Survey of High-Latency Tolerance in Future Microprocessor Architectures

Firas Al-Ali & Chris Jesshope
F.M.Al-Ali@massey.ac.nz, C.R.Jesshope@massey.ac.nz

Institute of Information Sciences and Technology,
Massey University, Palmerston North, New Zealand

Abstract

*This paper continues a survey presented in our previous paper [4]. Here, we survey future microprocessor architectures from the viewpoint of their different techniques used in tolerating highly-latent and non-deterministic events. Each architectural approach is presented with a brief discussion and an example commercial implementation or at least a proposed one. Dataflow architecture is revisited first. Then, Advanced Superscalar architectures are covered, which include the Superspeculative architecture, the Trac**e** architecture, the Multiscalar architecture, the Datascalar architecture, and the Superthreaded architecture. Following these Superscalar-based architectures, we move on to the Multithreading-based approaches, and review the different techniques. It is shown that the solution to the latency-tolerance problem by each one of these previous architectures is offset by the high overheads of the dataflow approach, the speculation involved in the superscalar approach, and the context switch time introduced by the multithreading architecture. This is where our own work on Microthreading is introduced to provide a new approach towards highly efficient latency-tolerance and elimination of non-determinism through the use of Microthreads drawn from the same context. Finally, a comparison between all these architectures is derived, and a conclusion is given.*

Keywords: Microprocessors, Architectures, RISC, Latency-tolerance, Microthreading

1. INTRODUCTION

At least four classes [1,8] of future possible developments for micro-architectures can be distinguished; all of which continue the ongoing evolution of the von Neumann computer introduced back in 1946:

- Microprocessors that retain the von Neumann architectural principle of Result Serialisation (where the order of the instruction flow as seen from the outside by the compiler still retains the sequential program order), despite the inherent use of out-of-order execution within the microprocessor. Microarchitectures that belong to this class are today's Superscalar architectures. There is still a considerable effort being directed towards improving such architectures, eg. Superspeculative [22,23,24], Multiscalar [28,29,30], Trace [25,26,27], Datascalar [31], and Superthreaded [1,5]. All these approaches belong to the same class because the result serialisation must be preserved. A reordering of results is performed in a Retirement or Commitment phase in order to fulfil this requirement. These architectures are discussed in this paper.

- Microprocessors that modestly deviate from the von Neumann architecture but allow the use of sequential von Neumann languages. Programs are compiled to the new instruction set principles. Examples of such architectures are Very Long Instruction Word (VLIW), Single Instruction Multiple Data (SIMD), and vector operations. We have covered VLIW in [4] and will revisit it again when discussing the MAJC multiprocessor (which is also a Multithreaded architecture). The vector approach is not covered in this paper, but is implemented in our Microthreading architecture discussed later.

- Microprocessors that optimise the throughput of a multi-programming workload by executing multiple threads of control simultaneously. Each thread of control is a sequential thread executable on a von Neumann computer. Two example architectures are our Microthreaded approach and the Chip Multi-Processor (CMP). Although we covered multithreaded architectures in [4], we discuss in this paper their future. CMP is not discussed explicitly, but rather implicitly as it is tightly coupled with multithreading.

- Architectures that totally break away from the von Neumann architecture and that need to use new languages, such as Dataflow with Dataflow Single-Assignment Languages (SALs). Here also, we have already covered dataflow in [4] but will revisit it to describe future developments.

One motivation for this diversity of approaches, is that of tolerating high latency and non-determinism in executing instructions, such as the latency in the access to main memory, responding to branches in control, or performing a floating-point division. Another example of non-determinism is the result of statically scheduling

concurrent operations and the need for synchronisation between these operations. Scheduling is the process of assigning specific instructions and their operand values to designated resources at designated times [6].

In the remaining sections of this paper, these future micro-architectures are reviewed along with a description of how they differently address the above issues of latency tolerance.

2. FUTURE MICRO-ARCHITECTURES

2.1 DATAFLOW ARCHITECTURE

Dataflow computers exploit all the parallelism available (finest-grained) in a program through the use of Dataflow Graphs as their machine language. Please refer to [4] for a rather more detailed description of dataflow architecture. The problems we raised there, of the need for deep pipelines, a high ratio of non-productive instructions, and the need for expensive matching logic, have resulted in new developments in dataflow architecture. The Monsoon Multiprocessor [11,12,13,38] is of direct relation to this paper and will be discussed below. It introduces two main features:

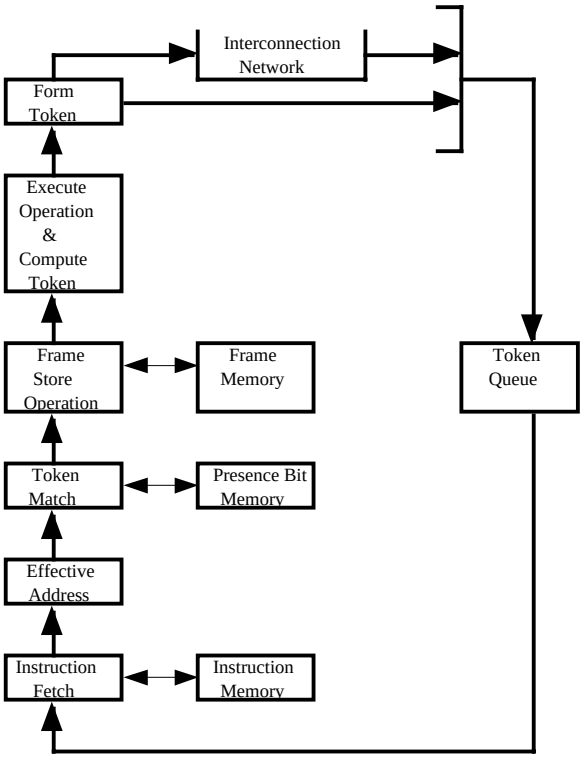

Figure 1: Monsoon execution pipeline.

- **Explicit Token-Store (ETS) Dataflow:** This is an evolution from the tagged-token dynamic dataflow principles [9]. ETS was the result of G M Papadopoulos' work [11,38] and was later incorporated in the Monsoon multiprocessor, the latter being the product of a joint effort of the MIT Computation Structures Group, and the Motorola company [12,13] after ETS was developed. The earlier tagged-token dataflow architectures used an associative matching store to determine when instructions are ready for execution. For a two-operand instruction to become enabled, tokens carrying the two-operand values must be received. The first token to arrive is held in the matching store until its mate has arrived. Two tokens provide a pair of operands to the instruction if they have the same tag. In ETS, the associative search for matching tokens is replaced by establishing a memory location where each synchronization takes place. This location is within an Activation Frame associated with each function activation. Thus, the waiting token memory is directly addressed, without the need for associative matching, which is slow and expensive [10]. Every location that corresponds to an activation of a two-input operation is augmented with a Presence Bit that is initially in the "empty" state. When the first token arrives, it notices the "empty" state and writes its value into the location. Then the presence bit is set into the "present" state. The second token notices the "present" state and reads the location, and then resets the presence bit back to "empty". The operation defined by the instruction (from the tag) is performed on the incoming token's value and the value read, and a new token is generated with the resulting value. Details of how ETS works are found in [11,38]. This ETS model is what is implemented in Monsoon**.** The benefit of eliminating the associative search is obtained, generally, for the cost of dynamically allocating storage for an activation frame at each function initiation (Preallocation is possible when a static analysis of the program shows that this is feasible [9]). Another innovation in the Monsoon is its support for I-structures, which refer to arrays of data in which reads of an element are made to wait until the element is defined by a write operation. We will see later that registers in our Microthreaded architecture are in fact I-structures.

- **Threaded Dataflow (Augmenting Dataflow with Control-flow):** Monsoon is a multithreaded architecture which evolved from the dataflow model [10]. The maximum configuration built consists of 8 processors and 8 I-structure memory modules using an 8x8 crossbar network, and became operational in

1991. The Monsoon can be viewed as a cycle-by-cycle-interleaving, multithreaded computer due to its capability of Direct Token Recycling [13]. Direct token recycling allows a particular thread to occupy only a pipeline frame slot in the 8-stage pipeline, which implies that at least 8 threads must be active for full pipeline utilization to be achieved [1]. *Figure 1* shows the Monsoon execution pipeline. The question that arises here is how are threads tracked and scheduled in this threaded dataflow architecture, since there are no PCs involved to refer to? The answer is related to the fact that massively parallel processors operate asynchronously in a network environment, where the asynchrony triggers the two fundamental latency problems: Remote Loads and Synchronization Loads [10]. Threaded dataflow is one solution to this problem, by multiplexing among many threads (when one thread issues a remote-load request, the processor switches to another thread) where "full/empty" bits present in memory words are used to synchronize remote loads associated with different threads [10]. In threaded dataflow, the threads are tracked and scheduled by associating each remote load and response with a thread identifier (referred to as a "continuation on a message") for the appropriate thread, so it could be reenabled upon arrival of a response. A large hardware-supported Continuation Name Space is provided to name an adequate number of threads for remote responses [10]. Threads in Monsoon are actually short sequences of instructions that access the local variables of its function initiation from its activation frame, and pass intermediate results using a small register set, thereby eliminating the requirement for dataflow synchronization at each instruction of a program [9].

2.2 SUPERSCALAR ARCHITECTURES

Superscalar microprocessors are, in essence, implicit multiple-issue processors where the instructions are dynamically scheduled by the hardware. This architecture was described in [4], where we also concluded that some tolerance of high latency is achieved. With wider issue pipelines, the performance depends critically on branch prediction accuracy and scheduling, as there is always a limit on the window in the instruction stream. Although static branch prediction can be used here, the alternative of dynamic branch prediction usually delivers superior performance. Speculative instruction issue used in superscalar architectures, also introduces write-after-read and write-after-write hazards, which can severely restrict instruction issue rates without register renaming [3], which, again, adds complexity and cost to the design. Wider superscalar issue puts even more pressure on the compiler to deliver on the performance potential of the hardware. But data and control dependencies in programs, together with instruction latencies, offer an upper limit on delivered performance because the processor must sometimes wait for a dependency to be resolved, such as with a misspredicted branch.

The techniques of out-of-order execution and dynamic branch prediction attempt to predict the nondeterminism in the areas of cache accesses and branching, or try to mitigate against the effects of missprediction [2]. These techniques do increase the processing power of superscalar processors, but rarely by a factor proportional to the width of the pipeline used. Even with 4 or 8-way superscalar pipelines, it is difficult to obtain an IPC count of very much more than 2 [3]. Moreover, speculation makes a computer's performance application-dependent and large penalties are paid for misspredicition both in execution time and, perhaps more importantly, in silicon area as much logic is dedicated to the prediction mechanisms and to missprediction clean-up [3].

Most of current microprocessors utilize ILP by a deep processor pipeline and by the superscalar instruction issue technique. Future VLSI technologies will allow future generations of microprocessors to exploit aggressively ILP up to 16 or even 32 IPC. Due to technological advances, gate-delay will be replaced by an on-chip wire-delay as the main obstacle to increase chip complexity and cycle rate. The implication for the microarchitecture is a functionally partitioned design with strict nearest neighbourhood connections [1,8].

Following is a description of the future trends in superscalar architectures:

2.2.1 Advanced Superscalar Processors

Although this direction still focuses on the use of ILP together with speculation, these are wide-issue superscalars with an IPC up to 32 [21]. This is achieved through the use of features such as; a large sophisticated trace cache for providing a contiguous instruction stream (see section 2.2.3 for more details), a multi-hybrid branch predictor, a large number of reservation stations to accommodate approximately 2000 instructions, and 24 to 48 pipelined functional units. *Figure 2* depicts the internal architecture for such a processor.

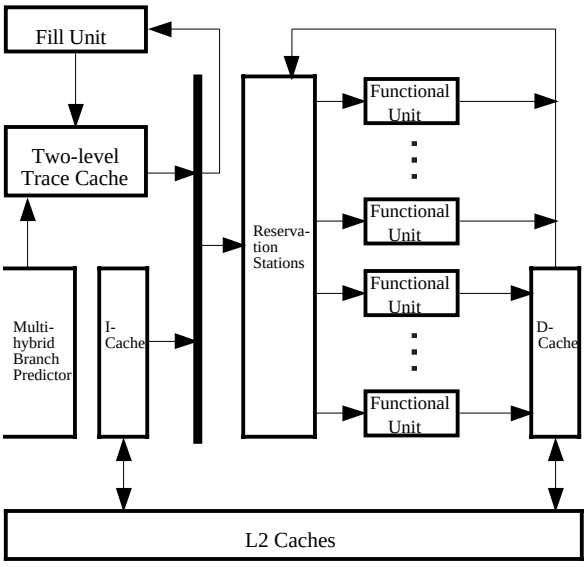

Figure 2: Advanced Superscalar Architecture.

2.2.2 Superspeculative Processors

These are also wide-issue superscalars that use speculation techniques [22,23,24]. This approach is based on the observation that in real programs, instructions generate many highly predictable result values. Therefore, consumer instructions can frequently and successfully speculate on their source operand values and begin execution without actual results from their producer instructions, thus removing the serialization constraints between producer and consumer instructions. As a result, it is claimed that superspeculative program performance can exceed the classical dataflow limit which states that even with unlimited machine resources, a program counter cannot execute any faster than the execution of the longest dependent chain introduced by the program's true data dependencies [1,8]. We still believe that the dataflow limit is so huge and unmatchable by any other architecture, and the problems are in managing the dependencies. Superspeculative processors speculate on data dependencies, instruction flow, register dataflow, and memory dataflow in addition to branch prediction [23]. This is all possible through the use of the Weak-Dependency Model [1,8,23], which specifies that dependencies can be temporarily violated during instruction execution as along as recovery can be performed prior to affecting the permanent machine state. If a significant percentage of speculations are correct, the machine can exceed the performance limit imposed by the traditional, Strong-Dependency Model.

2.2.3 Trace Processors

The trace processor is derived from the Multiscalar processor. The main ideas of this processor were presented in [25,26,27]. The trace processor extends the instruction window size to a trace, where traces are dynamic instruction sequences constructed and cached by the hardware. Thus, the trace cache stores dynamic instruction traces contiguously and fetches instructions from the trace cache rather than from the I-Cache. Since a dynamic trace of instructions may contain multiple taken branches, there is no need to fetch from multiple targets, as would be necessary when predicting multiple branches and fetching 16 or 32 instructions from the I-Cache. Trace processors also distribute the instruction window and register file to solve the instruction issue and register complexity problems (found in other designs such as Simultaneous Multithreading) by breaking up the processor into several PEs (similar to Multiscalar – see section 2.2.4) and the program into several traces so that the current trace is executed on one PE while the future traces are speculatively executed on other PEs. Because traces are neither scheduled by the compiler, nor guaranteed to be parallel, they still rely on control speculation and memory dependency speculation. The main difference between the Trace processor and Multiscalar processor is that the traces in a Trace processor are built as the program is executed, whereas the tasks in the Multiscalar processor require explicit compiler support.

2.2.4 Multiscalar Processors

The multiscalar model [28,29,30] represents another paradigm to extract a large amount of inherent parallelism from a sequential instruction flow. multiscalar and trace processors define several parallel processing cores, or PEs, that speculatively execute different parts of a sequential program in parallel. Multiscalar uses a compiler to partition the program segments, whereas a trace processor uses a trace cache to generate dynamically trace segments for the processing cores.

2.2.5 Datascalar Processors

Datascalar processors run the same sequential program redundantly across multiple processors using distributed datasets [31]. Loads and stores are only performed locally by the processor that owns the data, but a local load broadcasts the loaded value to all other processors. *Figure 3* demonstrates the execution of load and store operations for replicated and communicated memory. Assume that both processors execute a sequence of *load-1*, *store-1*, *load-2*, and *store-2*. Operations *load-1* and

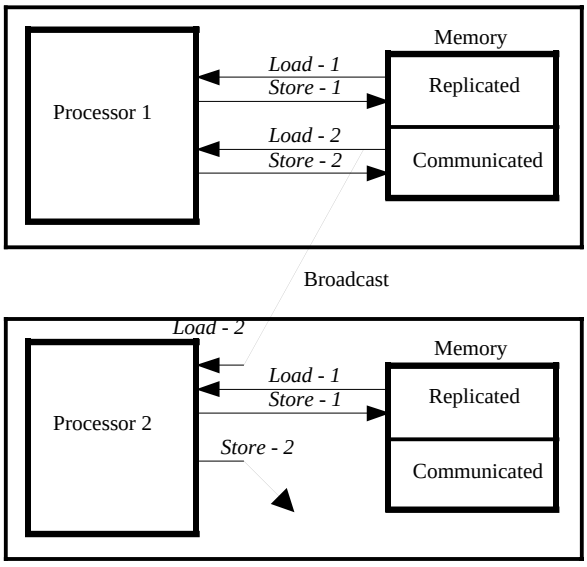

Figure 3: Datascalar processors access to replicated and communicated memory.

store-1 are issued to the replicated memory and can therefore complete locally on both processors. Operations *load-2* and *store-2* are issued to the communicated memory of the first processor. The *load-2* of this processor is deferred until the value is broadcast from it. Since all processors are running the same program, they all generate the same store value, which is stored only in the communicated memory of the processor that owns the address. Therefore, *store-2* is completed by the first processor, but is aborted on the second processor. It is quite clear here that the datascalar approach emphasises on more redundancy than performance. The speed-up here is not from ILP, but rather from data locality that is hiding latency to some degree.

2.2.6 Superthreaded Processors

This is a concurrent multiple-threaded architectural model for exploiting thread-level parallelism on a processor [1,5]. This architectural model adopts a thread pipelining execution model that allows threads with data dependencies and control dependencies to be executed in parallel, thereby enforcing data dependencies between concurrent threads. The basic idea of thread pipelining is to compute and forward recurrence data and possible dependent store addresses to the next thread as soon as possible, so the next thread can start execution and perform run-time data dependence checking on its own thread processing unit. Thread pipelining also forces contiguous threads to perform their memory write-backs in order, which enables the compiler to fork threads with control speculation. With run-time support for data dependence checking and control speculation, the superthreaded architectural model can exploit loop-level parallelism from a broad range of applications. The memory buffering and the in-order thread completion schemes allow control dependent threads to be executed concurrently with control speculation. Unlike the instruction pipelining mechanism in a superscalar processor, where instruction sequencing and data dependence checking and forwarding are performed by the processor hardware automatically, the superthreaded architecture performs thread initiation and data forwarding through explicit thread management and communication instructions. The execution of a thread is partitioned into several pipeline stages, each of them performing a specific function. The first pipeline stage is the Continuation Stage where a thread starts after being initiated by its predecessor thread. The next stage is the Target-Store-address-Generation (TSAG) stage which performs the address computation for Target Stores

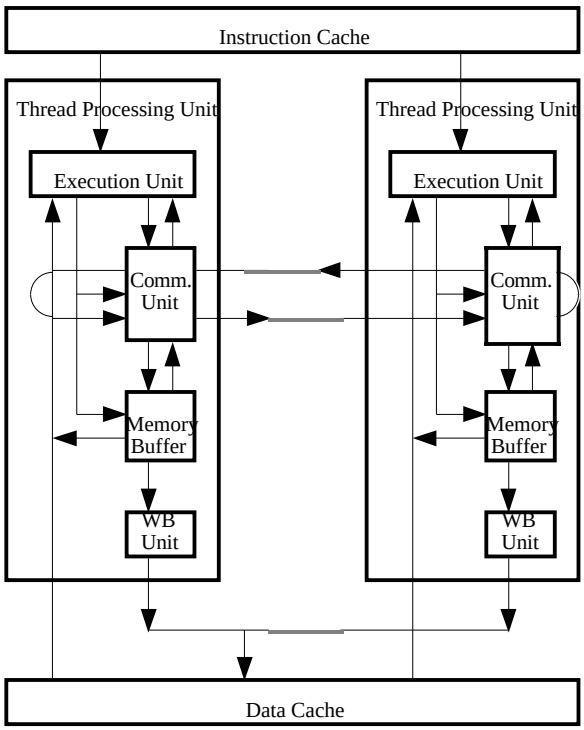

Figure 4: Superthreaded Processor.

(TSs). Target stores are performed by a thread, and are store operations on which later concurrent threads could be data-dependent. To facilitate run-time data dependence checking, the addresses of these target stores are calculated as soon as possible in the TSAG stage. The following Computation stage performs the main computation of a thread. The last stage is the Write-Back (WB). Because all of the stores are committed thread-by-thread, write-after-read 9anti-dependence) and write-after-write (output-dependence), hazards cannot occur during run time. *Figure 4* depicts the organization of the superthreaded processor.

2.3 MULTITHREADING ARCHITECTURES

We concluded earlier in [4] both superscalar and VLIW either attempt to predict the non-determinism in the areas of cache accesses and branching, or try to mitigate against the effects of misprediction (by executing both branches concurrently), prefetching, and speculative loading. Often however, these techniques introduce further speculation, which can have an even more detrimental effect on performance in the event of misprediction. So, it is quite obvious to us that microprocessor architects are looking in the wrong direction. They are designing processors that try to prejudge and predict a program's data accesses or branches, when instead; they should simply look at tolerating the latencies involved. The alternative approach is Multithreading, which is far from being a recent development.

Multithreaded architecture is one of the approaches that will be used in the processors of the near future, as a solution to the problem of limited ILP in a conventional instruction stream. A multithreaded processor is based on the philosophy of additional utilisation of more fine- to medium-grained parallelism. Full description of multithreading is given in [4].

Suffice it to say here that, when a long-latency or non-deterministic event occurs, such as branches and loads, the processor switches to another thread, executing instructions from that thread while the non-deterministic event is being handled. Context Switching must be very fast for this to be effective.

Still, the problem with the multithreaded architecture, in general, is that context switching might cause problems through the loss of cache locality. One alternative solution is our Microthreading approach, introduced in the next section.

The different multithreading architectural approaches are discussed below.

2.3.1 Cycle-by-Cycle Interleaving

In this model, the processor switches to a different thread after each instruction fetch. Some example implementations of this multithreaded architecture are:

- **Burton Smith's Delencor HEP:** The Heterogeneous Element Processor (HEP) [1,9,10,19,35] is a MIMD shared-memory multiprocessor system. Switching occurs between two queues of processes; one queue controlled program memory, register memory, and the functional memory while the other queue controlled data memory. The main processor pipeline had eight stages. Consequently, at least eight threads must be in cycle-by-cycle interleaving execution concurrently within a single HEP processor to give maximum performance. All threads within a HEP processor share the same register set. Multiple processors and data memories were interconnected via a pipelined switch and any register-memory or data-memory location could be used to synchronize two processes on a producer-consumer basis by a full/empty bit synchronization on a data memory word [1,7,35]. *Figure 5* illustrates the control loop for a single HEP pipeline. The 8-stage execution pipeline is shown, where IF denotes Instruction Fetch, DF for Data Fetch, INC for Increment, PSW for Process Status Word, and SFU for the shared

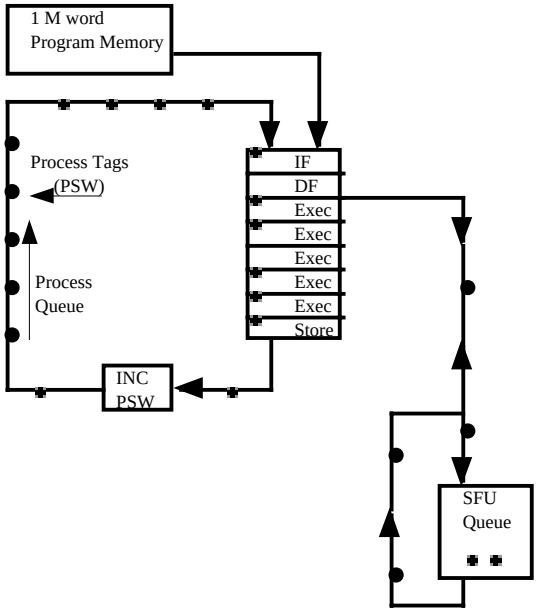

Figure 5: Control loop of a HEP pipeline.

memory. The pipeline is controlled by a queue of Process Tags (one for each thread representing an instruction stream). These process tags (or threads) rotate around the control loop, which executes one instruction from each thread every clock cycle. When an instruction accesses memory, it is removed from this loop and waits on memory in another queue (SFU queue). This is similar to Vertical Multithreading (VT) as we will see in section 3, but with one main difference, being that the threads in HEP are stored in the memory queue, while in microthreading, the microthreads are stored in the registers. The problem with HEP is that to tolerate long memory access latencies, a large number of

threads and non-blocking memory accesses are necessary.

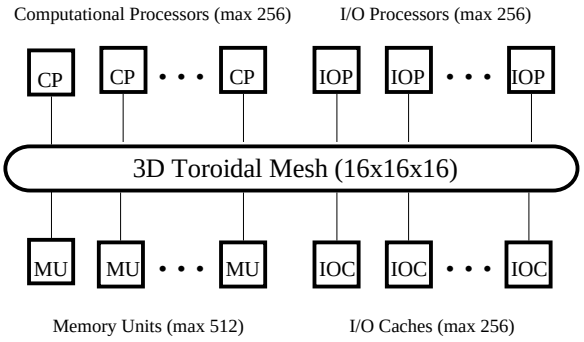

Figure 6: The Tera MTA 256 computer system.

- **Tera MTA Processor:** The Tera Multi-Threaded Architecture (MTA) computer features a VLIW instruction set, a three-dimensional toroidal interconnection mesh network of pipelined packet-switching nodes, uniform access time from any processor to any memory location, and zero-cost synchronization and swapping between threads of control [1,9,10,20,36]. The uniform access time is accomplished through distributing the resources (ie. processors, data memory units, I/O processors, and I/O cache units) uniformly throughout the network, instead of locating the processors on one side of the network and memories on the other. This permits data to be placed in memory units near the appropriate processor when that is possible and otherwise, generally, maximizes the distance between possibly interfering resources [20]. The Tera MTA exploits parallelism at all levels, from fine-grained ILP within a single processor to parallel programming across processors, to multiprogramming among several applications simultaneously. Consequently, processor scheduling occurs at many levels, and managing these levels poses unique and challenging scheduling concerns [36]. The Tera MTA contains 128 thread contexts and register sets per processor node to mask remote memory access latencies effectively. But still, this is considered too much overhead in order to tolerate such latencies, as 128 register sets is expensive to implement in logic. *Figure 6* shows the Tera 256 multiprocessor where the interconnection network is a 16x16x16 three-dimensional sparsely populated torus architecture, with 4096 pipelined packet-switching nodes. Every processor has a clock register that is synchronized exactly with its counterparts in the other processors and counts up once per clock cycle [1]. Since the average latency in the Tera is about 70 clock cycles, this means that when a latency occurs, this requires 70 different streams of instructions to be running on each processor to tolerate such latency. The Explicit-Dependence Lookahead technique detailed in [20] allows streams to issue multiple instructions in parallel; thereby reducing the number of streams needed to achieve peak performance.

2.3.2 Block Interleaving

In this approach, a single thread executes until it reaches a situation that triggers a context switch to another thread. Such situation could be a long-latency operation, which usually causes the pipeline to be flushed and a new register set is used. An example implementation of this multithreaded architecture is:

- **Sun MAJC-5200 Chip Multi-processor:** The Microprocessor Architecture for Java Computing (MAJC) processor architecture [18,38] from Sun Microsystems is based on a variable-length VLIW instruction set. Each Processing Unit (PU) contains 1 to 4 Functional Units (FUs). Each FU is viewed as a RISC processor in itself and is the basic building block of a PU. Individual instructions are issued to these FUs. Also, a new technique is introduced, called Space Time Computing (STC) is used to enable speculative threads (future instruction streams) to execute across separate processor units which substantially improves performance of many single-threaded and multithreaded applications. For example,if we have two processors on a chip, then two threads (Head and Speculative) execute on separate processors. They operate in a different space (speculative heap) and in a different time (future time) [37]. Also supported is Vertical Multithreading, where the CPU switches to a new instruction stream (thread) whenever there is a cache miss. Each processor can switch between four different threads. The large register file maintains these four thread references.

2.3.3 Nanothreading

Nanothreading [1,14] proposed for the Dansoft processor dismisses full multithreading for a ***nanothread*** that executes in the same register set as the main thread. The DanSoft nanothread requires only a 9-bit PC, some simple control logic, and it resides in the same page as the main thread. Whenever the processor stalls on the main thread, it automatically begins fetching instructions from the nanothread. Only one register set is available, so the two threads must share that register set. Typically, the nanothread will focus on a simple task, such as

prefetching data into a buffer, that can be done asynchronously to the main thread.

In the DanSoft processor, nanothreading is used to implement a new branch strategy that fetches both sides of a branch. A static branch prediction scheme is used, where branch instructions include 3 bits to direct the instruction stream. The bits specify eight levels of branch direction. For the middle four cases, denoting low confidence on the branch prediction, the processor fetches from both the branch target and the fall-through path. If the branch is mispredicted in the main thread, the back-up path executed in the nanothread generates a misprediction penalty of only 1 to 2 cycles.

The Dansoft processor proposal is a dual-processor CMP, called Dan 2433, each processor featuring a VLIW instruction set and the nanothreading technique. Each processor is an integer processor, but the 2 processor cores share a floating point unit as well as the system interface.

However, the nanothread technique might also be used to fill the instruction issue slots of a wide superscalar approach as in simultaneous multithreading. Finally, nanothreading is proposed in the context of a block-interleaving multithreading technique.

2.3.4 Simultaneous Speculation scheduling (S3)

S3 is a combined compiler and architecture technique to control multiple path execution [15]. The S3 technique can be applied to dual path branch speculation in case of unpredictable branches and to multiple path speculative execution of loop iterations. In this approach, separate threads are generated by the compiler that harness thread-level speculation by speculating on the outcome of branches executing in parallel on a multithreaded microprocessor. Loop-carried memory dependencies that cannot be disproven by the compiler are handled by data dependence speculation. The architectural requirements are the ability to run two or more threads in parallel and and three new instructions (fork, sync, wait) to control threads. This technique aims at simultaneous multithreaded, nanothreaded, and microthreaded processors, but can be modified for multiscalar, datascalar, and trace processors. Applying the S3 technique to branches in kernel sections of SPECint95 benchmark programs shows performance gains of up to 40% over purely static scheduling techniques.

2.3.5 Other Multithreading Architectures

Some new architectures based on the multithreaded trend are currently being developed, examples of which are:

- **SMT (Simultaneous Multi-threading)** processor [3,4,32,33], which is the result of combining multithreading with the superscalar technique, that leads to having all hardware contexts active simultaneously, competing each cycle for all available resources [1]. This is why SMT is also known as the **Multithreaded Superscalar** approach. SMT architecture implements a large register file; each thread can address 32 dedicated integer (and floating point) registers, and there are another additional 100 integer and floating-point renaming registers. Because of the access time of the larger register file, the SMT pipeline must be extended by using a two-cycle register read and a two-cycle register write [3].

- **DMT (Dynamic Multi-threading)** processor [3,4,34], which also uses an SMT pipeline to increase processor utilisation, except that the threads are created dynamically from the same program. The hardware breaks up a program automatically into loops and procedure threads that execute simultaneously on the superscalar processor.

3. MICROTHREADING

3.1 INTRODUCTION

Microthreading was introduced in [2,3,4,7,39]. Like nanothreading, microthreading is multithreading within a single context [2], where a thread (or sometimes we call it a **micro-thread**) is just a reference to a program counter. Non-deterministic events, such as branches and synchronizations which may fail, will cause a new thread (PC) to be executed, which may happen on every cycle. Thus, microthreading combines the best of both block and cycle-by-cycle thread interleaving techniques. The expectation is that such threads will be rather small, maybe only a few instructions long, and therefore, it is imperative that the overheads for fork, join and synchronisation are extremely low [2].

While existing multithreaded architectures are implicitly based on the assumption that latency tolerance requires massive parallelism, which must be found from diverse contexts, the quantitative analysis [2] carried out for the efficiency of multithreaded execution as a function of the number of threads, shows that there are fundamental reasons for the efficiency to grow very rapidly with the number of threads. This has been verified in [39] and, therefore, justifies the microthreading approach, where the original goal of latency tolerance is achieved with only relatively few threads; these can easily be drawn from within the same referential context and do not,

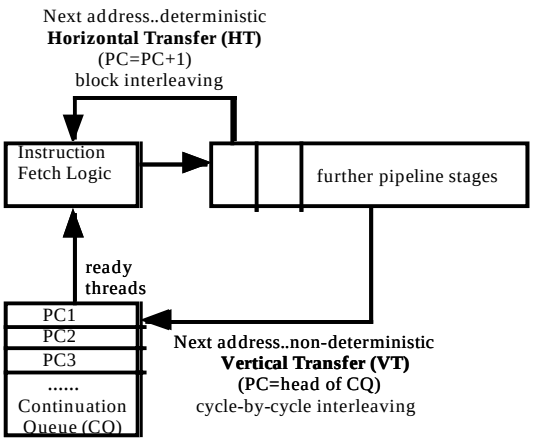

Figure 7: Microthreaded control transfer.

therefore, require the heavy weight hardware solutions of conventional multithreading [2]. This approach attempts to overcome the limitations of RISC instruction control (branch, loop, etc.) and data control (data miss, etc.) by providing such a low context switch time that it can be used not only to tolerate high latency memory, but also to avoid speculation in instruction execution [3]. It is, therefore, able to provide a more efficient approach to instruction pipelining. Microthreading performs true dynamic scheduling of several instruction streams by introducing the explicit notion of the manipulation of multiple program counters (PCs) by the processor. A PC represents the minimum possible context information kept for a given thread, and it is the only reference to a thread in the suggested microthreading architecture. Since several threads can be active simultaneously, an explicit storage for their PCs, called the Continuation Queue (CQ), must be provided. This is associated with the instruction fetch logic at the entry of the pipeline (as shown in *figure 7*).

In a normal RISC pipeline, the next address is transferred from the first stage of the pipeline in order to allow the next instruction to follow without delay. Branch instructions will normally involve speculation to predict the branch to be taken. If this prediction fails, any subsequent change of state must be "cleaned- up". We call this conventional mechanism of transferring control Horizontal Transfer (HT), and the alternative mechanism proposed here, which utilises the continuation queue, Vertical Transfer (VT). In a vertical transfer, the next instruction is fetched from the PC at the head of the continuation queue. This is performed on non-deterministic operations, like branch prediction, to avoid speculation.

3.2 MICROTHREADING ON A MULTI-CPU

This was discussed in [3,4]. Still, a recap is worthwhile here. Earlier, we proposed a two-level CQ (Continuation Queue). The first level holds ready threads that have not yet been allocated to any CPU. The register requirements for these ready threads are generic and subject to the limit of registers in any CPU's register file. We call this first level CQ the Global Continuation Queue (GCQ). From the GCQ, threads are allocated to a CPU but only when that CPU has resources available. Once allocated, the thread runs to completion on that CPU and is held in the CPU's own CQ, the second level, called the Local Continuation Queue (LCQ). Hence, there is a pool of unallocated threads in the GCQ, and a pool of allocated threads being held in the LCQ for each CPU. The allocation mechanism must ensure that register resources are available on the CPU where the threads are to be executed, and must rename the thread's local registers from their generic form to the actual registers allocated.

The proposed architecture for a single microthreaded CPU is shown in *figure 8*, while *figure 9* shows the multiple-CPU organization.

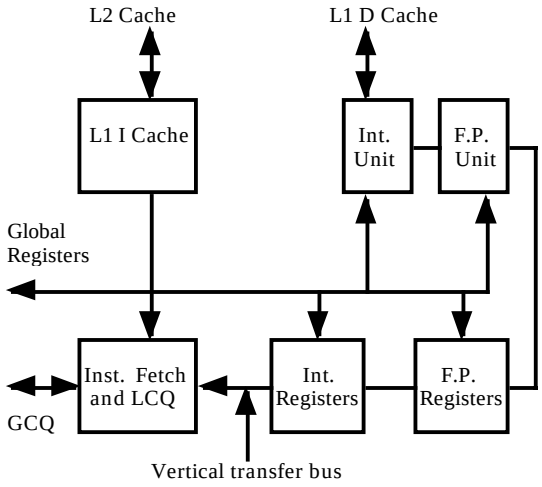

Figure 8: Microthreaded Processing Unit (PU).

3.3 MICROTHREADED VECTOR ISA

The following discussion is extracted from [7] where the microthreaded vector instruction set architecture is described in detail. This is a combination of two different techniques from quite different eras in terms of computer architecture; one of which, using a vector instruction set has a long history dating back to pipelined vector supercomputers, such as the Cray 1 and its successors. The other technique, multi-threading, is also well

understood. The combination can exploit both loop- and instruction-level parallelism without the need for speculation. This is important in the design of efficient chip-multi-processors, where large amounts of ILP are required. This novel approach proposed in [7] combines both vertical and horizontal microthreading with vector instruction descriptors, where it was shown that a family of threads can represent a vector instruction with dependencies between the instances of that family, the iterations. This technique gives a very low overhead in implementing an n-way loop and is able to tolerate high memory latency. The use of microthreading to handle dependencies between threads provides the ability to trade-off between instruction level parallelism and loop parallelism.

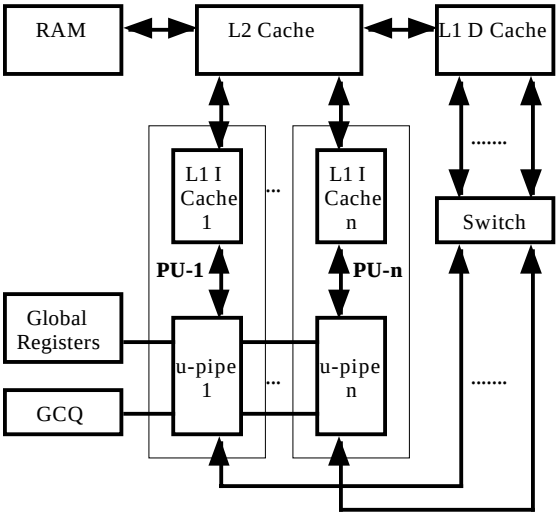

Figure 9: Microthreaded Multiprocessing CPU.

In a microthreaded, vector architecture, threads are used to execute multiple loop bodies simultaneously, which provides parallelism to:

- support multiple processors
- keep the pipelines full in the presence of both data and control dependencies, and
- tolerate high latency memory events.

To achieve parallelism on a large scale, it is imperative that just one instance of the loop body be used for all iterations. This is for reasons of code size and portability.

In pipeline vector architectures, such as the Cray 1 [7], vector instructions group single operations across the iterations in a loop. Thus a loop is transformed from a sequential execution model, where each instruction in the loop is executed for each iteration of the loop, to one where, each instruction in the loop is executed for all iterations of the loop before the next is executed. This grouping of the multiple scaler operations into vector operations allows the architecture to organise memory access and pipeline operation to achieve the optimal throughput of one cycle per operation, even for chained operations [35]. Because of the parallel semantics of this execution methodology, there can be no dependencies between loop iterations.

In a microthreaded, vector architecture, complete loop bodies can be executed in parallel for each loop index. This allows instruction level parallelism as well as loop parallelism to be exploited. Therefore any loop generates parallelism, even one containing a dependency between successive iterations. Code-generation techniques normally used to maximise ILP at compile time, such as loop unrolling and software pipelining occur automatically at run time through dynamic thread creation. Thus the depth of unrolling is determined by the resources available on the target processor rather than by the compiler, giving more portable code.

One problem faced, in devising a scheme to support the above execution model, is in the use of registers. This problem is addressed in [7,39], as well as the means by which instruction classes may be instanced as independent parallel micro-threads along with an illustration of the speed-up that may be obtained compared to using a conventional loop.

4. COMPARISON AND CONCLUSION

In this paper, we have surveyed how high-latency and non-deterministic events are tolerated in future dataflow, superscalar, and multithreaded architectures. It was shown that the solution to the latency-tolerance problem by each one of these previous architectures is offset by the high overheads of the dataflow approach, the speculation involved in the superscalar approach, and the context switch time introduced by the multithreading architecture. This is why we still insist that the introduced microthreading architecture is not only a new approach that achieves more ILP than a standard RISC pipeline, by totally eliminating speculation and replacing it with microthread interleaving, but is also the only architecture to address the latency-tolerance problem efficiently with minimum performance compromise. The overhead for microthreading was shown to be very small.

This was just a review. The research we are currently doing at Massey involves developing the VHDL simulator for the microthreaded pipeline. This simulator is still under development, and is currently defined at the behavioural level of VHDL abstraction. The next objective is to simulate at the RTL and Logic levels.

5. REFERENCES

1. J Silc, B Robic and T Ungerer, 1999, *Processor Architecture: From Dataflow to Superscalar and Beyond*, Springer-Verlag Berlin Heidelberg, ISBN 3-540-64798-8.
2. A Bolychevsky, C R Jesshope and V B Muchnick, 1996, *Dynamic Scheduling in RISC Architectures*, IEE Proc. Comput. Digit. Tech., Vol.143, No.5, September.
3. C R Jesshope and B Luo, 2000, *Micro-threading: A New Approach to Future RISC*, Proc. ACAC 2000, pp34-41, ISBN 0-7695-0512-0 (IEEE Computer Society Press), Canberra, January.
4. F M Al-Ali and C R Jesshope, 2000, *Survey of High-latency Tolerance in Contemporary Microprocessor Architectures*, Proc. 7th Annual New Zealand Engineering and Technology Postgraduate Conference, pp339-346, ISBN 0-473-07224-6, Massey University, Palmerston North, New Zealand, 23rd & 24th Nov.
5. J Y Tsai and P C Yew, 1996, *The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation*, Proc. 1996 Conference on Parallel Architectures and Compilation Techniques (PACT'96), pp35-46, Boston, MA, October.
6. M J Flynn, 1995, *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers, Sudbury, MA, ISBN 0-86720-204-1.
7. C R Jesshope, 2001, *Implementing an Efficient Vector Instruction Set in a Chip Multi-processor Using Micro-threaded Pipelines*, Proc. Australian Computer Systems Architecture, Gold Coast, Auastralia, February.
8. J Silc, T Ungerer and B Robic, 2000, *A Survey of New Research Directions in Microprocessors*, Microprocessors and Microsystems, 24, pp175-190.
9. R A Iannucci, G R Gao, R H Halstead Jr. and B Smith, 1994, *Multithreaded Computer Architecture: A Summary of the State of the Art*, Kluwer Academic Publishers, USA, ISBN 0-7923-9477-1.
10. K Hwang, 1993, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill Inc., USA, ISBN 0-07031622-8.
11. G M Papadopoulos, 1988, *Implementation of a General-Purpose Dataflow Multiprocessor*, Technical Report TR-432, Laboratory For Computer Science, MIT, Cambridge, MA.
12. G M Papadopoulos and D E Culler, 1990, *Monsoon: An Explicit Token-Store Architecture*, Proceedings of the 17th Annual Symposium on Computer Architecture, Seattle, WA, pp82-91, May.
13. G M Papadopoulos and K R Traub, 1991, *Multithreading: A Revisionist View of Dataflow Architectures*, Proc. of the 18th Annual International Symposium on Computer Architecture, Toronto, Ontario, pp342-351, May.
14. L Gwennap, 1997, *DanSoft Develops VLIW Design*, Microprocessor Report, 11(2), pp18-22, Feb. 17.
15. A Unger, T Ungerer and E Zehendner, The Website for *Compiler Techniques for Multithreaded Processors*, http://goethe.ira.uka.de/people/ungerer/mf-compiler/sss.html
16. D A Patterson and J L Hennessy, 1998, *Computer Organization & Design: The Hardware / Software Interface*, 2ed, Morgan Kaufmann Publishers, San Francisco, CA, ISBN 1-55860-491-X.
17. J L Hennessy and D A Patterson, 1996, *Computer Architecture: A Quantitative Approach*, 2ed, Morgan Kaufmann Publishers, San Francisco, CA, ISBN 1-55860-372-7.
18. M Tremblay, 1999, *MAJC-5200: A VLIW Convergent MPSOC*, Microprocessor Forum '99.
19. B J Smith, 1981, *Architecture and applications of the HEP multiprocessor computer system*, Proc. SPIE, 298, pp241-248
20. R Alverson, D Callahan, D Cummings, B Koblenz, A Porterfield and B Smith, 1990, *The Tera Computer System*, Proceedings of the ACM International Conference on Supercomputing, pp1-6.
21. Y N Patt, S J Patel, M Evers, D H Friendly, J Stark, 1997, *One Billion Transistors, One Uniprocessor, One Chip*, Computer, 30, pp51-57, September.
22. M H Lipasti and J P Shen, 1997, *The Performance Potential of Value and Dependence Prediction*, Lecture Notes in Computer Science, 1300, pp1043-1052, Springer-Verlag, Berlin.
23. M H Lipasti and J P Shen, 1997, *Superspeculative Microarchitecture for Beyond AD 2000*, Computer, 30, pp59-66, September.
24. M H Lipasti, C B Wilkerson and J P Shen, 1996, *Value Locality and Load ValuePrediction*, Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, pp134-147, October.
25. E Rotenberg, Q Jacobson, Y Sazeides, and J E Smith, 1997, *Trace Processors*, Proc. of the MICRO-30, Research Triangle Park, NC, pp138-148.
26. J E Smith, S Vajapeyam, 1997, *Trace Processors: Moving to Fourth-Generation Microarchitectures*, Computer, 30, pp 68-74.
27. S Vajapeyam and T Mitra, 1997, *Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences*, Proc. of the ISCA 24, Denver, CO, pp1-12.

28. M Franklin, 1993, *The Multiscalar Architecture*, Computer Science Technical Report No. 1196, University of Wisconsin-Madison, WI.

29. G S Sohi, 1997, *Multiscalar: Another Fourth-Generation Processor*, Computer, 30, pp72.

30. G S Sohi, S E Breach and T N Vijaykumar, 1995, *Multiscalar Processors*, Proc. ISCA-22, santa Margherita Ligure, Italy, pp414-425.

31. D Burger, S Kaxiras and J R Goodman, 1997, *Datascalar Architectures*, Proc. ISCA-24, Denver, CO, pp338-349.

32. S Eggers, J Emer, H Levy, J Lo, R Stamm, and D Tullsen, 1997, *Simultaneous Multithreading: A Platform for Next-generation Processors*, IEEE Micro, Sep/Oct, pp 12-18.

33. J L Lo, 1998, *Exploiting thread-level parallelism on simultaneous multithreaded processors*, Ph.D. Thesis, Dept. of Computer Science and Engineering, University of Washington.

34. H Akkary and M Driscoll, 1998, *A Dynamic Multithreading Processor*, 31st Annual ACM/IEEE International Symposium on Microarchitecture, Nov.30-Dec.2.

35. R W Hockney and C R Jesshope, 1988, *Parallel Computers 2: Architecture, Programming and Algorithms*, IOP Publishing Ltd., Bristol, ISBN 0-85274-811-6.

36. G Alverson, S Kahan, R Korry, C McCann, J B Smith, 1995, *Scheduling on the Tera MTA*, Lecture Notes in Computer Science, 949, Springer-Verlag, Berlin, pp19-44.

37. Sun Microsystems, *MAJC Architecture Tutorial (web site)*, http://www.sun.com/microelectronics/majc/documentation/docs/majctutorial.pdf

38. G M Papadopoulos, 1991, *Implementation of a General-Purpose Dataflow Multiprocessor*, Research Monographs in Parallel and Distributed Computing, The MIT Press, Cambridge, Massachusetts, Pitman Publishers, London, ISBN 0-273-08835-1.

39. C R Jesshope and B Luo, 2001, *Evaluation of Vector Instruction Set Micro-threaded Pipelines*, Submitted to Euro-Par 2001 Conference, Manchester, August.