics of the microprocessor market, the processor must execute existing codes with unaffected performance and should provide enhanced performance even if “sequential” code is recompiled for it, using established techniques such as the exploitation of functional parallelism in expressions, independent statements, loop unfolding, etc.

Let us consider the generic pipeline, illustrated in Fig. 1, which is typical of many current RISC processors. It has 6 stages with bypass buses from the final three. The first issue which must be considered is how synchronisation may be achieved on the non-deterministic events whose latency must be tolerated. On RISC processors which are based on the principle of decoupling memory accesses and processing, by adopting a load/store philosophy, it seems an obvious extension to base any synchronisation on additional state associated with the register, effectively providing a split-phase asynchronous LOAD mechanism.

2.1 Split-phase asynchronous LOAD

An asynchronous LOAD instruction can be implemented by simply providing an additional status bit on each register, which indicates whether the register contains valid data or not. Considering our generic RISC pipeline (Fig. 1), a LOAD is executed as an instruction which normally writes a “data invalid” state into its destination register. Meanwhile, the data cache

read begins at stage 3 and, in the case of cache hit, the fetched data is multiplexed into the output of stage 4 overloading the previously prepared “invalid” data. If the access misses the cache, then the “invalid” data state is written into the destination register. The register will be re-written later by the cache memory subsystem (by inserting a bubble into the pipeline or via a dedicated port to the register set). Thus any outstanding memory request must be tagged by the register number into which the data will be written.

We say that a memory access is split-phase if it misses the first level cache. More precisely, a split-phase LOAD is one that can not be satisfied during the life cycle of that LOAD in the pipeline.

Any instruction which reads an “invalid data” state from either of its operands stalls at stage 2. It can either wait for the data bypassed from the next stages or merely keep reading the register file. However, in either case, no further processing may take place until the requested data becomes available. Clearly this situation is undesirable, for although the compiler may be able to insert sufficient slack to tolerate the load, any dispersion of latency will destroy the schedule. In the next section we also consider a mechanism for removing this restriction of a fixed schedule.

2.2 Dynamic micro-threaded scheduling

To perform true dynamic scheduling of several instruction streams we have to introduce the explicit notion of independent points of control, i.e. the manipulation of multiple program counters (PCs) by the processor. A PC represents the minimum possible context information that we can keep for a given thread and in the architecture suggested it is the only reference to a thread. Since several threads can be active simultaneously, some explicit storage for their PCs, called the continuation queue, must be provided. This is associated with the instruction fetch logic at the entry of the pipeline (Fig. 2).

In a normal RISC pipe the next address is transferred from the first stage of the pipe in order to allow the next instruction to follow without delay. Of course, on branch instructions, this normally involves an element of speculation as the direction taken must be predicted and if this prediction fails any subsequent change of state must be “cleaned up”. We will call this conventional mechanism of transferring control *horizontal transfer* and the alternative mechanism that we propose here, which acts through the continuation queue, *vertical transfer*. Any instruction can transfer control vertically, horizontally, both vertically and horizontally or indeed not at all. Thus we already have a mechanism for the creation and termination of threads. Of course whenever an instruction does not transfer control horizontally, the next PC is taken from the continuation queue, providing that it is not empty. This mechanism, combined with a modified form of synchronisation described below provides the basic mechanism for

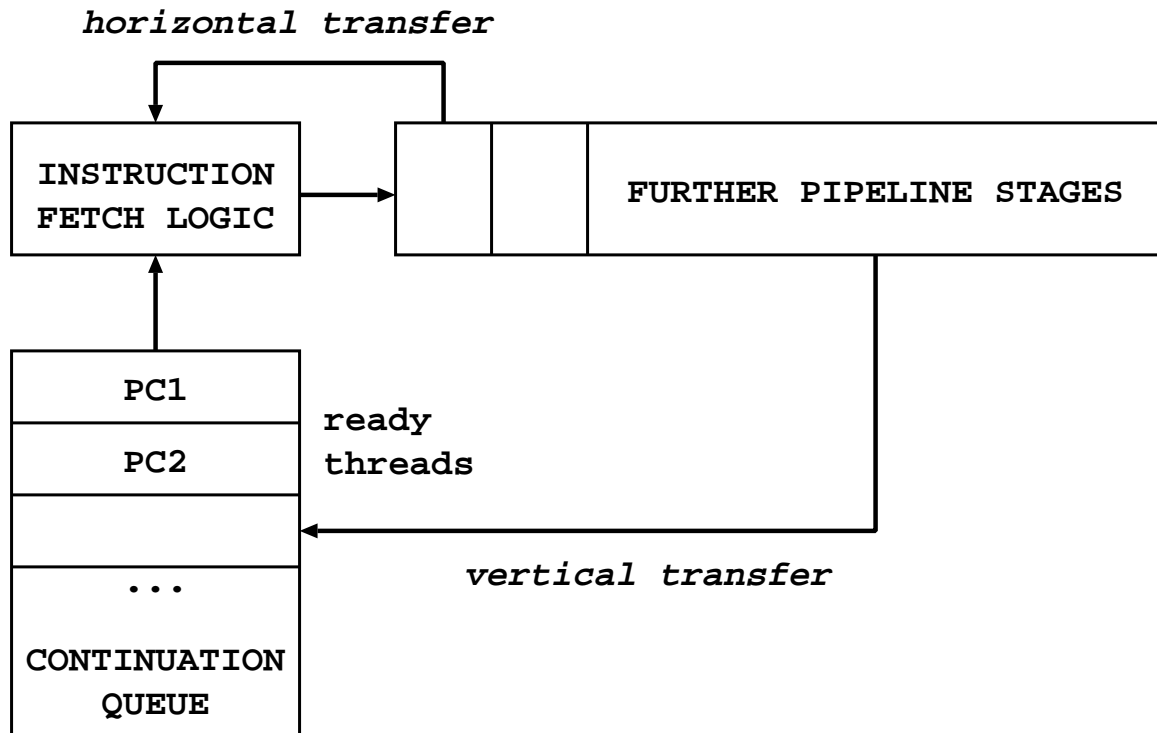


Figure 2: Modified mechanism of control transfer

latency tolerance.

A thread is created when an instruction is encoded to transfer control both horizontally and vertically and a thread is terminated when an instruction is encoded not to transfer control at all. How this encoding is achieved is a matter of detailed design but there are two obvious extremes. The first, if the instruction set allows it, is to use two spare bits in some or all instructions in order to encode the direction of transfer, thus allowing any instruction to create or terminate a thread. The other extreme is to add a pair of instructions to the instruction set in order to perform thread creation and termination explicitly, and then to provide a fixed encoding over the remaining instructions to determine whether the instruction transfers control horizontally or vertically. The former is usually preferable but should the two bits required not be available, the latter, which decodes the transfer strategy from the instruction code, is not always optimal. The instruction which is responsible for a non-deterministic delay (e.g. a LOAD) is not necessarily the one that needs to be coded for vertical transfer; it may be beneficial to code its consumer (i.e. the instruction that reads the register loaded) so that the compiler may insert a sequence of statically scheduled instructions between the two, thus maximising the concurrency available while still minimising the probability of the consumer being put to sleep.

With this scheduling mechanism it is no longer necessary to predict branch direction providing that enough parallelism is available in the code. However, in order to simplify the misprediction recovery, it makes sense to issue the vertical PC from a deeper pipeline stage at which the actual branch direction is already known. If the precise arithmetic exceptions are not required, this can be done right after the register file read (at stage 3 in our example in Fig. 1).

2.3 Sleep-wakeup synchronization

Now that we have introduced a mechanism for dynamic scheduling, it is necessary to revisit the synchronisation mechanism, involving the split-phase load, described above. Any instruction which is dependent on a non-deterministic event, such as access to a register previously loaded from external memory or conditional branch, will normally be encoded to transfer control vertically, pulling another thread into the pipe behind it. Now at the register file read stage (stage 3 in Fig. 1) the instruction either completes and transfers vertically to the continuation queue where the thread waits to be scheduled again or, if the data is not present (or some event has not occurred) the PC itself is written into the register read and that thread is neatly put to sleep.

Now all registers are required to be tagged with two status bits indicating three possible states:

1. The register contains invalid data. The register can be set into this state by a special instruction or a LOAD instruction which misses the primary cache.
2. The register contains a program counter. The register contains the PC of a sleeping thread.
3. The register contains valid data. Any other write of a data sets the register into this state.

The fourth combination of the status bits can be used for a postponed error report.

The way synchronisation is achieved is that when an instruction attempts to read an operand containing invalid data, the instruction is aborted and transformed into a “store program counter” instruction which replaces the register’s contents by the PC pointing to that instruction. Hence the thread is put asleep and its PC is kept in the register to which data is expected to be written later. The arrival of that data causes the PC to be pushed out of the register and put into the continuation queue. When rescheduled, the same instruction will find that data in the register its PC just vacated.

These two synchronization actions (sleep and wakeup) require mutually exclusive read-modify-write access to the register bank. This can be implemented using dynamic dependency control, stall and bypass logic which is embedded in most RISC pipelines.

In executing non-threaded code, all instructions would transfer control horizontally (as there are no additional threads to be executed). This situation is very similar to that of a conventional pipe, a prediction is made (in the case of a LOAD dependency the prediction is that the data is in the primary cache) and the next instruction is executed. Again if an operand contains invalid data a misprediction recovery is performed, i.e. the continuous chain of horizontally fetched instructions headed by the mispredicted instruction is cancelled. However, when there are no other ready threads available it is preferable to use the stall mechanism rather than the sleep-wakeup mechanism to reduce the penalty involved.

In threaded code, if both operands of a dyadic operation are invalid, we can choose either of them as a target for the thread's PC. However, a fixed rule (e.g. always the first) can allow the compiler to perform additional optimisation.

The wakeup action is performed explicitly by means of a special "move synchronizing" instruction which is inserted into a slot created in the pipeline by stalling earlier stages. This instruction moves one register to another. However, it reads both the source and the destination registers and if the destination contains a PC, that PC is issued vertically later in the pipeline. In fact, any instruction reading only one register (e.g. an operation with literal) can be used for this purpose provided it does not itself transfer control vertically.

The manner in which a split-phased memory request delivers the data using a "move synchronizing" instruction is illustrated in (Fig. 3).

2.4 Analysis of overheads

We have shown in the sections above how fork, kill, wait and signalling may be implemented in a conventional RISC pipeline using its register set as the synchronisation resource. In the introduction we argued that in order for micro-threading to be viable it must be possible to initiate and synchronise threads of a few instructions with no significant overhead. In this section we analyse the overheads involved in these various actions.

Thread creation

The overhead of a fork depends on detailed design at the instruction set level. Any instruction which has space for an additional address may be encoded to transfer both horizontally and vertically and hence yield a fork at zero cost. Alternatively a variant of a branch instruction may be encoded as a fork. There is also a trade-off in implementation, as it may be necessary (for example in a conditional fork) to transfer control vertically on both continuations and without

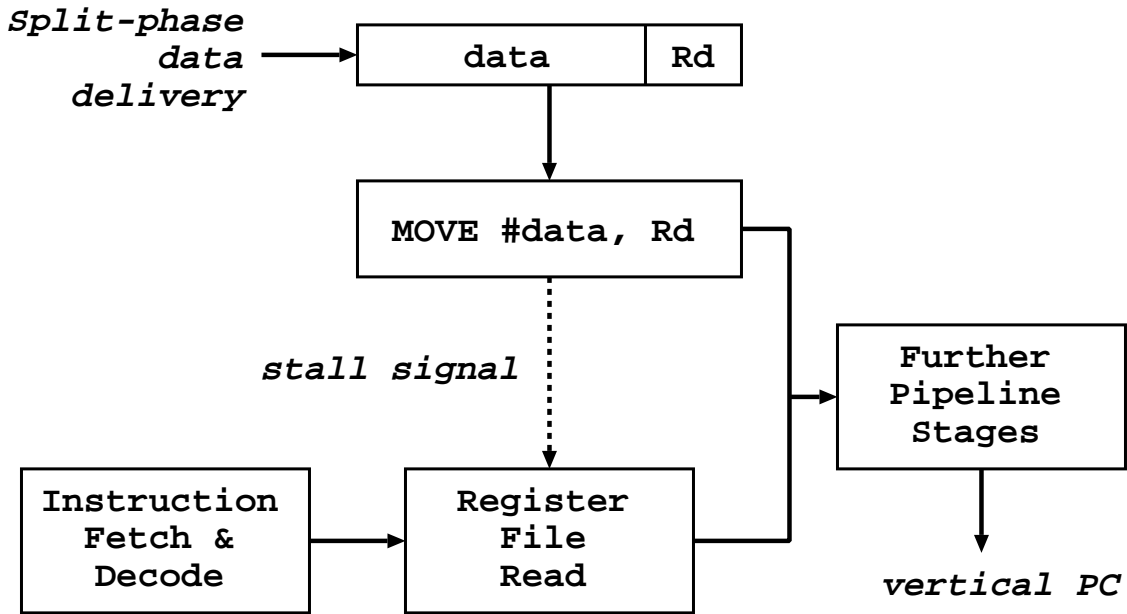


Figure 3: Delivering data requested by split-phase transaction

additional hardware support this could require two additional instructions. The overhead for a fork therefore, is 0, 1 or 2 cycles, known statically, depending on instruction set encoding and hardware tradeoff. The frequency of finding a double vertical transfer in compiled code will determine whether such a tradeoff should be made in order to keep the overhead bounded by a single cycle.

Thread termination

The overhead for a kill also depends on detailed design but as no transfer of control is required the overhead is 0 or 1 cycle. Again this will depend on whether a special instruction is added or existing instructions are encoded for transfer of control (in this case the encoding is for “no transfer”).

Sleep

This action is implicit in any instruction that reads a register. Storing the PC does not require an additional cycle, as the result of the current instruction is not written if its thread is suspended. However, the suspended instruction must be reissued once the dependency which put it to sleep is resolved. The overhead for a sleep instruction is therefore 0 or 1 cycle, which is not known statically, for it will depend on whether the register contains “invalid data” or not. More importantly, the overhead for a valid prediction is 0 cycles.

Wakeup

A wakeup signal may be generated internally or externally, as is the case with a split-phase load. An internal signal may again require a separate instruction, although it can be encoded as an option on any instruction which reads no more than one register and does not transfer control vertically. This latter condition will often be satisfied as signalling is often performed as the last action of a thread, when it will transfer neither horizontally nor vertically. Thus for an internal signal the overhead is 0 or 1 cycle, known statically. Again there is a hardware tradeoff in the case where no additional instruction is added, as by providing an additional register port the restriction of reading no more than one register may be removed, thus completely eliminating the overhead of internal signalling.

An external signal, such as split-phase load, must create a slot in the pipeline in order to insert the “move synchronizing” instruction. Thus the overhead here is always 1 cycle.

2.5 Summary

It is clear from the above analysis that micro-threading may be achieved with little overhead. In the case where existing instructions are overloaded with transfer method, the overheads of fork, kill, internal signals and a successfully predicted wait can be eliminated entirely. In the case where additional instructions must be added to an existing instruction set, the overhead for all threading operations may be limited to a single additional cycle. The following section now demonstrates the viability of finding sufficient threads within a single context to maintain a high percentage of peak sustainable performance.

3 On the required number of threads

The question we must ask ourselves now is how many threads are required to tolerate the latency found in typical memory systems, as this will determine whether micro-threading is a viable architectural model. In order to answer this question, let us simply study performance as a function of the number of threads, as the ultimate goal is maintaining a high percentage of maximum possible performance. In this analysis, we will see that, contrary to the common belief, latency of the memory system is not the only factor that influences this function. Latency has to be considered in combination with other factors including the program behaviour.

What then is the correct measure of performance? We are looking for a function $P(n)$ that depends on the program as a whole, on the number of threads in its multi-threaded representation (this is the explicit parameter of the function), and on the characteristics of the memory system.

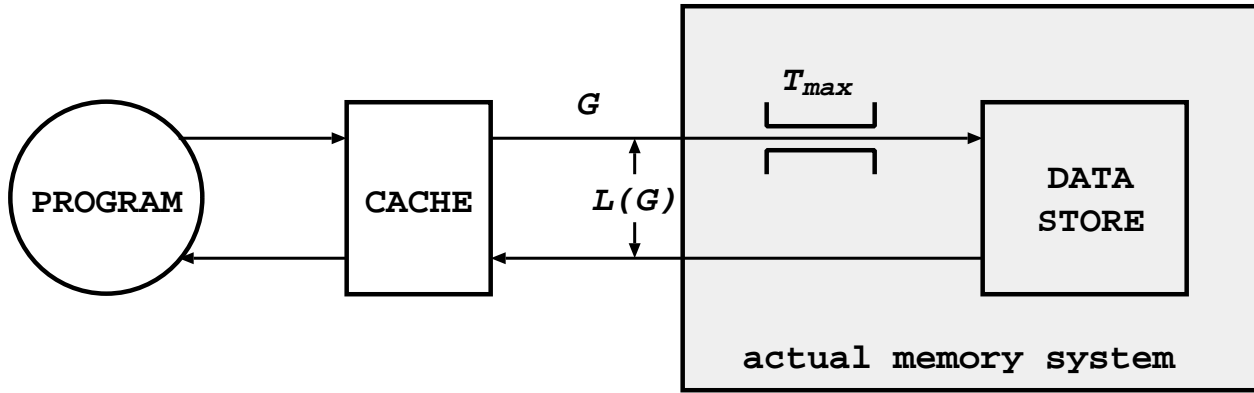


Figure 4: Communication with a memory system

We will introduce $P(n)$ in the framework of the simplified model presented in Fig. 4. The program tries to read its data from cache (writes are considered non-blocking and are therefore unrelated to the problem in question). We assume that a cache miss suspends the current thread and causes a request to the memory system. Although we do not specify the nature of the memory system at this stage, we presume that it has some throughput limitation and some latency. By observing the execution of a given program on such a system, we can determine the following parameters:

1. Average memory throughput G (measured in requests per unit of time) granted to the program. This parameter depends both on the technical characteristics of the memory system and on the program's behaviour: a program that has high degree of temporal locality and therefore receives most of its data from cache will cause low throughput on the memory channel.
2. Average latency $L(G)$ of the memory system. It, again, depends both on the memory system as well as on the program (indirectly via G). For example, $L(G)$ tends to grow with G : the heavier is the workload, the longer are the delays caused by queueing.
3. The maximum sustained throughput T_{max} of the memory system. It does not depend on the behaviour of the program, but restricts the average memory throughput granted to it: $G \leq T_{max}$.

As G and L depend on the program's behaviour, we need an adequate measure of it. In order to establish this measure, let us return to the original problem of compensating for memory delays. The most natural way to assess efficiency of such compensation for a given program is to see what would happen if there were no delays whatsoever.

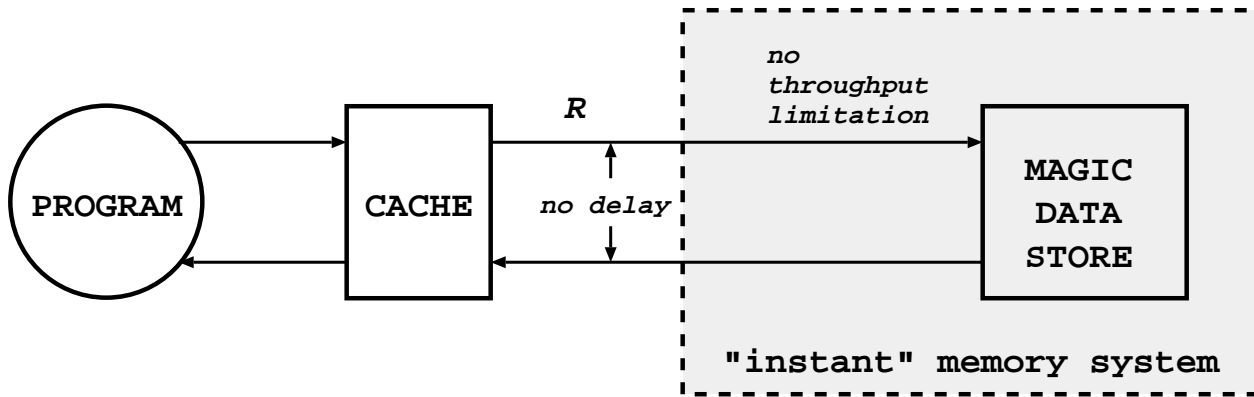


Figure 5: “Instant” memory system

Let us temporarily replace the actual memory system by an imaginary “instant” one that does not limit throughput and responds with data in zero time (Fig. 5). The average throughput R of the memory channel observed in such an experiment is a characteristics of the program: it is its requested throughput, i.e. the throughput which this program would cause if the memory system were capable of responding instantaneously.

Of course, it may be argued that R depends on a number of factors including locality, which may not be constant but vary with the number of threads in the program’s representation. However, what we are interested in is the performance of a system for a given R but with varying number of threads. Micro-threading in any case will minimise the effects of data locality as a function of number of threads.

In order to simplify the analysis we assume that the probability of a load instruction causing a cache miss is a constant which reflects the degree of temporal (and, to an extent, spatial) locality of the program as a whole. In particular, this probability does not depend on the number of threads in the multi-threaded representation of the program, nor on the manner in which those threads are scheduled for execution, nor on the nature of the memory system.

Under this assumption, the statistics of memory requests can be approximated by the Poisson distribution: at each clock cycle of the processor a request is issued with some constant probability (if throughput is measured per clock cycle then this probability is R , though our analysis is invariant to the choice of time unit). The underlying logic is as follows. According to the RISC community folklore, on average one in 3–4 instructions is a load, which makes it a fairly frequent event. As no memory system is capable of handling such workload, caching is essential and the average probability of a cache miss is much less than 1, which means that memory requests are, conversely, infrequent (in both cases the time scale is given by the clock cycle). Moreover, in a multithreaded processor the actual order in which the instructions are executed is randomised by the dynamic interleaving of threads: see Fig. 6. Therefore, the

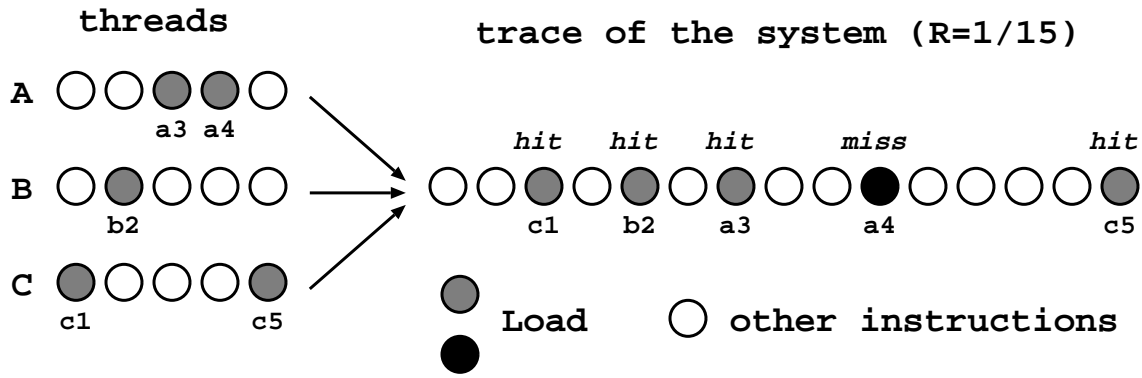


Figure 6: Probabilistic nature of memory requests

event of a memory request (i.e. a load instruction that causes a cache miss) occurring at any given clock cycle is of predominantly probabilistic nature, and treating the statistics of memory requests as Poissonian is justified.

We can now define the performance function as processor utilisation that varies between 0 and 1:

$$P(n) = \frac{G(n)}{R}. \quad (1)$$

The average proportion of memory requests per executed instruction is the same with or without delays due to our assumption that the probability of a cache miss is constant. Therefore, the ratio of the two throughput values equals the ratio of the corresponding numbers of executed instructions per unit of time. As R is the memory request rate in the situation when there are no delays and therefore the processor is utilised completely, $P(n)$ as defined above is the degree of the processor utilisation achieved by the program.

Note that if $R > T_{max}$ then the processor utilisation can not exceed T_{max}/R , as $G(n) \leq T_{max}$ for any number of threads. This is what should be expected: $R > T_{max}$ means that the program consistently issues more memory requests than the memory system can handle; in such situation, the processor is bound to remain idle for some proportion of its time.

We have reduced the problem to finding the function $G(n)$ which, of course, depends on the program behaviour R and on the characteristics of the memory system; these characteristics, in turn, may be dependent on G . Now we will obtain an equation that links all the relevant parameters.

Our general model of the memory system shown in Fig. 4 may be treated as a black box that may contain unfinished transactions. The average number of transactions being simultaneously processed by the memory system is given by the product $GL(G)$. In the following analysis, we will ignore the discrete nature of the processor, i.e. the fact that the time difference between two consecutive transactions coming into the black box can not be less than one processor cycle.

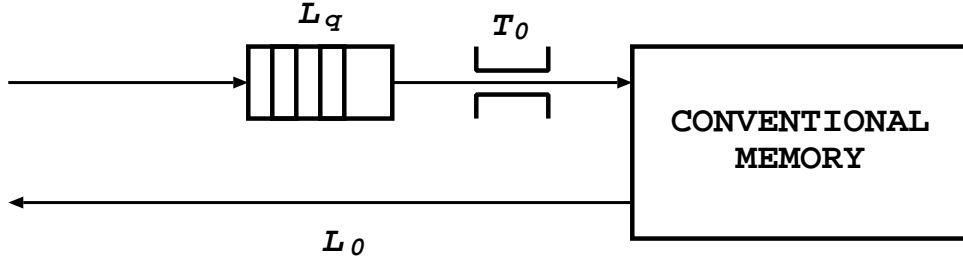


Figure 7: Conventional memory system

Each unfinished transaction represents a sleeping thread. We assume here that a thread is ready for execution unless it has placed a request into the memory system. Let S be the probability that a particular thread is sleeping. The average number of sleeping threads in the system is therefore Sn . At the same time it is equal to the average number of transaction contained in the black box. Therefore,

$$GL(G) = Sn. \quad (2)$$

On the other hand, the probability of the processor being idle (because all threads are sleeping) is equal to the normalised performance loss:

$$S^n = 1 - \frac{G}{R}. \quad (3)$$

From the (2) and (3) we obtain the fundamental equation of statistical balance:

$$GL(G) = n \left(1 - \frac{G}{R}\right)^{\frac{1}{n}}. \quad (4)$$

This equation for function $G(n)$ has, of course, to be solved numerically. But before this can be done, we have to know how latency L of the memory system depends on the granted throughput G . In the following section we will address this question for two important types of memory systems: conventional memory and network.

3.1 Latency as a function of granted throughput

3.1.1 Conventional memory

In the conventional memory system (see Fig. 7), the latency L_0 of the memory itself does not depend on the request rate. However, as the throughput of such memory is limited by the constant T_0 , requests have to be queued and the total latency is the sum of the queueing time and latency proper:

$$L(G) = L_q(G) + L_0. \quad (5)$$

We assume here that memory cycle begins immediately after the first request has arrived to the queue and that the memory takes a request out of the queue for service. The average queue length is therefore equal to the number of waiting customers in the system with deterministic service time¹. The average queueing time for this situation is

$$L_q = \frac{G}{2T_0(T_0 - G)}. \quad (6)$$

Therefore, equation (4) takes for the conventional memory the following form:

$$G \left(L_0 + \frac{G}{2T_0(T_0 - G)} \right) = n \left(1 - \frac{G}{R} \right)^{\frac{1}{n}}. \quad (7)$$

The maximum sustained throughput is in this case defined by the hardware characteristics of memory: $T_{max} = T_0$.

3.1.2 Network

Network model

When analysing the network, we will concentrate only on the fundamental features of the network structure itself, not taking into account such factors as contemporary technology constrains etc. In our model, the network is synchronous and messages make one hop in one network cycle. We assume that the network is direct and each node is connected to each of the K adjacent nodes by a pair of channels: input and output. Routers have two queues for each output channel: one for transit messages and another for messages injected by the processor connected to the router. The transit queue is granted the higher priority, that is the internal queue is served only when the transit queue is empty (in fact, relative priorities of transit and locally-injected messages make very little difference for the final result, but the above convention simplifies the calculations). The network is presumed to be loaded by random uniform traffic, and the routing strategy comprises a random choice between the channels prefixing any shortest path to the target.

Let D be the doubled average distance between two randomly chosen nodes in the network (for symmetrical networks it is equal to the diameter). Each transaction is of request-reply type (remote memory LOAD or STORE with confirmation). Thus, each processor effectively sends a message to itself via random route; the average message hop count is therefore equal to D . We ignore the request processing time at the target node assuming that it is done by independent hardware. So, from the traffic point of view, the request passes the target node as a part of transit traffic.

¹This and similar issues are discussed in queueing theory; see, for instance, [17].

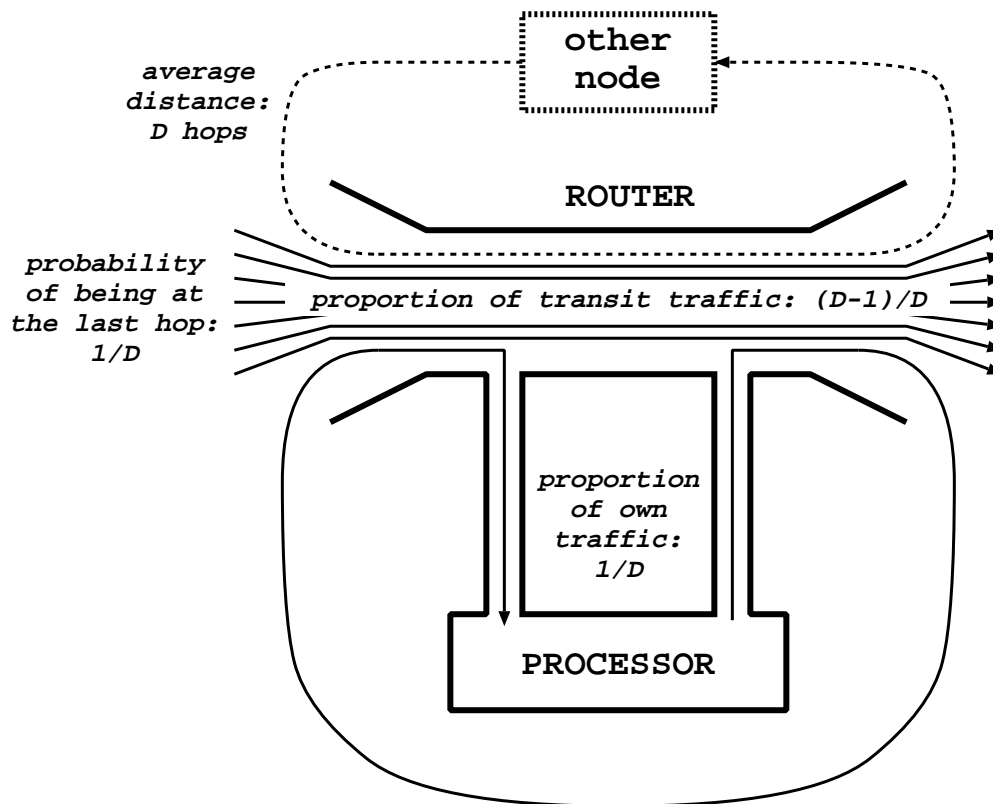


Figure 8: Proportion between transit traffic and “own” messages

Network behaviour: does the size matter?

In order to understand basic properties of the network acting as a memory subsystem, let us find an approximation of the number of requests from a given node that can be simultaneously active in the network.

Since any message makes on average $D - 1$ transit hops before being consumed at the final hop, only $1/D$ part of the incoming messages are consumed by any given node while $(D - 1)/D$ part are forwarded². Due to the statistical balance between the incoming and outgoing transit flows, the proportion of the node’s “own” messages in the outgoing traffic is also $1/D$: see Fig. 8.

In the following analysis, we take the network cycle for the unit of time. Let $r < 1$ be the observed average traffic rate in the network; due to the random uniform nature of traffic, this average figure is the same for all communication channels. According to the proportion between transit and injected/consumed traffic flows discussed above, the rate of transit traffic

²This is, in fact, only an approximation (though a rather precise one). The exact proportion depends on the topology of the network, and its analysis is beyond the scope of this article.

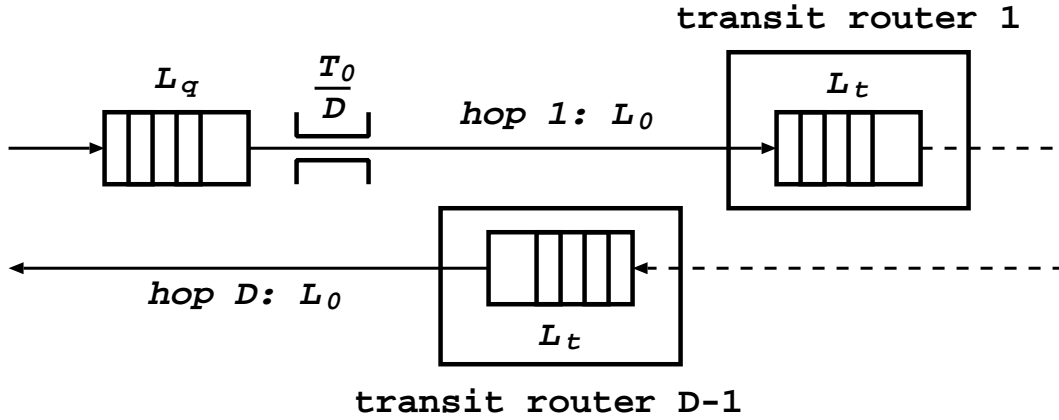


Figure 9: Output channel as seen by a processor

is

$$t = \frac{D-1}{D} r. \quad (8)$$

For a given processor, each output channel of the respective router can be considered a black box with the input rate of r/D and some latency which is proportional to the average number of hops D (at this stage we are ignoring initial queueing of requests at the originating node). So, as a first order approximation, we can conclude that the number of messages contained in all K channels is

$$GL(G) \approx K \frac{r}{D} O(D) = K O(r). \quad (9)$$

This means that, contrary to intuitive expectation, the average number of requests from a given processor contained in the network does not grow with the size of the network. The reason is that due to the necessity to support transit, the granted throughput of each outgoing channel falls in inverse proportion to D (while average latency is proportional to it). Note that this fact is invariant to the degree of spatial locality, since variation of the locality scale leads to the recalculation of the average route length, D , that does not affect the figure.

In order to find the actual form of the function $L(G)$ for the network, we have to take into account queueing times of both the internal queue and the transit queues.

More accurate analysis

The model of an output channel is shown in Fig. 9. Each message spends some time L_q in the internal queue at the original node and then performs an average of D hops. Each hop takes time L_0 ; as we have chosen the network cycle for the time unit, $L_0 = 1$. On its way, the message passes an average of $D-1$ transit nodes and spends some time L_t in the transit queue in each

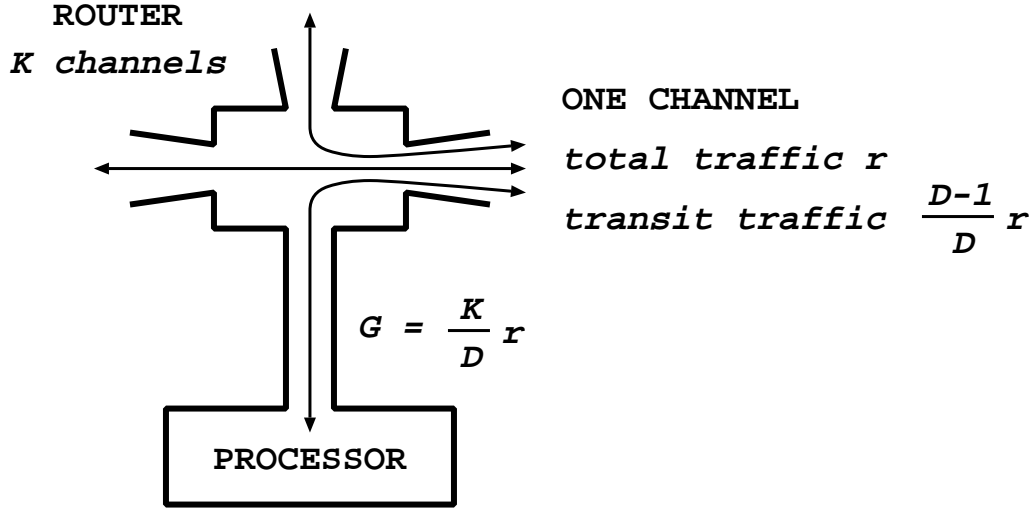


Figure 10: Granted request rate and network traffic rate

of them. Therefore, the total latency is given by the following formula:

$$L(G) = L_q(G) + D + (D - 1) L_t(G). \quad (10)$$

The channel serves the internal queue with the Poisson distributed service rate $1 - t$. Since a message is removed from the queue only by the end of the service period (transit priority), the internal queue latency is

$$L_q = \frac{1}{1 - t - \frac{r}{D}}. \quad (11)$$

The transit queue is served with deterministic service rate T_0 which in the chosen units of time is, of course, equal to 1: one hop at each network cycle. Thus, a non-zero transit queue length can be produced only by the collision of several messages queueing at the same output channel. Unfolding the instantaneous queueing process in discrete time, we observe that we are effectively dealing with the average number of waiting customers in a system with deterministic service rate (see [17]). Taking into account the discrete nature of the process, we obtain

$$L_t = \frac{t(1 - \frac{1}{K})}{2(1 - t)}. \quad (12)$$

Note that the network traffic rate r and the granted throughput G have an obvious relationship (see Fig. 10):

$$G = \frac{K}{D} r. \quad (13)$$

Using equations (8) and (13) to eliminate parameters t and r from the queue length formulas,

we obtain the following final form of the fundamental equation (4):

$$G \left(\frac{K}{K - GD} + D + \frac{G(D - 1)^2(1 - \frac{1}{K})}{2(K - G(D - 1))} \right) = n \left(1 - \frac{G}{R} \right)^{\frac{1}{n}}. \quad (14)$$

The maximum throughput available for a processor is

$$T_{max} = \frac{K}{D} T_0, \quad (15)$$

where T_0 is the physical throughput of one channel (this formula does not depend on the choice of the unit of time).

3.1.3 Combination of memory and network

$L(G)$ for the combination of memory and network (which is the architecture of contemporary massively parallel computers) depends on the degree of spatial locality; the two systems considered above correspond to full locality and zero locality, respectively.

3.2 Numerical solutions

Functions $P(n)$ corresponding to the results of solving equations (7) and (14) numerically are presented in Fig. 11 and Fig. 12, respectively. The first of these figures corresponds to conventional memory, the second to nearly any feasible symmetric network (note that for the network the number of threads is expressed per bidirectional channel in a router). For very small networks the graphs for $P(n)$ deviate from those shown in Fig. 12 due to the impact of terms of the order of $(D - 1)/D$, but generally $P(n)$ has very little to do with D — as it was suggested by the approximate formula (9).

As expected, in our contiguous model the asymptotic performance (1 for $R \leq T_{max}$ and T_{max}/R for $R > T_{max}$) is possible only with infinite number of threads. However, the 80% level with respect to the theoretic maximum is achieved with just a few threads for the conventional memory and with about 2 threads per channel for the network.

The conclusion that can be made from these results is that using heavy-weight multithreading for tolerating latency is not justified. The limited number of threads that is sufficient for fairly efficient execution should apparently be drawn from one referential environment with common registers and common stack, which will provide for low-overhead management of threads.

Similar conclusion was made by Culler in [7]. However, his analysis of the maximum number of outstanding requests is based on the technological limitations of existing network-based memory systems. Our “80% at 2 threads per channel” figure has nothing to do with those limitations. It is, in fact, a fundamental implication of the fact that the majority of random uniform traffic is in transit.

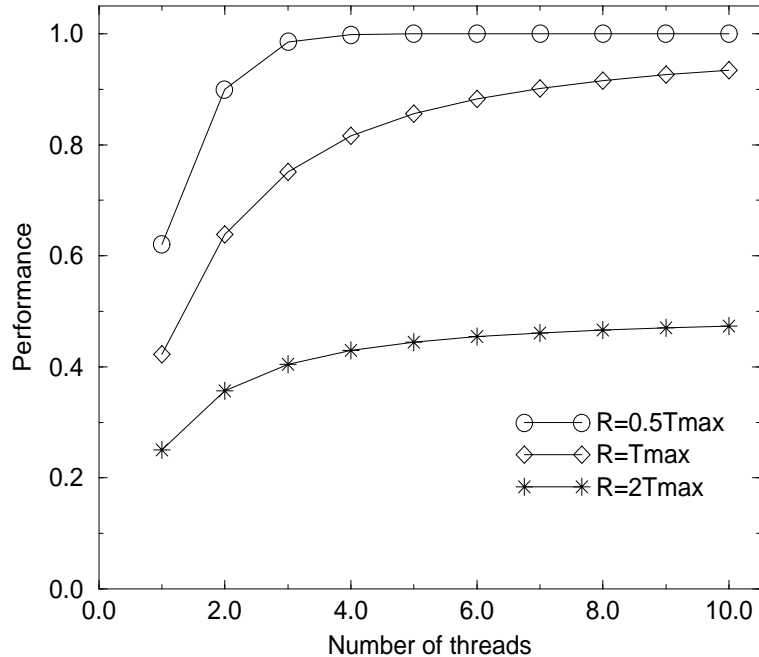


Figure 11: $P(n)$ for conventional memory with $L_0 = 1/T_0$

4 Discussion

In section 2 we have introduced mechanisms which can be applied to conventional RISC architectures to allow micro-threading, i.e. multi-threading within a single context. We have also shown that the cost of the primitives required is small and that we can expect to obtain a substantial fraction of peak performance with surprisingly few threads. However, there are a number of issues which may provide problems when compared to conventional designs and these are discussed below together with some possible solutions and future developments.

Spatial locality of instruction stream

The rotation of several threads in the pipeline obviously damages the spatial locality of instruction cache access. However, the effects of this in micro-threading are less likely to be observed than when threading on larger contexts. One straightforward solution is to use a fast, associative level 0 cache (or refill buffer) that can keep track of several points of control. However, when we miss all levels of instruction cache memory, a long stall is unavoidable. In the dynamic scheduling model we can go further. Since the hardware is not obliged to take PCs from the

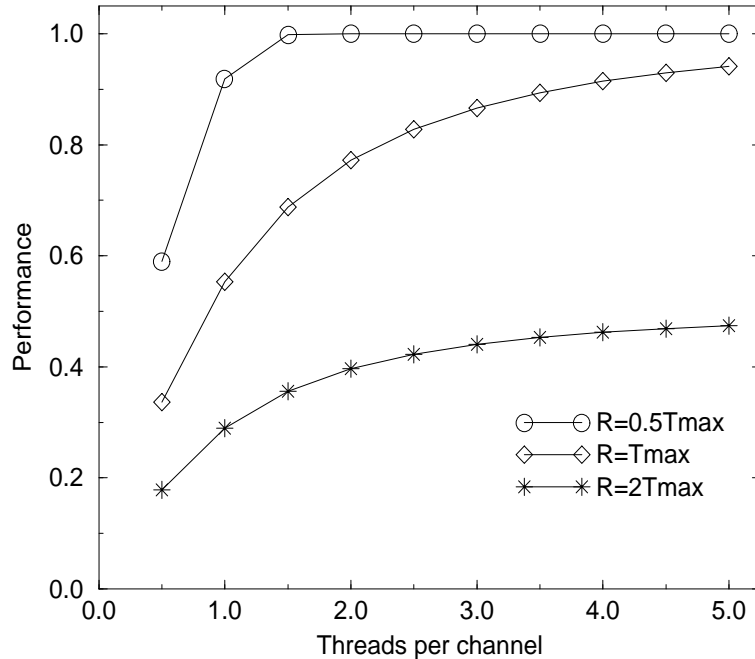


Figure 12: $P(n)$ for network

continuation queue in any strict order, several instructions from different threads can be fetched simultaneously and issued at will. This does however complicate the level 0 cache design.

Interrupts and context switching

Interrupt hardware in this architecture is more complex. To be able to stop the pipeline instantly, we require a recovery mechanism that can both track down the vertical PC issued by a speculatively executed instruction and keep the list of threads being executed. There is a range of simpler solutions affecting the interrupt response time and the complexity of interrupt processing.

The continuation queue and the register tags put an additional burden on context switching. The tags can be organised as a separate memory structure that can be accessed in two ways: horizontally (together with a register) and vertically (32 tags per word).

Overcoming a temporal narrowing of parallelism

When the available parallelism is dynamically not large enough to tolerate the internal pipe delay, we can apply a number different strategies in order to perform at least not worse than

the conventional RISC pipeline.

The simplest solution is to replace vertical transfer of control by horizontal one if the continuation queue is empty. In many cases, it is better to stall on possible data dependency.

In the most complex solution, any PC, no matter whether it is issued vertically or horizontally, is forwarded immediately to the parallel multi-fetching logic, which is shared by all the processing pipelines. The instruction which can be processed with the shortest stall time is issued first. Conditional branches are predicted only if postponing the decision damages the overall performance (i.e. there is lack of parallelism). A full-power back-trace recovery mechanism has to be provided if this approach is adopted. This mechanism prevents an instruction from any unrecoverable action until the direct ancestor of the instruction has reached the retirement point. Such a mechanism is feasible, but keeping it out of the critical paths is a non-trivial design challenge.

5 Conclusions

This paper has introduced and justified novel architectural techniques for micro-threading, which we define as multi-threading within a single context. This solution has an extremely small cost in terms of additional cycles required and can be implemented over conventional RISC designs with few modifications to the instruction set. An analysis has shown that this solution to latency tolerance is quite viable in both conventional and network based memory systems, due to the small number of threads required. The results of this analysis are surprisingly insensitive to architectural parameters because they are predicated on two fundamental facts, namely the exponential fall of the probability of stalling with the number of threads and the proportion of transit traffic found in a router node. These first order effects both contribute to the small number of threads required to reach a substantial fraction of the maximum possible performance.

This work has the ability to impact over the complete range of architectures used commercially today. In cheap PC based systems, where little or no cache is used, concurrency may be used to mask the relatively high frequency of accesses to main memory and thereby mitigate any performance loss. This is also the case where cache is available but that the nature of the problem or algorithm means that locality is very difficult to find. At the other end of the scale micro-tasking provides an architectural means to tolerate the very high latency and dispersion found in network based shared memory solutions. We have shown in our work on the compilation of of data-parallel languages [5, 14] that we can exploit the parametric parallelism found in this style in generating and if necessary throttling the large number of threads that this programming paradigm yields.

References

- [1] Abraham, S.G, Sugumar, R.A., Rau, B.R. and Gupta, R., “Predictability of Load/Store Instruction Latencies”, *Proc. of the 26th Int. Symp. on Microarchitecture*, Dec. 1993, pp. 139-152.
- [2] Agarwal, A., “Performance Tradeoffs in Multithreaded Processors”, *IEEE Trans. PDS* **3/5**, Sep. 1992, pp. 525-539.
- [3] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. and Smith, B., “The Tera Computer System”, *Proc. ACM Int. Conf. on Supercomputing*, Jun. 1990, pp. 1-6.
- [4] Barsky, D.B. and Shafarenko, A.V., “Uniform Random Traffic in Massively-parallel Data-driven Computer”, Tech. Rep. CSRG95-12, Dept. EE Eng., University of Surrey, Dec. 1995.
- [5] Bolychevsky, A., Jesshope, C.R. and Shafarenko, A.V., “Fundamental Issues in Designing Data-parallel Dataflow Computers”, to be published *Proc. IEE part E, CDT*, 1996.
- [6] Chen, T.F. and Baer, J.L., “A Performance Study of Software and Hardware Prefetching Schemes”, *Proc. of the 21st Ann. Int. Symp. on Computer Architecture*, Apr. 1994, pp. 223-232.
- [7] Culler, D.E., “Multithreading: Fundamental Limits, Potential Gains, and Alternatives”, in: Iannucci, R.A., Gao, G.R., Halstead, R.H. and Smith, B. (*eds.*), **Multithreaded Computer Architecture: A Summary of the State of the Art**, Kluwer Academic Publishers, 1994, pp. 97-138.
- [8] Damianakis, S., Li, K. and Rogers, A., “An Analysis of a Combined Hardware-Software Mechanism for Speculative Loads”, Tech. Rep. TR-455-94, Princeton University, Princeton, NJ, Apr. 1994.
- [9] Edmondson, J.H., *et al.*, “Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor”, *Digital Technical Journal* **7/1**, 1995, pp. 119-132.
- [10] Fu, J.W.C. and Patel, J.H., “Data Prefetching in Multiprocessor Vector Cache Memories”, *Proc. of the 18th Ann. Int. Symp. on Computer Architecture*, May 1991, pp. 54-63.

- [11] Fu, J.W.C., Patel, J.H. and Janssens, B.L., “Stride Directed Prefetching in Scalar Processor”, *Proc. of the 25th Int. Symp. on Microarchitecture*, Dec. 1992, pp. 102-110.
- [12] Gaudiot, J.L. and Bic, L. (*eds.*), **Advanced Topics in Data-Flow Computing**, Prentice Hall, 1991.
- [13] Iannucci, R.A., Gao, G.R., Halstead, R.H. and Smith, B. (*eds.*), **Multithreaded Computer Architecture: A Summary of the State of the Art**, Kluwer Academic Publishers, 1994.
- [14] Jesshope, C.R. and Sutton, C.D., “Compiling Data-parallel Languages for Distributed Memory Multi-processors”, *Proc. of the 5th Workshop on Compilers for Parallel Computers*, Malaga, 1995.
- [15] Jouppi, N.P., “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, *Proc. of the 17th Ann. Int. Symp. on Computer Architecture*, May 1990, pp. 364-373.
- [16] Kaigi, B.D. and Goodman, J.R., “Memory Bandwidth Limitations of Future Microprocessors”, Tech. Rep. No 1295, Dept. Comp. Sci., Univ. Madison, 1995.
- [17] Kleinrock, L., **Queueing Systems**, Wiley-Interscience, 1975.
- [18] Kroft, D., “Lockup-Free Instruction Fetch/Prefetch Cache Organization”, *Proc. of the 8th Ann. Int. Symp. on Computer Architecture*, May 1981, pp. 81-87.
- [19] Palacharla, S. and Kessler, R.E., “Evaluating Stream Buffers as a Secondary Cache Replacement”, *Proc. of the 21st Ann. Int. Symp. on Computer Architecture*, Apr. 1994, pp. 24-33.
- [20] Smith, B.J., “Architecture and Applications of the HEP Multiprocessor Computer System”, *Proc. SPIE* **298**, Aug. 1981, pp. 241-248.
- [21] Sohi, G. and Franklin, M., “High-Performance Data Memory Systems for Superscalar Processors”, *Proc. of the 4th Symp. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991, pp. 53-62.