RISC-Based Moving Threads Multicore Architecture

Jari-Matti Mäkelä, Ville Leppänen and Martti Forsell

Abstract: *In this paper, we describe the architectural output of our 'Moving threads realization study' (MOTH) project, which is a RISC-based multicore architecture framework. Each fraction of the memory can be accesses only via a certain core, via its cache memory. This approach leads to moving light-weight threads but at the same time provides strong memory coherences as no main memory location is replicated to several caches. We describe the overall multicore architecture, but special emphasis is put on describing the functionality of individual RISC-based core.*

Key words: *Parallel computing, moving threads, processor architecture, RISC, multithreading.*

INTRODUCTION

Previously in [8] we outlined a RISC-based architecture for our moving threads approach that has been the focus of MOTH project[1]. We have also studied non-RISC based solutions [3]. The goal in these studies is to construct processor core solutions that support easy-to-use programming approach based on the PRAM model (Parallel Random Access Machine) [4]. We aim to follow the PRAM model more closely than was done in the ParaLeap project [10].

In this paper, we describe the overall multicore architecture, but special emphasis is put on describing the functionality of individual RISC cores. A simulator and a compiler [6] also exist for this experimental architecture, but those are not discussed in this paper.

RISC-BASED ARCHITECTURAL FRAMEWORK FOR MOVING THREADS

Multicore System

An overview of our architectural framework is shown in Fig. 1. The system consists of *c* RISC-based cores, an interconnection network between the cores, and a main memory system. Each core maintains a set of threads, can execute instructions from those, send and receive threads via the network, and has a cache memory for accessing a part of the main memory. Each core C_i "sees" a unique fraction of the main memory via its data cache – such memory locations are called local to C_i. Thus, if a thread residing at core C_i issues a memory instruction concerning some memory location local to core C_j, then the thread must be moved to C_j before executing the instruction. Moving a thread basically means moving the contents of its registers and program counter. The program, being executed by a thread to be moved, is not moved, since each core has an instruction cache, which contains fractions of all program codes being executed by the threads residing at that core.

Each memory location is local to only one core. Thus, there are no consistency problems, since there is no real replication of the contents of memory locations. Each memory location can be cached. The data caches of cores act as root access points into the main memory. In the framework, we do not specify how the main memory is organized – e.g. it can be partitioned into blocks. We neither do fix the organization of the memory hierarchy – there can be multiple levels of caches. The mapping of memory locations into cores is not fixed in our architectural framework. We expect such a mapping to be balanced, but leave it open whether the mapping is static or dynamically set by the executed programs.

---

[1] *This work was supported by the grants 128729 and 128733 of the Academy of Finland.*

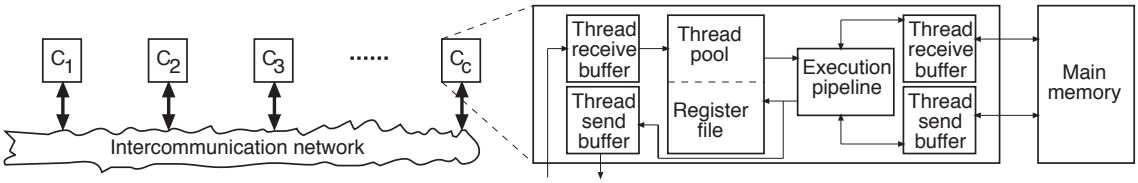

Fig. 1 Overview of our multicore system

We explain the basic function of a core next. Each core maintains a dynamically varying set of threads by storing their register values in a register file and maintaining other information regarding them in a thread pool. A core extracts instructions from the threads (by using their program counter value) in its thread pool and injects such instruction into its instruction execution pipeline. None of instruction in the pipeline is a non-local memory instruction. The nature of next instruction is determined at the end of execution pipeline – thus, the need to move a thread is determined as early as possible.

The network connecting the cores is for moving threads between the cores. Thus, each core has separate thread buffers for sending and receiving threads. The received threads are moved into the thread pool of the receiving core, and respectively sending means removing a thread from the pool of the sending core.

In the framework, we do not specify any exact topology for the interconnect. However, we assume the network between the send and the receive buffers to consist of a number of intermediate nodes whose connections form a DAG (from the send to the receive buffers). We assume the throughput of the network to be such that each node can send and receive a thread approximately every δ cycles. We denote by T_L the average latency (in cycles) of moving a thread in the network from one core to another. Moreover, we assume that δ is some small constant, independent of c. There exists various such sparse networks, e.g. the butterfly, mesh of trees, and various sparse meshes.

The execution of all threads in the whole system is synchronous. The strict interpretation of PRAM execution is that all threads execute synchronously stepwise – meaning that there is implicit synchronization after each step (i.e. atomic instruction). A relaxed interpretation is that there is a separate synchronization instruction in the instruction set, and encountering such an instruction in the execution is treated as a barrier synchronization point (all threads pass over a barrier when all the threads have reached it). The instructions of a thread between two barriers can be called as a superstep (notice that the length of superstep does not need to be static). The approach to the nature of execution synchrony is very crucial considering the semantics of programs and ease of programming. It is obvious that the more strict synchrony the easier to program but the more costly to implement. In our architectural framework, we do not specify how often the threads are synchronized, but we fix the architectural method of keeping the threads in synchrony. Our method is the synchronization wave method which can be seen to have been outlined already in the Fluent machine [9]. The idea of synchronization wave is that a wave front separates two consecutive (super)steps. The wave front moves over an element (whether an interconnection node or an element related to the execution pipeline of a core) once it has arrived into the element via all input "links". Moving over a node means that the wave front is forwarded to all possible output "links" of the node.

Architectural framework for a single core

The bedrock of our thread processor model is a pipelined RISC architecture. The major change to the basic 5-stage textbook model is the adoption of the moving threads; the address space is distributed among all cores, and a thread move occurs when the next instruction in the control flow refers to a non-local memory address or performs a special thread control instruction. This makes it necessary for an efficient implementation to pre-calculate the address of a reference before execution. Another difference is the

reorganized pipeline feeding technique, which achieves fine-grained thread level parallelism by alternating the executable thread between pipeline stages instead of executing instructions in coarse-grained blocks from a same thread at a time.

The core operational flow comprises six pipeline stages: select, decode, execute & fetch next, writeback & predecode, address calculation, and data memory access (Fig. 2, Tab. 1). The seventh stage, data buffering, is an independent background task running concurrently with the main pipeline. Each pipeline stage has been balanced to execute in one cycle, which results in instruction completion time of at least eight cycles. In case of instruction / data cache miss, a NOP placeholder operation is executed and in case of data cache miss, the thread's status remains unavailable until the data is available.

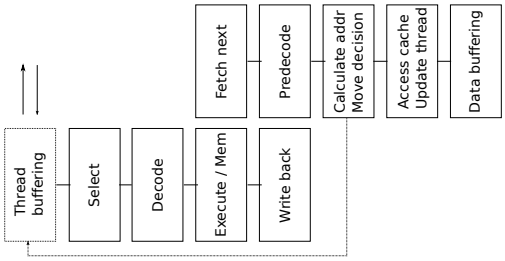

Fig. 2 The pipeline stages of the processor core

Instruction select	The first pipeline stage selects the next available thread for execution from the thread table. The table is organized as rows comprising three fields: status, program counter value, and a preloaded next instruction. While a more realistic implementation might require banking or a tree like selection logic, our model uses a simple linear array. The status field represents seven possible thread states (Fig. 3); free – empty slot where threads can be assigned, ready – ready to be selected for execution, exec – being processed in the pipeline, wait – execution is blocked by a pending data memory request, sync – waiting for the next sync point before executing, move & recv – the thread table is sending/receiving thread data from the I/O buffers. A search is performed every cycle to feed the pipeline with the program counter, instruction, and row id values of an available entry from the thread table (Fig. 4) and a placeholder NOP is executed if the search fails.
Instruction decode	The second stage decomposes the prefetched instruction, extracts a possible immediate value, concurrently issues register file fetches, and calculates the new program counter value from this data. Basic RISC instructions require two dedicated register file read ports for decoding, one for predecoding, and a variable amount of extra ports for thread moves. Wider data type support (e.g. SIMD, MADD/MSUB instructions) increases the requirements. The instruction data, program counter, thread row id, register and immediate values are passed to the next pipeline stage.
Execute and fetch next instruction	The third stage executes the decoded instruction and performs two fetches, one from the instruction cache and other from the data buffer. The data buffer is guaranteed to hold the correct data in the slot indexed by the thread row id, but an instruction fetch miss is hidden with a placeholder NOP with a program counter value decremented by four to repeat the failed fetch. The fetched data & instruction values along with the result of the ALU operation from the immediate and register values are fed to the next pipeline stage.
Write back and predecode	The fourth stage ends the instruction execution by either writing the result to the register file or to a memory location. A full write operation might take place asynchronously, but the synchronous part is assumed to finish during the stage. Basic instructions require one dedicated register file write port or if hardware MULT/DIV are supported, two ports. The predecoding concurrently initiates the execution of the next instruction by decoding the instruction. If the next instruction is a load or store using a reference calculated from a register value, that register value is fetched via a dedicated read port. The instruction data along with the address value components are fed to the next pipeline stage.
Calculate address and access data memory	The last two pipeline stages check with a hardcoded hash function whether the next instruction is accessing the local memory or if the thread needs to move to another core, and update thread state accordingly to move or sync. A local reference is also propagated to the data cache queue in the latter stage. The select and table control unit initiates a thread move in the background and changes the thread status to free when

	the thread's register values have been completely copied to the transmission queue. The details of the thread move have been omitted from the Fig. 4.
Buffer data	The data memory requests performed in the last stage are added to the data cache's queue. After processing the request, the cache stores the result in one of its two pipeline registers along with the accompanying requesting thread row id. The data buffer unit reads these values via two input ports and updates the values in a single cycle. The data cache also signals the select and table control unit with the thread's row id associated with the data. The unit then updates the status of the row from wait to ready. The latency of this operation depends on which point of the memory hierarchy the data is fetched.
Thread management	A fork–join concurrency model is adopted by the hardware; new threads are created with a fork instruction before a parallel section and combined with a join instruction into a single control stream after the section. The end of the section acts as an implicit barrier synchronization. Nested parallel sections are also supported, but expect software support. The thread forks and joins use a distributed two phased instruction model to support inexpensive coordination of a high number of threads; the thread operations are initiated with a broadcast packet and executed locally by all cores. The first thread operation creates a local coordinator thread for generating and managing the lifetime of the generated child threads and the same coordinator thread is used to propagate back a single join operation to the original parent thread. The child thread register values are clones of the parent thread's registers modulo the thread id value and the special register values for storing the cores of the coordinator and parent threads. The parent thread is unavailable during the parallel block, but the last join message from the child threads wakes it up again for the following sequential control flow section.

Tab. 1 The flowchart of the threads states

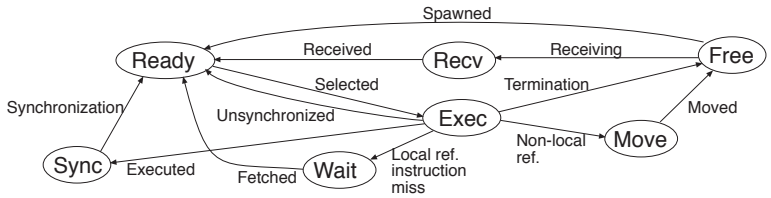

Fig. 3 The flowchart of the threads states

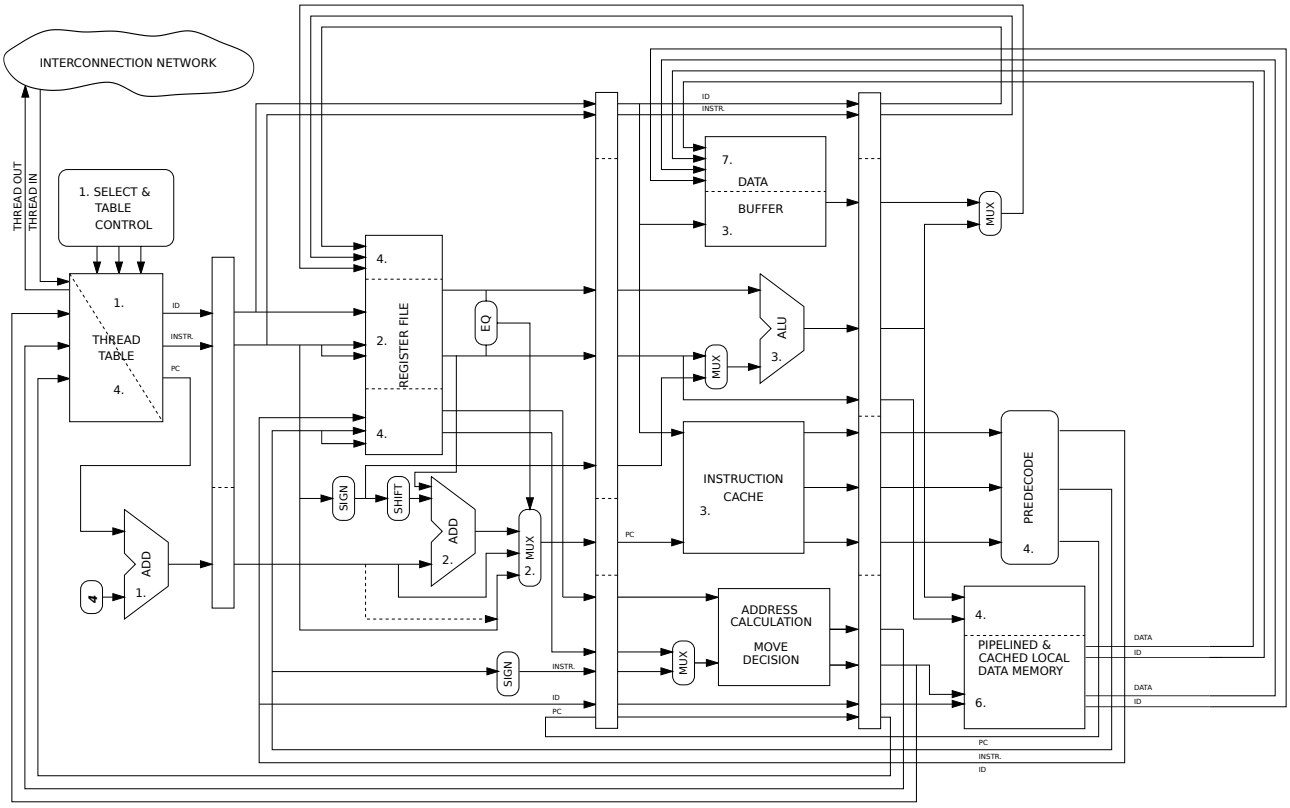

Fig. 4 The datapath model of the processor core

EVALUATION

We evaluated the performance of the proposed processor by simulating the execution of four simple benchmarks (see Tab. 2) in 3 moving threads configurations having 4, 16 and 64 processor cores, applying initial synchronization wave implementation (see Tab. 3), and ideal PRAM having the same configuration.

Name	Description
aprefix	Compute ordered multiprefix of a table of *T* integers
block	Copy an array of 32*T* integers distributed across the processor storage modules to another array in parallel.
max	Compute the maximum of a table of *T* words in parallel
spread	Spread an integer to the all *T* elements of a table

Tab. 2 Benchmark programs

Configuration	M4	M16	M64
Number of processors	4	16	64
Maximum number of threads	1024	4096	16384
Size of data memory (MB)	4	16	64

Tab. 3 Moving thread system configurations

The simulations were performed in our IPSMSim simulation environment modified for moving threads and mimicking the behavior of the RISC-based processor as closely as possible and adopting the network and buffering mechanisms form the Moving Threads Eclipse system [3]. The programs were written in e-language, and compiled with ecc optimization settings -O2. From each simulation we recorded the execution time of the benchmark algorithm itself, percentage of thread moves and the maximum length of FIFOs reflecting the packing of threads to a single processor core. Fig. 5 shows the e code for aprefix, its equivalent in MOTHC language [7], and the result of the compilation. The results of measurements are shown in Fig. 6.

```
#include "e.h"
#define     size 32768
int source_[size];
int main()
{
   int i;
   source_[_thread_id] = _thread_id;

   // Logartihmic algorithm for block sum
   for_ ( i=1 , i<_number_of_threads , i<<=1 ,
      if (_thread_id-i>=0)
         source_[_thread_id]
            += source_[_thread_id-i]; );
   return 0;
}
```

```
#include "moth.h"
#define size 32768
int source[size];
int main(void) {
  int i;
  pardo( t, size,
    source[t] = t;
  )
  pardo( t, size,
    for ( i=1; i < size; i<<=1)
      if (t-i>=0) source[t] += source[t-i];
  )
}
```

```
li      v0,0x8000
fork    v0
forkl
lui     a1,0x0
move    v1,k1
addiu   a1,a1,140
sll     a0,v1,0x2
addu    a0,a0,a1
sw      v1,0(a0)
join
joinc
fork    v0
forkl
li      a2,15
li      v0,1
subu    a3,v1,v0
sll     t0,a3,0x2
addu    t0,t0,a1
bltz    a3,0x74
addiu   a2,a2,-1
lw      t0,0(t0)
lw      a3,0(a0)
addu    a3,t0,a3
sw      a3,0(a0)
bnez    a2,0x50
sll     v0,v0,0x1
join
joinc
jr      ra
move    v0,zero
```

Fig. 5 E code, MOTHC code and result of compilation for the aprefix benchmark

We can make the following observations from the results:

• The performance of moving threads systems scales well but not ideally with respect to the number of processor cores. This because the configurations feature a fixed number of threads decreasing the relative chances to hide the latency.

• The overall performance of this system is clearly lower that that of the MTPA-based system because there is no support for continuing execution of moved instruction without a restart, the chaining of functional units, concurrent memory access, nor multioperations.

• The execution overhead with respect to a PRAM system with ideal memory system was between less than 1% and 41%, and increased as the number of cores increased. This

happens because the average number of threads per processor core stays the same although the actual latency of the network increases as a function of the number of cores.

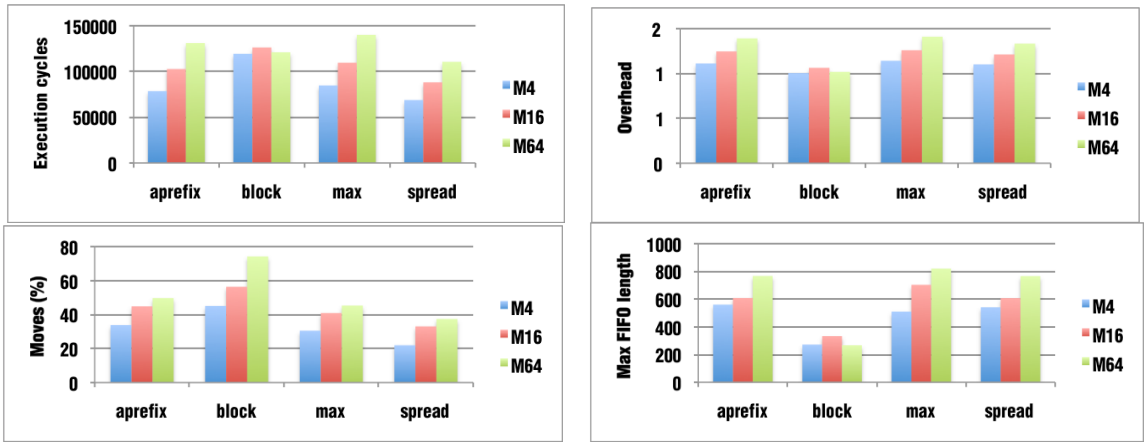

Fig. 6 The execution time, overhead with respect to ideal PRAM, percentage of moves and maximum length of FIFO buffers of the benchmarks

CONCLUSIONS

We have proposed a RISC-based architecture for the moving threads approach. Our contribution focus on explaining the new kind of pipelined execution model for multithreaded RISC-based architecture and outlining a fork—join based parallel programming model. According to our evaluation, the performance of the systems scales well as the number of cores increases.

REFERENCES

[1] M. Forsell. A Scalable High-Performance Computing Solution for Networks on Chip. IEEE Micro 22, 5 (September-October 2002), 46-55.

[2] M. Forsell and V. Leppänen. Moving Threads: A Non-Conventional Approach for Mapping Computation to MP-SOC. Proc. PDPTA 2007, 232-238.

[3] M. Forsell and V. Leppänen, A moving threads processor architecture MTPA, Journal of Supercomputing 57, 1 (2011), 5-19.

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

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

[6] J.M. Mäkelä and V. Leppänen. Towards programming on the moving threads architecture. Proc. CompSysTech 2010, 137-142.

[7] J.M. Mäkelä and V. Leppänen. MOTHC Compiler Manual, Version 1.0. TUCS Technical report. 978-952-12-2553-6. Feb. 2011.

[8] J. Paakkulainen, J.M. Mäkelä, V. Leppänen, and M. Forsell. Outline of RISC-based core for multiprocessor on chip architecture supporting moving threads. Proc. CompSysTech 2001, 1–6.

[9] A. Ranade. How to Emulate Shared Memory. Journal of Computer and System Sciences, 42, 3 (1991), 307–326.

[10] X. Wen, U. Vishkin. FPGA-based prototype of a PRAM-On-Chip processor. Proc. Computer Frontiers 2008.

ABOUT THE AUTHORS

PhD student Jari-Matti Mäkelä, Adjuct professor Ville Leppänen, Dept. of Information Technology, University of Turku, Finland. E-mail: jmjmak@utu.fi, Ville.Leppanen@utu.fi.
Adjuct professor Martti Forsell, VTT, Oulu, Finland. E-mail: Martti.Forsell@VTT.Fi.