Towards Programming on the Moving Threads Architecture

Jari-Matti Mäkelä, Ville Leppänen

Abstract

We have proposed a RISC-based multi-core architecture for the moving threads approach. A simulator is implemented for it, and in this paper we consider the toolchain for implementing parallel programs to be executed on the proposed architecture by our simulator. We give special emphasis for thread creation and management issues as well as for the synchronous execution model, and discuss programming constructs as well as compiling issues.

1 INTRODUCTION

In this paper we describe a compilation toolchain for our RISC-based multi-core architectural framework that is designed to implement a PRAM-based (Parallel Random Access Machine; [5]) approach for parallel programming. With this architecture, we aim to provide better programmability of parallel systems, since the basis of PRAM approach is a synchronous shared memory based execution of threads. The synchronous nature of execution essentially means that there are plenty of points in the program, where the programmer can rely that the previous memory write (and read) instructions have taken place. Consequently, the state of the program (concerning all threads) is clear and therefore designing a correctly functioning multithreaded program becomes easier. The PRAM has several variations regarding the choice of synchronization points. The most strict interpretation is that (implicit) synchronization takes place after executing a single step from all currently existing threads.

In the following, we give a short overview of our architecture for the moving threads approach in Section 2. In Section 3 we clarify the programming model, special emphasis is given for the synchronous nature of thread execution as well as for dynamic creation and termination of new threads. Section 4 describes our programming language constructs supporting dynamically changing number of synchronous threads. We describe the usage of compiling and simulation tools, and consider the implementation of thread management issues on our architecture. Finally in Section 5, we propose future work and draw conclusions.

2 ARCHITECTURE

A RISC architecture based on the MIPS32 instruction set is used as a base for the architecture. The classic 5-stage RISC pipeline has been redesigned and extended, both to host a large buffer of concurrent, time-interleaved threads inside a single processor core and to connect to a sparse inter-processor communication network.

The network uses advanced routing methods to scale beyond traditional multi-core to large many-core designs. It is responsible for two tasks: a) propagating a global synchronization signal using a technique known as the synchronization wave and b) distributing the load evenly by migrating threads and the spawning of new threads.

The memory layout is physically distributed; each core has a unique view of the address range via its local, cached memory module. Given a properly distributed algorithm, the solution allows utilizing a huge memory bandwidth. Since no memory is shared among the cores, it also avoids typical multi-core cache coherence issues. However, together with the moving threads im-

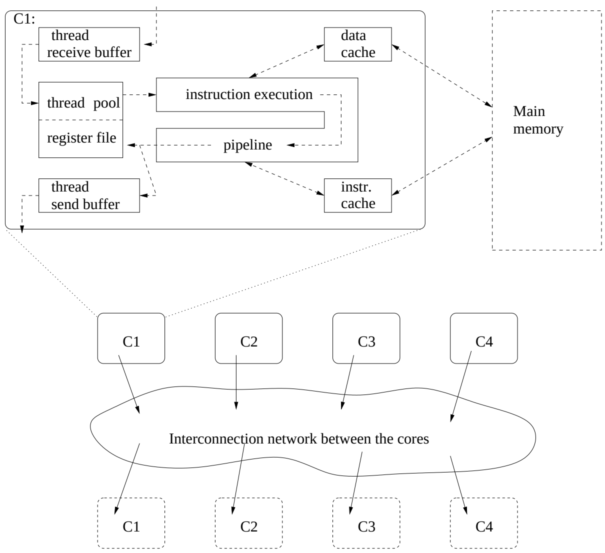

Figure 1: Overview of our multicore system.

plementation, the memory modules form a single shared memory abstraction for the executable programs. An overview of our architectural framework is shown in Fig. 1.

2.1 EXECUTION CORE

Our previous paper [10] on the moving threads architecture discussed the construction of the execution core in more detail. Since the design has further developed in subtle ways, a short overview of the current design is given.

The pipeline consists of five stages: select, decode, execution, combined memory access and writeback, and predecode. In addition, the core contains buffers for the state of enqueued threads, pre-fetched data and the packets from and to the communication network. A register file contains thread-local registers.

Unlike in a traditional RISC processor, the barrel processor style core executes instructions from different threads in different pipeline stages. Along with the various buffers, this design avoids pipeline stalls in most cases because a) the threads are guaranteed to execute without delays once they become active and b) the likelihood of always having active threads in the large thread table for the first pipeline stage to choose from is rather high.

2.2 SYNCHRONIZATION AND COMMUNICATION

The purpose of synchronization wave is to provide a rather cheap way to enable the execution of all threads to advance synchronously. Moreover, the idea is to enable each thread to advance in their computations only one logical step at a time, yet still avoid the usage of expensive barrier synchronization.

A *synchronization wave* is sent by the sources (cores) to the destinations (the same cores reached via the network between the cores). This technique has been successfully used e.g. in [1, 7, 9, 12]. In connection of the moving threads, the idea is that when a core has processed one logical step from each of the threads it is hosting and as a consequence of processing it might have sent some of the threads on their way to other cores, the core will send a synchronization packet (possibly implemented as an "empty" thread) to all of its outgoing network links.

Synchronization packets from various sources push on the actual packets (= moving threads), and spread to all possible paths that the actual packets could go. When a node receives a synchronization packet from one of its inputs, it waits, until it has received a synchronization packet from all of its inputs, then it forwards the synchronization wave to all of its outputs. While waiting, the node of course forwards other packets. The synchronization wave may not bypass any actual packets and vice versa. When a synchronization wave sweeps over a DAG based routing machinery, all routing machinery nodes and cores receive exactly one synchronization packet via each input link and send exactly one via each output link.

The network between the cores must be such that each core can send and receive one thread per every approx. c steps, where c is some small constant (the frequency of moving). Although the network has some average latency of ϕ steps to move a thread from one core to another, the network must still be able to meet the requirement to receive and deliver a thread per core every $\approx c$ steps. If the routing network has diameter (or average routing distance) ϕ, then a precondition of hiding diameter influenced latency is that the network with p inputs and outputs can move $\Omega(p\phi)$ packets (threads) in each step. If the sources can provide lot of packets, say h per source, that the network routes in a "pipelined" way, then it is possible to decrease the average routing time per packet to a constant.

Many kinds of architectural solutions satisfying the above have been proposed; see e.g. [4, 8, 11, 13, 14]. The internal structure of a scalable network should be such that simple routing nodes are connected to each other with constant length connections and have constant degree. Meshes and tori are such architectures. The requirement that the network must be able to move at least ϕ packets per source and target, or core, (and nodes have constant degree) means that at most $O(1/\phi)$'th of the nodes can be cores. Such an architecture is called *sparse*. We have proposed such architectures in [4, 8, 11]. The architecture of Eclipse [3] is a 3-dimensional sparse mesh that has been flattened on a 2-dimensional plane (its connections are only between physically neighboring components).

3 PROGRAMMING MODEL

3.1 MEMORY MODEL

As mentioned in Section 2, the memory modules are physically distributed, yet provide a conceptual shared memory model. The idea behind the shared memory abstraction is that when a thread attempts to access a memory location that is not locally available in the executing core, the thread is sent to the core that has access to the location. The algorithm for determining the correct core is currently hard-wired to the architecture and uses a simple modulo arithmetics or some other hash function for determining the core from the reference's address.

The model imposes no restrictions on the use of memory locations from different threads. The physical architecture prevents simultaneous memory accesses from occurring, but the order of memory accesses to the same location during a single step is left undetermined.

In addition, the per-thread registers can be seen as a form of local memory. A thread can only directly access its own registers and the register values are copied when child threads are created. Since the provided compiler does not yet support the notion of thread-local storage, the local state has to be explicitly carried in the registers.

Accesses to both memory types have a unit time amortized cost, although the real latency of the operation depends on the need to move the thread and cache misses. The locality of a memory reference has no effect on the program semantics per se, but having good data locality can increase the algorithm's execution performance, especially when the latencies cannot be hidden with the interleaved thread execution.

3.2 SYNCHRONIZATION

The moving threads architecture currently supports two types of synchronization. First, the tightly synchronized lock-step execution model of the underlying PRAM model provides an inherent mechanism of synchronization for the execution of algorithms. Conceptually, a global implicit barrier synchronization occurs after each time step. This information can be used to reason about any two threads based on their previous execution history since the last explicit synchronization (e.g. after a fork).

The second way is to manage the control flow using two categories of concurrency instructions that follow the widely known fork–join model. In this model the work distribution can be achieved by cloning the so-called master thread using the fork construct. A unique runtime thread id value is used to distinguish between threads. The lifetime of the created child thread is typically shorter than the master thread's and ends with an explicit join construct.

Higher level synchronization constructs can be built on these basic building blocks. A non-terminating synchronization construct is also planned, but not already implemented.

4 PROGRAMMING LANGUAGE

The concurrency support for our architecture is built as an extension to the C programming language, but the support is somewhat limited and currently consists of a runtime library which provides access to runtime variables, low level concurrency instructions (`fork`, `join`) and a high level `pardo`-loop. We follow the ideas of parallel frameworks such as OpenMP [2] and Fork [6].

4.1 RUNTIME VARIABLES

Certain parameters of the architecture can be accessed via the runtime system. For instance, the amount of execution cores on the system may have a large effect on the optimal number of concurrent threads on the system. The core count can be queried using the `int moth_core_count()` function. The function returns the implementation dependent number with a dedicated machine instruction. In the future, the core count may also be implicitly used by the higher level parallel programming constructs provided by the runtime system.

The fork command follows the Unix tradition in that it clones much of the parent thread's state. The thread id number provides a way to distinguish between cloned threads. The id can be read using the runtime function `moth_thread_id()`.

The thread id is actually held in a shared register value, which makes it possible for the user code to overwrite the value in case an extra register is needed by the algorithm.

4.2 LANGUAGE CONSTRUCTS AND COMPILATION

The `fork` and `join` constructs map directly to the architecture's machine instructions. The fork translates to a fork instruction and join into two subsequent join instructions. The `pardo` loop is a sequence of `fork`, `moth_core_count()`, the provided code block, and `join`.

We show the translation process through a short example of summing the elements of two two-dimensional matrices (N and O). The summing is encoded as a pair of nested parallel loops. The operation can be simply expressed as follows (where M, N, and O are m-by-n matrices):

$$\forall i \in 1 \dots m, j \in 1 \dots n : M_{i,j} = N_{i,j} + O_{i,j} \tag{1}$$

The algorithm does not take advantage of the knowledge of the physical memory layout to maximize the data locality even though that could be done by calculating the originating cores of each reference using the `moth_core_count()` function and distributing the child threads in a way that eliminates unnecessary moving of the child threads — in this case completely.

The C language implementation using the MOTH runtime library for the Equation 1 is shown in Table 1 alongside with its assembly translation. The macro expansion of the `pardo` loops is commented out below the actual code to help understanding the algorithm.

```c
#include "moth.h"
#define m 16 // dimensions
#define n 32

struct matrix { int _[m][n]; }; // matrix definition

int main(void) {
  struct matrix M,N,O; // inputs and the result

  pardo(i, m,
    pardo(j, n,
      M._[i][j] = N._[i][j] + O._[i][j];
    )
  )
}

/*
   fork(16);
     int i = moth_thread_id();
     fork(32);
       int j = moth_thread_id();
       M._[i][j] = N._[i][j] + O._[i][j];
     join();
   join();
*/
```

```asm
li v1,16
fork v1 # pardo #1
move v1,k1
li a0,32
fork a0 # pardo #2
sll v0,v1,0x5
addu v1,v0,v1
lui a0,0x0
lui v0,0x0
sll v1,v1,0x2
addiu a0,a0,136
addiu v0,v0,2184
addu a0,v1,a0
addu v0,v1,v0 # matrix B & C indices
lw a0,0(a0) # load B's element
lw v0,0(v0) # load C's element
lui a1,0x0
addiu a1,a1,4232
addu v1,v1,a1
addu v0,a0,v0 # matrix A index
sw v0,0(v1) # store A's element
join_move
join_dec # join #1 (implicit)
join_move
join_dec # join #2 (implicit)
```

Table 1: Program code of the matrix sum in C and MIPS assembly.

5 Conclusions and future work

The proposed programming model and the language implementation still have weak points. The simple fork–join model is good for solving simple parallel tasks, but its inflexibility is revealed when complex synchronization methods are needed. For example, a barrier synchronization without terminating the child threads is now impossible.

The largest shortcoming in the implementation of the compiler is that a simple macro based runtime library cannot capture the domain model as well as a full compiler with support for rewriting expressions and doing semantic analysis. In the future, the implementation of a moving threads target for some existing parallel compiler will be considered.

A preliminary toolchain featuring a cycle-accurate full architecture simulator has been built. At this point, the simulator was not yet ready to run reliable benchmarks, but we are planning to measure various micro-benchmarks such as the parallel merge sort and matrix multiplication.

The architecture does not yet have support for protected memory via MMUs or a stack construct. Also a runtime system for dynamic memory allocation is missing. The distributed physical memory model imposes additional non-trivial limitations on these so we have not considered the memory system on this paper.

References

[1] F. Abolhassan, J. Keller, and W.J. Paul. On the Cost-Effectiveness of PRAMs. In *Proceedings, 3rd IEEE Symposium on Parallel and Distributed Computing, ACM Special Interest Group on Computer Architecture, and IEEE Computer Society*, pages 2 – 9, 1991.

[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., SF, CA, USA, 2001.

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

[4] M. Forsell, V. Leppänen, and M. Penttonen. Efficient Two-Level Mesh based Simulation of PRAMs. In *Proceedings of International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN'96*, pages 29 – 35. IEEE Computer Society, 1996.

[5] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, 1978.

[6] C.W. Keßler and H. Seidl. The fork95 parallel programming language: Design, implementation, application. *Int. Journal of parallel programming*, 1997.

[7] V. Leppänen. On Implementing EREW Work-Optimally on Mesh of Trees. *Journal of Universal Computer Science*, 1(1):23 – 34, January 1995.

[8] V. Leppänen and M. Penttonen. Work-Optimal Simulation of PRAM Models on Meshes. *Nordic Journal on Computing*, 2(1):51 – 69, 1995.

[9] B.M. Maggs and R.K. Sitaraman. Simple Algorithms for Routing on Butterfly Networks with Bounded Queues. In *Proceedings of 24th Annual ACM Symposium on Theory of Computing*, pages 150 – 161, 1992.

[10] J. Paakkulainen, J-M Mäkelä, V. Leppänen, and M. Forsell. Outline of risc-based core for multiprocessor on chip architecture supporting moving threads. In *CompSysTech '09: Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, pages 1–6, New York, NY, USA, 2009. ACM.

[11] V. Leppänen R. Honkanen and M. Penttonen. Address-Free All-to-All Routing in Sparse Torus. In *Proceedings of 9th International Conference on Parallel Computing Technologies, PaCT-2007, LNCS 4671*, pages 200–205, 2007.

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

[13] J. Sibeyn. Solving Fundamental Problems on Sparse-Meshes. In *Scandinavian Workshop on Algorithm Theory, SWAT'98, LNCS 1432*, pages 288 – 299, 1998.

[14] L.G. Valiant. General Purpose Parallel Architectures. In *Algorithms and Complexity, Handbook of Theoretical Computer Science*, volume A, pages 943–971, 1990.

6 ABOUT THE AUTHOR

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