Outline of RISC-based Core for Multiprocessor on Chip Architecture Supporting Moving Threads

Jani Paakkulainen, Jari-Matti Mäkelä, Ville Leppänen and Martti Forsell

Abstract: *Programming multicore systems is currently considered very difficult. One reason is that those are mostly constructed from the hardware point of view. Many of the processor core design solutions in contemporary constructions emphasize execution speed of a single thread. Since the memory access delay is the real bottleneck, such techniques often aim at maximizing cache hits by programmer guided locality of memory references and prefetching memory locations, etc.*

In this paper, we consider constructing processor core solutions that support easy-to-use programming approach based on the PRAM model. Specifically, we consider a processor core design of a multicore system, where the aim is to amortize the memory access delays by having multiple simultaneous executable software threads per each processor core. The core switches the executed extremely light-weight thread at each step, and thus the core can wait for pending memory requests to complete without any penalty (as long as it has non-blocked threads). Moreover, we consider the core to support moving threads paradigm instead of traditional moving data paradigm. We present an outline of such a processor core architecture, where we change the traditional pipelined execution model of RISC.

Key words: *Moving Threads, Multiprocessor on Chip Architecture, Multicore on Chip, Parallel Random Access Machine*

1 INTRODUCTION

Multicore based architectures have become the basic contemporary solution for processors, since the increasing of clock frequencies seems to approach its end (mainly due to heating problems). Although the processor manufacturing industry can construct various kinds of multicore architectures, their programmability has been seen as a major problem for several years now. Many authors and institutions have called for approaches supporting better programmability. Considering programming, the efficiency has often required the programmer to explicitly map data and parallel threads into the physical memories and cores of the underlying parallel computer. It is considered very difficult to maintain and express such dynamic mapping efficiently and correctly using programming language constructs and e.g. guiding the functionality of caches as in the Cell processor.

In our MOTH[1] (Moving threads realization study) project we have considered an approach for multicore on chip architectures, where the programming model is based on the classical Parallel Random Access Machine (PRAM) [6],[7] model. The PRAM approach abstracts away all kinds of mapping problems by providing a shared memory abstraction with unit (amortized) access cost. It also provides flexible expression of threads as well as strong synchronous execution of different kinds of threads. These properties together provide easy-to-program approach for programming multicore systems, but at the same time ask for efficient implementations of the PRAM approach.

SB-PRAM project [1],[9] provided one implementation of the PRAM in the multicomputer context. Recently, such PRAM implementations have been provided in the multicore on chip context by Forsell [3],[4] (Eclipse architecture) and by Vishkin et al [14] (Paraleap architecture).

We have invented a completely new kind of approach for mapping the computing of an application to MP-SOC architectures [5] (some preliminary ideas appear in [10],[11]). Instead of moving data read and write requests, we move extremely lightweight threads between the processor cores. Each processor core is coupled with memory module and parts of each memory module together form a virtual shared memory abstraction. Applications are written using a high-level language based on shared memory. As a consequence of moving threads instead of data we avoid all kinds of cache coherence

[1] This research has been funded by the Academy of Finland project numbers 128729 and 128733.

problems. Another advantage is flexible and efficient creation of new threads. In our architecture, the challenge of having efficient implementation of an application reduces to mapping the used data so that the need to move threads is balanced with respect to the bandwidth of the communication lines. Writing an application to use lots of threads is rather easy (due to rich literature of parallel algorithms using shared memory abstraction). This method also eliminates the need for separate reply network and introduces a natural way to exploit locality without sacrificing the synchronicity of the PRAM model.

Besides the PRAM approach, moving threads or thread migration has also been studied e.g. in [8],[2],[13].

Next in Section 2 we consider the moving threads approach in more detail. In Section 3, we provide our main contribution, which is an architecture outline for processor core supporting moving threads in the multicore on chip context. We also provide some preliminary cost evaluation of our architecture. Finally, conclusions are drawn in Section 4.

2 MOVING THREADS APPROACH

In the moving threads approach, a multicore system consists of P processor cores that are connected to each other with some sparse network [10], e.g. with a butterfly, a sparse mesh, a mesh of tree, etc. While the sparse networks have different properties concerning scalability, physical connection length and degree of nodes, they all share a property: a P-core sparse network can accept $\Theta(P)$ new messages per step and deliver $\Theta(P)$ messages to their targets per step. The delivery of a message involves a delay comparable with the diameter φ of the network, and φ can be non-modest. In traditional approaches, the messages correspond to read or write requests and replies, whereas in the moving threads approach, a message moves a thread consisting of a program counter, an id number, and a small set of registers. The messages in the moving threads approach are longer, but respectively there is no need for a network deliver the replies of read requests.

A cache-based access to the memory system is provided via each processor core. However, each core sees only a unique fraction of the overall memory space, and thus there are no cache coherence problems and when a thread makes a reference out of the scope of the core's memory area, the referencing thread must be moved to the core that can access that part of the main memory. Besides a cache to access the data memory, each core also has another cache for program instructions.

Each of the cores has $\Theta(X)$ threads to execute, and the threads are independent of each other — i.e. the core can take any of them and advance its execution. By taking an instruction cyclically from each thread, the core can wait for memory access taking a long time (and even tolerate the delays caused by moving the threads). The key to hide the memory (as well as network and other) delays is that the average number of threads X per core must be higher than the expected delay of executing a single instruction from any thread.

In this paper we assume that the program's address space is statically distributed into the memories accessible via cache modules attached to each core. The advantage of this is that the programmer can have influence on the physical allocation of data — and consequently on the physical allocation of the work of each thread on the processor-storage modules. Careful design of the allocation of actual data used in the program, thus allows the programmer to balance the work-loads and to minimize the movement of data.

For the creation and termination of threads in the programming language level, we take the approach of supporting only implicit termination as well as creation of threads. We do not consider Java-like explicit declaration of threads as first-class objects as a feasible solution. In practice, we have a parallel loop-like construction which creates threads with logical id-numbers in the interval [low, high] and each threads is running the same program block. The code in the program block can of course depend on the logical processor id-

number. The id-numbers are program controlled, but the runtime system expects them to be unique at anytime during the program execution. We also consider supporting nested thread creations. Each thread faces an implicit termination at the end of the program block (which was defined in the thread creation statement).

3 ARCHITECTURE FOR A CORE OF RISC BASED-PROCESSOR

3.1 General pipeline structure

A thread based processor core is modified from the well-known basic RISC architecture [12]. The conventional textbook model of RISC architecture consists of five pipeline stages: *fetch, decode, execute, memory access* and *write back*. A typical traditional single thread core contains sequencer (program counter), arithmetic logic unit and memory units (typically register file, data and instruction cache memories). The thread processor core architecture has thread table and selector unit, register file, arithmetic logic unit and two cache memories respectively. The major differencies compared to the basic architecture are the thread table and selector structure, large register file and the way instructions are fetched from the instruction cache memory. Another remarkable difference concerns a data memory access and an address space handling. Number of the pipeline stages (see Figure 1) is corresponding to the classical textbook model, but functional model of the pipeline is reorganised and extended from the original setup.

The 32-bit integer part of the DLX-type instruction set is considered to be a starting point of the thread processor design. Only the integer part of the pipeline is modelled in an initial architecture scheme. Although a pipeline for the integer computations is more regular and easier to design than such for floating point operations, it is possible to enhance the processor architecture to handle floating point calculations with needed extension of the hardware later. A couple of special instructions is included in the instruction set to fulfil the PRAM approach requirements.

Instead of one program counter the thread processor core has many, for example 256, parallel program counters. And for each thread there is a dedicated program counter with additional information fields. All program counters and information fields are ordered in a table form. Each entry contains present instruction address (program counter value), *prefetched* instruction itself (32-bit instruction), thread identity number (id) and thread state fields. In every cycle one executable entry is selected and fetched for further processing in the processor's pipeline.

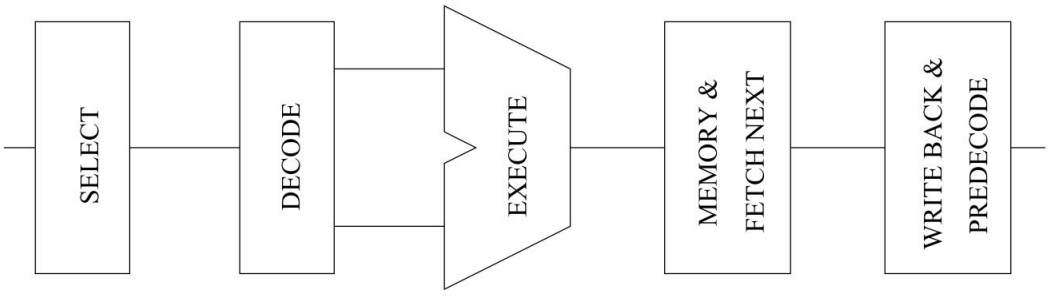

Figure 1: The pipeline stages of the processor core.

3.2 Execution stages

A thread pipeline is organised in a similar way as the classical five stage pipeline of the RISC processor. Each stage is separated with extended pipeline registers. These registers contain extra fields that are reserved to store thread's id number and program counter value. Both fields are carried through all pipeline stages.

In the first phase of the pipeline the selection control of the processor finds out the next instruction. The control points previously selected table entry and the next executable instruction is chosen based on the thread id numbers and the state of these threads. After

a suitable entry is located, command, id and program counter value of selected thread are forwarded to the decode stage.

A decode phase operates with the same instruction as a normal pipeline. The operation code (*opcode*) bits indicate command type under decoding and depending on the *opcode* the rest of the instruction bits are connected to a desired path. The important part of this phase is the thread id number, which is used to select correct set of registers from the register file. The number of the register groups is equal to the maximum number of threads in the processor core, meaning that every thread has its own dedicated set of registers. A program counter value is updated in this stage by adding four to the current value (memory addresses are aligned). In the case of a jump instruction a new value is concatenated from updated program counter value and instruction bits.

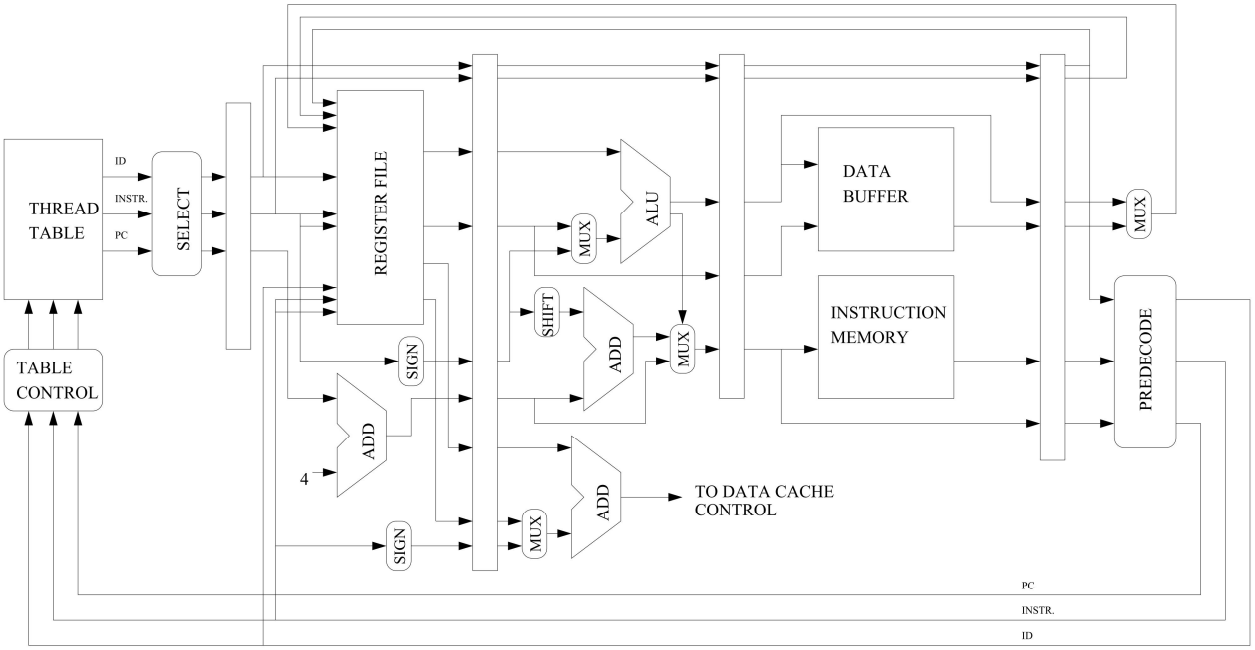

Figure 2: A Simplified data path model without control logic.

An arithmetic logic unit executes demanded operations in the next pipeline phase. The terms of branch instructions are tested in this stage and the correct address for the next instruction is solved. The result of ALU operations and the next program counter value are sent forward and the thread id is passed to the next pipeline registers.

A memory access stage is heavily extended from the conventional model. The data memory access operates with small preloaded data buffer, assumed that necessary data is available in this phase almost in every case due to *predecode* method. Also the instruction fetch is carried out in this step parallel to the data access. The feeding pipeline registers contain the final program counter value for the next instruction and a new command is fetched from that address. If an instruction reference request caused instruction cache miss, thread's status would be changed by processor control. A data request should not cause any data miss, but in this rare event, thread's status would be fixed to the corresponding value. Thread's id points the target entry, where fetched instruction is stored in the end of the next stage.

A pipeline is finalized by *write back and predecode* stage. The *write back* part saves the result of arithmetic and logic operations and data memory requests. The correct register bank and the target register is decoded using both thread's id and passed destination register value. A *predecode* part detects whether just fetch instruction is load or store type of command, and in the case of memory related *opcode*s the data address is evaluated with a separate adder structure. The adder calculates final data address and

verifies that the requested address is inside the processor's address space. When the address is between limits, the address request is transmitted to the data cache control, but if the address belonged to an address space of another processor core, a thread moving process would begin by the processor control. In the end of each cycle, the new program counter value and the instruction are stored in a table entry pointed by thread id. The thread's status is updated depending on the type of *predecoded* instruction.

3.3 Handling of threads

The indexed thread table is a key element of the core execution control. A table memory has **n** entries that indicate the maximum number of parallel threads in each processor core. The thread id is concatenated from the core id bits and a row index of the table. Each row has three fields; present program counter value (initially 32-bit), thread status (2-bit) and preloaded instruction (32-bit).

A status field has four possible values; *free, run, wait* and *sync*. When processor starts a new thread, the thread is placed in the next line with *free* status counted from the last selected entry. If thread's status is *run*, a selection unit can send out the thread to the pipeline. When the thread is fetched to an operation, thread's status is switched to *wait* state by the processor control. A status is *wait* until instruction is gone through all required pipeline stages. With jump, branch, arithmetic and logic commands, the thread status is updated back to *run* state after *write back and predecode* stage. With data access instructions, the status is *wait*, until the data cache control indicates hit. In the case of data address being outside of processor core's address space after *predecode* stage and address generation, the thread is moved from the core and thread's status value stays in *wait* mode until all values related to the thread are sent out to the new target processor and finally reserved entry is released. The thread is ended with a special instruction and after the final stage of command execution, the thread entry is released. If the fetch next stage failed with instruction fetch, thread's status would be kept *wait* until the instruction cache is updated. A *sync* status is reserved for future extension.

3.4 Memory Systems

A processor system has the main memory behind a network rooted at the data caches. A minimum access latency through the network is dozens of cycles and therefore each processor core has both the local instruction memory and data memories. The access delay of the local memories is estimated to be one or two cycles. Naturally, the local and global latencies depend on final implementation.

The local instruction memory is implemented by using typical one level cache memory structure. A direct mapping or two-way associative block replacement method fulfil operational requirements of the instruction cache. The executable threads are lightweight and multiple treads exploit the same instruction code sequence in parallel with each other. For this reason local cache memory is used efficiently and communication between the instruction cache memory and the main memory is expected to be lower than in the case of single thread core with the same code sequence.

The data memory model of the processor is organised in two levels. The nearest part of the pipeline consists of small buffer memory module, which has only one slot for each thread. In memory access stage, data is located by thread id and read from that position. When data write occurs, write-through method is used to update the data cache and the data buffer.

The lower part is set-associative data cache memory. A data address is generated and relayed to a cache memory in *predecode* phase. When the data address is a cache hit, the asked data is transferred immediately and thread's status is changed to *run* by the main control. Whereas, if the result of data access was cache miss, the thread would be pending as long as the required data block was on the way from the main memory. Long data latency for one entry associated with this situation will be masked from the individual thread, if the total executable thread count in processor core is high enough.

4 CONCLUSIONS

We have proposed a RISC-based architecture for a processor core supporting the moving threads approach. Among the contributions is a new kind of pipelined execution model for multithreaded RISC-based architecture. Our architecture models each executed thread as a register set, an execution state information, a program counter, and a core-related unique id number. As the idea is to support hundreds (if not more) threads per core, the thread data is arranged as id number indexed register sets that can be efficiently accessed with the execution pipeline.

REFERENCES

[1] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, C. Lichtenau. "Building the 4 processor SB-PRAM prototype". In *Proc. of the 30th Hawaii International Conference on System Sciences: Advanced Technology Track - Vol. 5*, 1997.

[2] V. Chaudhary and H. Jiang, Techniques for Migrating Computations on the Grid, In Engineering the Grid: Status and Perspective, Editors: Beniamino Di Martino, Jack Dongarra, Adolfy Hoisie, Hans Zima, and Laurence T. Yang, American Scientific Publishers, January 2006, 399-415.

[3] M. Forsell. "A Scalable High-Performance Computing Solution for Network on Chips." *IEEE Micro 22(5)* (September-October 2002), pp. 46-55.

[4] M. Forsell, V. Leppänen. "High-Bandwidth On-Chip Communication Architecture for General Purpose Computing." *Proceedings of 9th World Multi-Conference on Systems, Cybernetics and Informatics, WMSCI'2005*, pages 1--6, 2005.

[5] M. Forsell and V. Leppänen. "Moving Threads: A Non-Conventional Approach for Mapping Computation to MP-SOC." In *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'07)*, pages 232-238, Jun 2007.

[6] S. Fortune, J. Wyllie. "Parallelism in Random Access Machines". In *Proc. 10th ACM Symposium on Theory of Computing*, pp 114-118, 1978.

[7] J. J'aj'a. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[8] S. Jenks and J-L. Gaudiot, A Multithreaded Runtime System with Thread Migration for Distributed Memory Parallel Computing, In Proceedings of High Performance Computing Symposium, 2003, Advanced Simulation Technologies Conference, Orlando, FL, 2003,

[9] J. Keller, C. Kessler, and J. Träff. *Practical PRAM Programming*. Wiley, 2001.

[10] V. Leppänen: *Studies on the Realization of PRAM*, PhD thesis, University of Turku, Department of Computer Science, TUCS Dissertation 3, November, 1996.

[11] V. Leppänen. "Balanced PRAM Simulations via Moving Threads and Hashing." *Journal of Universal Computer Science*, 4:8, 675--689, 1998.

[12] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design; The Hardware/Software Interface,* third edition, Morgan Kaufmann, San Francisco, 2005

[13] K.A. Shaw and W.J. Dally, Migration in Single Chip Multiprocessors, Computer Architecture Letters, Vol. 1, No. 3, Nov. 2002, pp. 2-5.

[14] X. Wen, U. Vishkin. "FPGA-based prototype of a PRAM-On-Chip processor." *Computer Frontiers 2008*, May 5-7, 2008.

ABOUT THE AUTHORS

Researcher Jani Paakkulainen, MSc, Department of Information Technology, University of Turku, Finland, E-mail: jani.paakkulainen@utu.fi

Researcher Jari-Matti Mäkelä, BSc, Department of Information Technology, University of Turku, Finland, E-mail: jmjmak@utu.fi

Adjunct Professor Ville Leppänen, PhD, Department of Information Technology, University of Turku, Finland, E-mail: ville.leppanen@it.utu.fi

Chief Research Scientist Martti Forsell, PhD, Platform Architectures Team, VTT Oulu, Finland, E-mail: martti.forsell@vtt.fi