JMA: The Java-Multithreading Architecture for Embedded Processors*

Panit Watcharawitch, Simon Moore
Computer Laboratory, University of Cambridge, UK
{Panit.Watcharawitch, Simon.Moore}@cl.cam.ac.uk

Abstract

Embedded processors are increasingly deployed in applications requiring high performance with good real-time characteristics whilst being low power. Parallelism has to be extracted in order to improve the performance at an architectural level. Extracting instruction level parallelism requires extensive speculation which adds complexity and increases power consumption. Alternatively, parallelism can be provided at the thread level. Many embedded applications can be written in a threaded manner in Java which can be directly translated to use hardware-level multithreaded operations. This paper presents an architectural study of JMA, a high-performance multithreaded architecture which supports Java-multithreading and real-time scheduling whilst remaining low-power.

1. Introduction

In processor design, *functional requirements* and *performance goals* are crucial factors. The functional requirements are inspired by the processor market. The trends from the market indicate that personal hand-held devices have an interesting potential to grow. These devices require advanced functional features like complex multimedia animation, images/audio encoding, cryptography and speech recognition/synthesis. Many of these complex applications are naturally written in a threaded manner (e.g. decoding each block of encoded animation or calculating each HMM node during speech recognition).

Today's high-end processors execute many instructions per cycle, in order to work effectively, speculative execution or branch prediction are required. Such techniques add a great deal of complexity, increase the power consumption, and make real-time performance difficult to predict. Also of great concern is memory latency, since there is a growing gap between processor cycle and memory access times at approximately 50% per year [1]. Memory latency should be hidden by increasing the parallelism in the system such that a pending load instruction does not stall the whole processor. However, memory latency tolerance is difficult within a single thread because of limited single-thread instruction level parallelism. Thus, the system that supports *thread level parallelism* seems to be the best alternative to provide latency tolerance and offer predictable performance gains.

For this study, we chose the Java concurrency model because it is increasingly used in embedded applications, and supports light weight threads through its Thread object class. We used the MIPS instruction set as a basis for our study, adding just four instructions (see section 3.4).

A brief background about Java-multithreading is presented in section 2. Section 3 illustrates the hardware architecture. Experimental results are presented in section 4 followed by the conclusion in section 5.

2. Java Multithreading

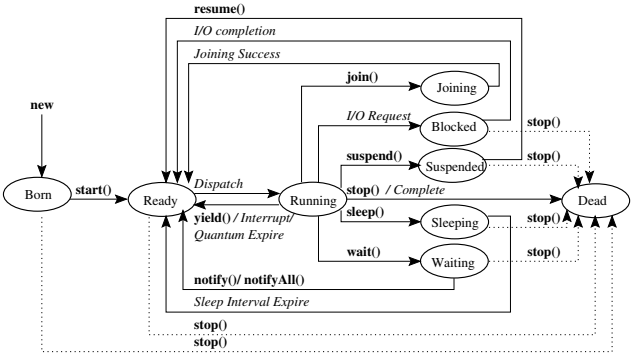

Figure 1: Java thread's life cycle

Figure 1 shows a Java thread's life cycle that requires the following multithreading support [2]:

1. **Synchronisation:** Synchronisation prevents interference between threads. Resources are guarded by having locks on them. Only one thread is handed a key to access its protected resources while the others have to wait until it finishes. Each thread can hold many keys simultaniously.

* This work is supported by the EPSRC (GR/N64427/01)

2. **Interthread Communication:** Threads communicate by sharing data via the same memory space using the synchronisation primitive.
3. **Scheduling:** Scheduling determines the execution order of multiple threads using their priorities. A thread can voluntarily pass its privilege to waiting threads by using *yield()*, or via the time-slicing mechanism.
4. **Daemon Thread:** A daemon thread is an endless loop waiting to provide important services to the other threads (e.g. a low-priority garbage collector, a high-priority timer thread that wakes up in regular interval).

3. Multithreaded Architecture

To operate multiple threads concurrently, most architectures employ software support by translating each Java-thread operation with a sequence of machine instructions. Unfortunately, these sequences consume a number of cycles that is often exacerbated by cache misses. Hence, an alternative system that can process multiple threads at the hardware level using a minimised set of instructions has been investigated. Our Java-Multithreading Architecture (JMA) requires a smaller number of instructions per Java-thread method and handles multithreading at the hardware level.

3.1. System Overview

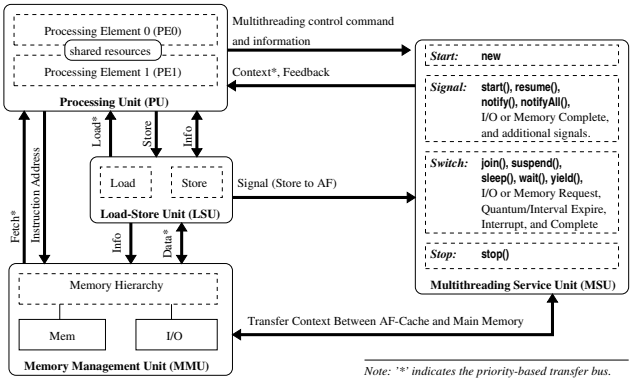

Figure 2: The multithreaded architecture for JMA

From the user level, a Java program is compiled into native code by additional compiler tools that are capable of preserving Java-thread controls in the form of native plus special multithreading instructions. These binaries will then be executed by our multithreaded processor. A *coarse-grained* context switching [3] is used so each short sequential thread segments is effectively executed. It combines *sequential behaviour* from the control-flow architectures [4] and *concurrent execution* from the data-flow architectures [5]. A directed graph represent a program where *nodes* represent thread segments, *arcs* represent their communication and control events. A matching-store mechanism [6] indicates that a thread is runnalbe when all inputs are present.

The system has four main components connected to one another as illustrated in Figure 2. The Processing Unit (PU) containing two pipelines. The Multithreading Service Unit provides four operations (*start*, *signal*, *switch* and *stop*) for synchronisation and scheduling mechanisms. Load and store operations are handled by the Load-Store Unit. the Memory Management Unit handles transactions for data, instructions and I/O.

3.2. The Processing Unit

As presented in Figure 3, two five-stage RISC pipelines [7] share four contexts, each of which is preloaded with the highest-priority runnable thread from a ready queue. The fetch stage was associated with eight-blocks of small L0 instruction cache for pre-fetching up to four multiple threads. Each thread is tagged with its own *colour* (Thread ID) while executed in the pipeline. The decode stage switches the context when receiving a context-switch indication or detecting that it requires the data that is still in a loading process. By providing pre-loading, pre-fetching, and colour tagging features, each pipeline has a capability to switch to another thread with zero overhead.

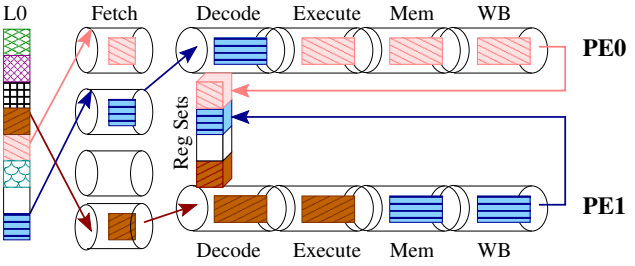

Figure 3: The PU for 0 context-switching overhead

3.3. Context and Activation Frame

The context of execution is a *program counter* + 31 registers (excluding register $zero whose data is always 0). This context is preserved in the form of an *Activation Frame* (AF) [8] when stored in memory. Each AF can be efficiently cached in a special fully-associative AF-Cache located near the PU. The place for register $zero is replaced by two fields. The first 8 bits are the *priority* field for scheduling purposes. The second 24 bits are the *presence-flags* field indicating the presence/absence of 24 input parameters for handling complex data dependencies that may arise when parallelizing loops. A thread which has an incomplete AF has to wait for data events to arrive. This allows synchronisation of both data communication and thread control events via the available load/store commands. When flags are present, each AF will be ready to be dispatched to the PU depending on its priority.

3.4. Additional Multithreading Instructions

The architecture can support most multithreading control using load/store instructions (e.g. to access the AF to obtain a thread's data or signalling by setting a thread's presence flags). Four additional instructions are added for faster multithreading based on MIPS R3000 ISA as follows:

1. **start** $reg, address$

0x18	reg	$address$
6 bits	5 bits	21 bits

2. **wait** reg

0x19	reg		0
6 bits	5 bits	20 bits	1

3. **switch**

0x19		1
6 bits	25 bits	1

4. **stop** reg

0x1A	reg	
6 bits	5 bits	21 bits

Using these four additional instructions, Java's thread life cycle (Figure 1) is reduced to a simpler diagram as shown in Figure 4, where both *wait* and *store* instructions are associated with suitable arguments.

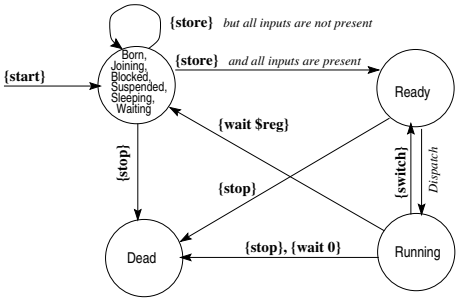

Figure 4: The real life cycle of thread in JMA

4. Results

The proposed system has been simulated. The simulator is used for simulating both the normal MIPS architecture and the JMA. The memory hierarchy used for this simulator has a main memory access latency of 200 clock cycles. IL1 is the first 4K direct-mapped instruction-level cache with a 5-cycle access latency. D-cache is a 4K 2-way set-associative data cache with a 5-cycle access latency. Both caches use an LRU replacement policy with write back.

Two version of Livermore Loop 7 (LL7) with n=1000 iterations is used as a benchmark. One is single-threaded code and the other is multithreaded code. The multithreaded version can fork 5 threads, each of which is assigned to handle 200 iterations. The experimental results is displayed in Figure 5. The workload is the number of LL7 programs operating in the system.

The results indicate that with a multiple Single-threaded workload, the JMA tolerates long latencies by switching in another workload, thereby offering a better execution time.

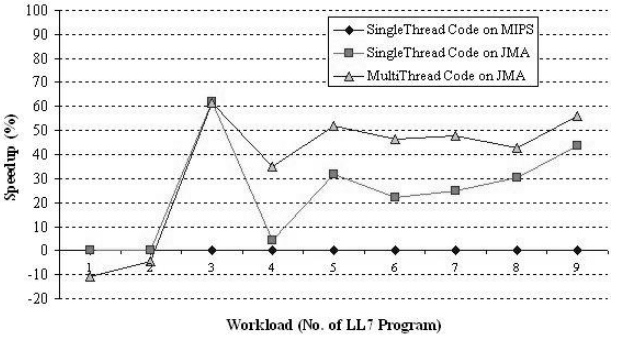

Figure 5: Speedup results compared with a single-thread performance operating on a normal MIPS

In particular, efficiency improves when executing multithreaded workloads.

5. Conclusions

The architecture of JMA provides high-performance multithreading based on a simple design without speculation or branch prediction. It hides long latency transactions like cache misses and I/O accesses using a low overhead (often zero cycle) context switch to another thread to avoid stalling the pipeline. The pipelines are similar to simple RISC with slightly more complex register files, an activation frame cache (which is similar in complexity to the instruction and data caches) and a real-time scheduler. Efficient scheduling and synchronisation mechanisms result in good resource utilisation with real-time characteristics which are ideal for many embedded applications.

References

[1] D. Patterson. New direction in computer architecture. In *PARCON: Symposium on New Directions in Parallel and Concurrent Computing*, November 1998.

[2] Laurence Vanhelsuwé and et al. *Mastering Java*. BPB Publications, 1996.

[3] G. T. Byrd and M. A. Holliday. Multithreaded processor architectures. In *IEEE Spectrum*.

[4] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approch*. Morgan Kaufmann, 1996.

[5] J. A. Sharp. *Data flow Computing: Theory and Practice*. Ablex Publishing Corporation, 1992.

[6] R. A. Iannucci. *Multithreaded Computer Architecture: A summery of the state of the art*. Kluwer Academic Publishers, January 1994.

[7] D. A. Patterson and J. Hennessy. *Computer Organization and Design*. Morgan Kaufmann Publisher, 1996.

[8] Simon W. Moore. *Multithreaded Processor Design*. Kluwer Academic Publishers, June 1996.