Asynchronous Multithreaded Processor Cores for System Level Integration

D. K. Arvind and R. Rangaswami
Division of Informatics, The University of Edinburgh,
Edinburgh EH9 3JZ, United Kingdom.
E-mail: dka,ror@dcs.ed.ac.uk

Abstract

A radical alternative based on an asynchronous design methodology is proposed for the design of future system level integration devices. An asynchronous multithreaded processor core is presented along with glimpses into some design issues for a multithreaded Java byte-code compiler. Such a platform expoits concurrency at different levels: between coarse-grained threads running on a multi-node processor, between medium-grained threads running on the thread processing units in each processor node, and between fine-grained threads running on the functional units in each thread processing unit. Such as platform implemented using an asynchronous design methodology is capable of delivering high performance at a competitive power consumption.

1 Introduction

System Level Integration (SLI) is defined as the *design, analysis and implementation of integrated software and hardware systems in silicon.* It is widely believed that an SLI device should contain at least a processor core with accompanying embedded software, on-chip memory, and interfaces to off-chip peripherals. It might also incorporate other features such as on-chip buses, programmable or hardwired signal processing blocks, analogue/mixed-signal blocks, programmable or dedicated hardware and possibly micro-electrical mechanical devices.

There is considerable debate regarding the nature of the methodology for the design of future SLI devices. The impetus for SLI comes from two main drivers - improvements in silicon technology which will result in a billion transistors on a single chip by the middle of the next decade, and the growth of new application areas such as mobile embedded systems.

The opportunity presented to the designer, thanks to the shrinkage in feature sizes, has to be exploited in the face of quite considerable technical challenges. For instance, the interconnect delay does not scale with feature size. Therefore, this delay becomes more significant in deep submicron regions as it decreases far less rapidly than the gate delay. Methodologies which reduce or work around interconnect delays, especially global ones, will win in the design of future SLI devices. Power dissipation is another important consideration, given the expected explosion in the demand for portable and wireless devices running on batteries. For these devices to work on battery power for extended periods, the power consumption must remain small, in the order of a few microwatts. The challenge lies in achieving these targets without greatly sacrificing the performance characteristics of the devices.

Future SLI chips will necessarily be heterogeneous, containing a mixture of: digital and analogue devices, sensors operating at radio frequency and noisy micro-mechanical actuators - all living cheek by jowl on the same silicon substrate. This presents a whole host of new problems at high clock frequencies such as electromagnetic interferences and crosstalk which has to be reduced by means of costly shielding.

It is envisaged that future SLI devices will be designed by composing pre-designed and preverified parts - the so-called IP blocks - which will come from different sources. This will result in an increase in the number of domains with different clock frequencies and circuit-level timing conventions.

1.1 An Asynchronous Design Methodology for SLI

We advocate a radical alternative to the design of future SLI devices by removing the *clock* - the agent of global control. The SLI device is viewed uniformly, from the transistors upward, as networks of computing structures, called a *micronet* [Arv94], [Arv95]. These computing structures operate concurrently and communicate asynchronously.

These structures could, in their purest form, be fully asynchronous, implemented using selftimed elements. Such an approach avoids the overheads and problems associated with distributing clock signals. Asynchronous circuits exhibit average-case performance and exploit datadependent delays. Low power consumption is a natural side-effect of an asynchronous approach. Asynchronous circuits are event-driven and consume power only when and where active.

Micronet-based designs distribute control locally to the functional modules. This reduces the overhead of interconnect delays and enables the exploitation of fine-grained instruction level concurrency. Micronets support a compositional approach to system design by separating the timing issues from behavioural ones. One can modify the implementation of a given behaviour in a module, without influencing the timing of the rest of the system.

An asynchronous approach can better deal with IPs from different sources which can be integrated by having an asynchronous wrapper around them, without having to worry about finding the optimal clock frequency for the sub-modules. In general, the increasing significance of interconnect delays, the reuse of IPs with different timing characteristics and the need to integrate heterogeneous elements lead to a complex on-chip timing organisation which can be best solved using an asynchronous approach.

We recognise that the embedded program is a key differentiator between SLI products. An important thrust in our SLI design methodology is the efficient execution of these programs, written in high-level languages, on asynchronous processor cores. We believe that this is best achieved by understanding the close interplay between the compiler and the processor. The design of micronet-based asynchronous superscalar [Arv99] and VLIW [Arv97] architectures have been influenced by this approach. These architectures exhibit fine-grained concurrency and the parallelising compiler strives to expose microthreads in the given application program.

In the rest of this paper, Section 2 considers some issues in design of an asynchronous multithreaded processor architecture (Figure 1) and its operation, and Section 3 gives some glimpses into the design of a multithreaded Java compiler.

2 The Asynchronous Multithreaded Architecture

The architecure consists of a collection of four identical *Thread Processing Units (TPU)* with a *Private Register Area*, a *Bus Interface*, *Instruction Cache* and a *First-level Scheduler*. The *First-level Scheduler* (FLS) distributes instructions to the TPUs. Initially the *Head-thread*, which is the oldest running thread, always runs on TPU 0. The *Head-thread* can move around during program execution; should a *Head-thread* terminate, then a successor is nominated to take over this role. The FLS keeps track of the latest *Head-thread*.

Each TPU consists of an *Instruction Buffer*, *Thread Issue Unit (TIU)*, the *Execution Unit* with temporary registers, *Inter-thread Communication Unit (ICU)* and the *External Communication Unit (ECU)*. The TIU decodes the instructions passed to it by the FLS and conveys the information to one of the three units. The *Execution Unit* is an asynchronous micronet-based processor consisting of a collection of functional units such as ALUs, multipliers and adders, which can exploit instruction level parallelism in the program. The ICU handles communication between the threads through the Private Register Area. The ECU handles data communication outwith the node, either to the main memory or with the other nodes in the processor core. In the latter case, only the *Head-thread* of the target node can be accessed as this is the only thread which can be guaranteed to be active.

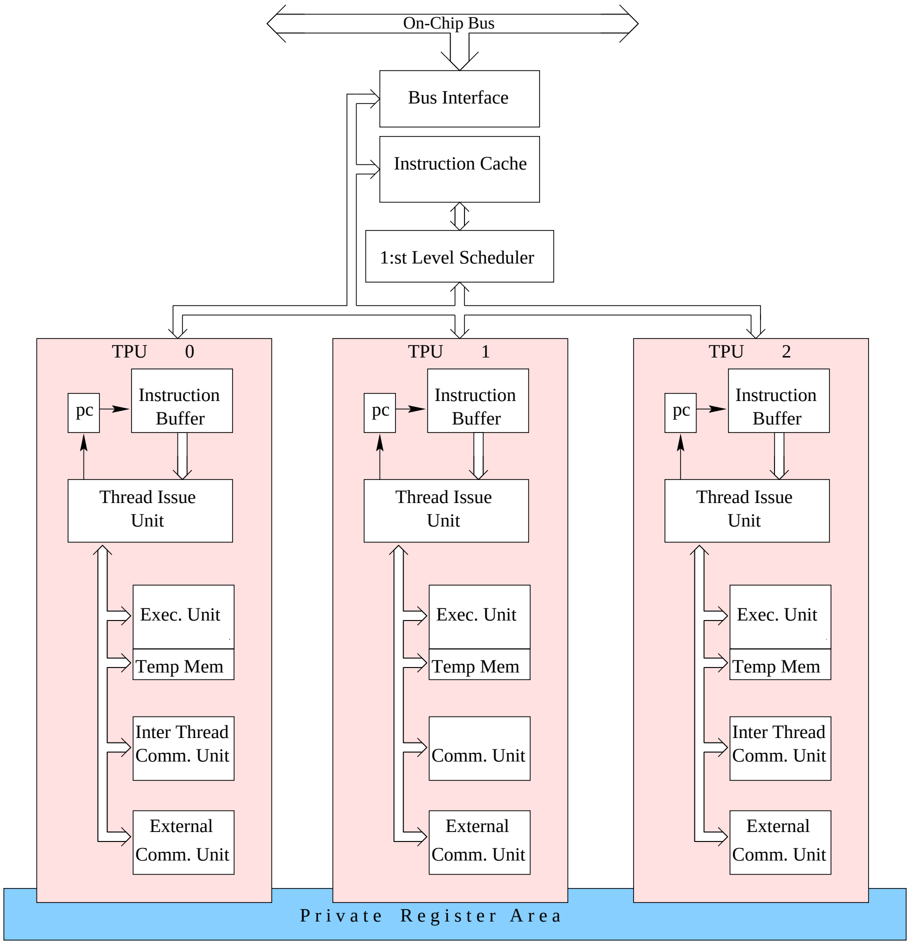

Figure 1: An asynchronous multithreaded processor node

2.1 Instructions for Multithreading

A new thread is started by the *Fork* instruction. It specifies the target TPU and the point at which the program counter should should start on the new thread. If the target TPU is on a different node, then the node address is specified and the *Head-thread* is the default target. A thread is terminated by the *Stop* instruction. It specifies a boolean guard value which can abort the *Stop*, and a register which contains the identity of the successor TPU if the thread is a *Head-thread*. The *Passive Signal* instruction serves as non-blocking interthread communication by transmitting a single word between two TPUs. It specifies the target TPU's address, and the value to be communicated which can be read by the target TPU at its convenience. No confirmation is sent that a signal has been received, and indeed signal can be lost if two different *Passive Signals* target the same TPU. The *Wait* instruction is used to receive the single word sent by the *Passive Signal*. It is a blocking receive and is the only method for receiving *Passive Signals*. The *Interrupt Signal* is used to interrupt an active TPU, and specifies the target TPU and the label indicating the value of the new Program Counter (PC). This interrupt is also used to terminate mispredicted threads in speculative executions on multithreaded architectures. The *Move Old Program Counter (MOPC)* instruction enables an interrupted program to be restored to its original state after the interrupt handler has been executed.

2.2 Operation of the Multithreaded Architecture

Dependent instructions in the program have to be implemented in the correct order. This can be achieved by either running the instructions sequentially on the same TPU, or on different TPUs by stalling the dependent thread until the first thread has finished execution, by using the pair of *Passive Signal* and *Wait* instructions. All the data dependencies are checked by the compiler.

At start-up the instructions are loaded into the *Instruction Cache*, some of which are transmitted by the *First level Scheduler* into the *Instruction Buffer* of one of the TPUs, and subsequently loaded into the *Thread Issue Unit*. The latter decodes the instruction and sends it either to one of the functional units in the *Execution Unit*, or to one of the *Inter Thread Communication Unit* or the *External Communication Unit*.

3 Compiling Java for the Multithreaded Architecture

The Java programming language uses a platform-independent program representation in a 32-bit stack-based run-time environment known as the *Java Virtual Machine* (JVM). A Java program is composed of interacting classes, each of which contains data members and methods. The strict format of the *classfile* ensures portability of Java programs between different implementations of the run-time system. The static compilation of Java classfiles attempts to provide the performance while retaining the benefits of the Java language. Other issues such as the Java security model are not considered here.

The threads are incorporated in the method compilation phase of the Java bytecode compiler called J2MAP. In the first stage the method's *basic blocks* are identified and used to construct the *Control Flow Graph*. In the next stage the loops that are to be threaded are identified. Both `for` and `while` loops in Java have a characteristic signature in the CFG. There is an initial *Prologue* block which links to a single successor - the *Test-and-Jump* block. This has two successors - the *Loop Body*, whose single successor is the the Test-and-Jump block, and the *Epilogue* whose entry at run-time signifies the loop exit. To identify these signatures it is sufficient to locate the single *back-edge* which links the *Test-and-Jump* block to the *Loop Body*. Each loop structure selected for threading is altered to include an additional *Fork Target* block. This block will be used to contain TPU initialisation instructions, and prepares the TPU for execution.

In the subsequent stage each bytecode instruction is replaced by equivalent MIPS instruction(s) in the internal method structure. Some bytecode instructions are used to manipulate the positions of variables on the stack; these are flagged and eliminated subsequently in the optimisation stages. For each loop which is threaded, code is inserted in the following areas: in the *Prologue* and *Target* blocks, those variables used for forking are initialised; *Passive Signal*, *Wait* and *Stop* instructions, with code to manipulate their respective arguments, are inserted at the beginning of the *Test-and-Jump* block; a *Fork* is inserted at the beginning of the *Loop Body* and a *Wait* is inserted in the *Epilogue* block; the *Passive Signal* and *Wait* are used to synchronise threads before they terminate; if a thread is not the *Head-thread* then it excutes a *Wait* prior to a *Stop*, to ensure that threads terminate in the correct order; a *Wait* is inserted in the *Epilogue* which is executed if the exiting thread is not the *Head-thread*. After register allocation and removal of redundant instructions, liveness analysis is necessary to identify variables whose values need to be transmitted to new threads as they are being created by the threading mechanism. The dependent instructions are identified which redefine the live variables in the *Loop Body*. The assembly code is output by assigning unique names for function calls by using the class name, the method name, and the arguments list. These are guaranteed to be unique due to the naming regulations of the Java programming language.

4 Conclusions

The motivation for a radical alternative to the design of future system level integration devices has been made. The micronet model treats the system uniformly at the different levels of abstraction as networks of computational units. An asynchronous multithreaded processor core which can potentially exploit concurrency at least three levels has been described. Some of the issues in the design of a multithreaded Java compiler have been highlighted. The resulting J2MAP compiler has been successfully used to run programs on an instruction cycle simulator of the multithreaded processor architecture. A mobile web browser consisting of four threads - the web browser, the communication protocol stack, a doppler-tolerant code acquisition in CDMA systems, and image rendering algorithms is currently being exercised on the multithreaded platform.

Acknowledgements

We wish to thank the following students: Alastair Patrick for the implementation of the first version of the J2MAP compiler and the simulation environment, Jae Hossell for implementing the multithreaded extensions to J2MAP and Christoffer Arvidsson for modelling and simulating the multithreaded architecture. The authors wish to acknowledge the SHEFC RDG project entitled "Design Clusters for System Level Integration" for equipment grants.

References

[Arv94] Arvind, D. K. and Rebello, V . E. F. 1994. "Instruction-level Parallelism in Asynchronous Processor Architectures" Proc. of the Third International Workshop of Algorithms and Parallel VLSI Architectures, Elsevier, Leuven, Belgium, August 1994.

[Arv95] Arvind, D. K., Mullins, R. D. and Rebello, V .E. F. 1995. "Micronets: A model for decentralising control in asynchronous processor architectures", In Proc. Int. Workshop on Asynchronous Design Methodologies, pp 190-99, IEEE Press, London, England, May 1995.

[Arv97] Arvind, D. K. and Sotelo-Salazar, S. 1997. "Scheduling instructions with uncertain latencies in asynchronous architectures", In Proc. EUROPAR'97, Springer-Verlag, Passau, Germany, Aug, 1997.

[Arv99] Arvind, D. K. and Mullins, R. D. 1999. "A fully asynchronous superscalar architecture", In Proc. PACT'99 - Workshop on Parallel Architectures and Compilation Techniques, IEEE Press, Newport Beach CA, USA, Oct. 1999.