Virtualization and legacy support, parallelization and speculation

Dr. Ian Rogers*, Prof. Ian Watson*

* *School of Computer Science, The University of Manchester, UK.*

ABSTRACT

In this extended abstract we describe the main themes of our research, namely, where computer architecture is leading us in terms of highly parallel general purpose processors. We discuss what the model for programs is becoming and the challenges faced. The main challenge is how these architectures bring about their potential performance whilst being programmable, transparent and a seamless migration for the user.

1 Introduction

This abstract is laid out into the following sections:

- **Future architectures** - in this section we gather what are important future hardware architecture trends.
- **A new model for programs** - in this section we give a picture on how future software will exploit this hardware.
- **Supporting the present** - in this section we describe how the world shifts from the status quo to the new architectures.

The abstract ends with some conclusions on what has been presented.

2 Future architectures

2.1 Chip multiprocessor

The last generation of processors had multiple functional units capable of executing many instructions in parallel. This led to performance improvements beyond what was possible by technology improvements. To keep scaling performance the current generation of processors have introduced having multiple CPU cores.

Currently the amount of parallelism exposed in a general purpose multi-core processors is much less than that provided by an array of more task specific cores, such as ClearSpeed's CSX 600 [cle06]. These cores are simpler than a general purpose core and have local memory. They will often lack the instruction level parallelism, floating-point units and the memory hierarchy of a general purpose processor. On paper their performance, however, is superior to general purpose designs.

An interesting compromise in between these two designs is the Cell processor, where some of the design is general purpose, but part of the design encompasses more task specific components each having local memories [Pham05].

[1]E-mail: {ian.rogers, ian.watson}@manchester.ac.uk

2.2 Core multithreaded

As the performance hit in going to memory increases, performance of a core can be improved by allowing it to speculatively execute the next instructions with the memory access still pending. An alternate approach is to execute more work on the same core but from a different runnable context. The Niagara processor allows for 4 contexts to be executed on a single core to hide memory latencies [Kong05].

2.3 Non-uniform memory accesses

For performance reasons, some commodity processors are integrating memory controllers on chip. Such processors, in a multiprocessor set up, inherently have non-uniform memory access times. That is, to get to memory from a chip you either have to go directly to your memory controller or via the memory controller of another chip. Local memories are inherently faster.

2.4 Reducing synchronisation

Thread synchronisation prevents a waiting context from being runnable. With databases the need for globally synchronising locks is avoided by the introduction of transactions. The same can be true in programming languages. To support transactions in hardware it becomes important to have commit and rollback mechanisms local to a core [Carl05]. This can be done using buffers or by modifying the cache to act as a buffer. Although transactions remove the need for synchronisation, a check must be performed to check transactions don't interfere. Any checks are likely to steal time, threads and/or cache lines from the parallel work. Synchronisation free parallelism, exposed by the programmer and compiler, will have higher performance than transactional schemes. Where many synchronisations are necessary, transactions provide a useful extension, just as load-lock/store-conditional did for multiprocessor RISC systems, wishing to split previously atomic bus operations.

2.5 Object-oriented memory design

Scaling the amount of multi-context parallelism leads to problems in the memory system design. Bus snooping scales badly with current technologies and so directories with point-to-point links are preferred for performance reasons. The directories maintain information on what addresses are in use, and by which cores with granularities down to that of a cache line. A problem of false sharing can occur when two cores are accessing different addresses that share the same cache line.

An object-oriented memory views memory in terms of object identifiers and offsets [Wrig05]. As memory accesses are primarily to caches, the translation of object identifier to memory location need only be performed when filling cache lines, and a TLB can cache the translation in the same way it caches mappings of virtual to real memory addresses. Modifying the view of memory to objects may simplify the job of any directories. As objects can have well defined locking primitives, the efficency of the memory system design can be improved.

A side-effect of object-oriented memories is that handles to objects are reintroduced. This can lead to improved garbage collection designs.

2.6 Summary

From these research areas we see that future general purpose architectures are going to have two distinguishing features:

- **many threads/contexts**: multiple cores and contexts per core.
- **radically different memory**: placing the main working memory of a context near the core, distributing the memory design, giving it a transaction supporting object-oriented view.

3 A new model for programs

Languages such as C weren't designed to be multithreaded. Unless homogeneity of cores is to be a principal of future architectures, the instruction set architecture needs to be abstracted too. We believe that virtual machines, such as the Java Virtual Machine, are the only way to design software for future architectures. Conventional virtual machines aren't self-hosting[2], in contrast we focus our research on the Jikes RVM [jik06].

3.1 Locally distributed virtual machines

Distributed virtual machines, such as dJVM (built upon the Jikes RVM), allow separate memories to be presented as a single object space [Zigm02]. Their utility is primarily in clusters, but by considering their use on chip with a distributed on chip memory, the cost of scheduling work and maintaining the memory model is greatly reduced.

3.2 The virtual machine becomes the OS

Having threads with 100s of bytes of contexts makes distribution expensive and prevents work distribution from being agile. Having scheduling controlled by a separate context is another performance bottle neck. Combining the OS and virtual machine leads to simplified memory system design and exposes performance improvements. For example, the device driver code to perform IO can be inlined across the previously present library and operating system barriers. In our research we have looked to extend the JNode operating system to be integrated with the Jikes RVM [Roge05].

3.3 Parallelization

Exposing parallelism is fundamentally the job of the programmer and the programming language. Writing threaded applications and removing synchronisation aid this. Having high-level mathematical languages do this is also important. To expose more, possibly fine grain, parallelism the virtual machine's compiler is required.

3.3.1 Supercompiler parallelization

Supercompiler parallelization parallelises loops in situations where dependency analysis is provable. We have written optimisations that dynamically parallelise Java loops at runtime, with predicted performance improvements when used in conjunction with lightweight threads [Zhao05].

3.3.2 Speculative parallelization

When dependencies can't be proven the transaction mechanism can be used to speculatively execute work in parallel. A check needs to be performed, possibly in parallel, to determine if the speculation was correct before committing it to memory. When there are idle contexts, speculation provides an answer as to what can make use of this parallel resource and possibly gain performance.

4 Supporting the present

Having a great new system is unlikely to make software developers and users migrate to it. Whilst targeting the hardware and software architecture for performance, a means for legacy support is required.

4.1 Binary translation

PearColator, a binary translator integrated with the Jikes RVM, extends the Jikes RVM's functionality so that it becomes a virtual machine capable of running IA32 and PowerPC binaries [Roge05]. Such a binary translators lacks the rich information available to a virtual machine compiled to a virtual instruction set, such as Java bytecode, so it must work harder to expose new performance

[2] they are themselves statically compiled from a different programming language

(in terms of dependence analysis). The ability to inline and remove performance barriers isn't lost though.

4.1.1 Virtual memory

To support the memory system of the program's efficiently the binary translator needs an efficient mapping from the program's virtual addresses to those of the object memory. One approach is to break the virtual address into indexes of a page table and page, where the page-table is an array of array pages. The cost of the memory indirection can be removed in the same way as the handle indirection is removed in the object-oriented memory.

5 Conclusions

The software architecture is experiencing a time of change brought about by the new design complexities of new hardware. There are barriers to achieving performance on new hardware as well as barriers to its adoption. With the JAMAICA project we are tackling these problems on a number of fronts [jam06].

References

[Carl05] B. CARLSTROM, J. CHUNG, H. CHAFI, A. MCDONALD, C. CAO MINH, L. HAMMOND, C. KOZYRAKIS, AND K. OLUKOTUN. Transactional Execution of Java Programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. Oct 2005.

[cle06] CSX Architecture Whitepaper. `http://www.clearspeed.com/downloads/Architecture%20Whitepaper.pdf`, 2006.

[jam06] The Jamaica Project. `http://www.cs.manchester.ac.uk/apt/projects/jamaica`, June 2006.

[jik06] Jikes™Research Virtual Machine (RVM). `http://jikesrvm.sourceforge.net/`, June 2006.

[Kong05] P. KONGETIRA, K. AINGARAN, AND K. OLUKOTUN. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[Pham05] D. PHAM, S. ASANO, M. BOLLIGER, M. DAY, H. HOFSTEE, C. JOHNS, J. KAHLE, A. KAMEYAMA, J. KEATY, Y. MASUBUCHI, M. RILEY, D. SHIPPY, D. STASIAK, M. SUZUOKI, M. WANG, J. WARNOCK, S. WEITZEL, D. WENDEL, T. YAMAZAKI, AND K. YAZAWA. The design and implementation of a first-generation CELL processor. In *Solid-State Circuits Conference*. ISSCC, February 2005.

[Roge05] I. ROGERS AND C. KIRKHAM. JikesNODE and PearColator: A Jikes RVM Operating System and Legacy Code Execution Environment. In *2nd ECOOP Workshop on Programm Languages and Operating Systems (ECOOP-PLOS'05)*, July 2005.

[Wrig05] G. WRIGHT, M. SEIDL, AND M. WOLCZKO. An object aware memory architecture. Technical Report TR-2005-143, Sun Microsystems Laboratories, February 2005.

[Zhao05] J. ZHAO, I. ROGERS, C. KIRKHAM, AND I. WATSON. Loop Parallelisation for the Jikes RVM. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, Dalian, China, December 2005.

[Zigm02] J. ZIGMAN AND R. SANKARANARAYANA. dJVM - A distributed JVM on a Cluster. Technical Report TR-CS-02-04, The Australia National University, September 2002.