TUCS

Jari-Matti Mäkelä | Ville Leppänen

MOTHC manual, version 1.0

Turku Centre for Computer Science

TUCS Technical Report
No 999, February 2011

MOTHC manual, version 1.0

Jari-Matti Mäkelä
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
jmjmak@utu.fi
Ville Leppänen
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
ville.leppanen@it.utu.fi

TUCS Technical Report
No 999, February 2011

Abstract

This document describes a compiler and programming constructs for our experimental RISC-based moving threads architecture. We discuss of the usage of this compiler and show examples. However, this document is not just a user manual, but we also describe implementation related details as the RISC-based moving threads architecture is experimental in nature and more a framework than a product, and thus it as well as its simulator, compiler and related programming constructs are all subject to future developments.

As it requires huge resources to write an industrial-level compiler, we heavily use an existing compiler. The provided compiler, `MOTHC`, is an experimental light-weight "extension" of the GNU GCC compiler (MIPS32) by macro constructions and a runtime library for the architecture. Consequently, the programming for the moving threads architecture does not involve a new programming language, but rather the typical language-level constructions are provided for the programmer through macro definitions and library functions and procedures. As the base language, we use the plain C language. Naturally, the new language constructions are mainly related to threads, thread creation and termination using the special RISC instructions of the moving threads architecture.

Keywords: multi-core, moving threads, compiler

TUCS Laboratory
TUCS Algorithmics Laboratory

Contents

1 Introduction

This document describes a compiler and programming constructs for our experimental RISC-based moving threads architecture [11,13]. Observe that there exists also a non-RISC-based moving threads architecture [1–3]. We discuss of the usage of this compiler and show examples. However, this document is not just a user manual, but we also describe implementation related details as the RISC-based moving threads architecture is experimental in nature and more a framework than a product, and thus it as well as its simulator, compiler and related programming constructs are all subject to future developments.

As it requires huge resources to write an industrial-level compiler, we have decided to heavily use an existing compiler. The provided compiler, `MOTHC`, is an experimental light-weight "extension" of the GNU GCC compiler (MIPS32) by macro constructions and a runtime library for the architecture (`moth.h`). Consequently, the programming for the moving threads architecture does not involve a new programming language, but rather the typical language-level constructions are provided for the programmer through macro definitions and library functions and procedures. As the base language, we use the plain C language [6].

The moving threads architecture has a non-standard RISC instruction set, although most of the instructions are as in the ordinary RISC [14]. For this reason, we also discuss implementation of thread creation and termination using the special RISC instructions of the moving threads architecture.

Next, we give an overview of the moving threads approach and its architecture in Section 2. The implemented subset of the MIPS32 instruction set along with the moving threads instructions are presented in Section 3. The programming language and library extensions of the architecture is given in Section 4. The guides for installing and using our compiler are listed in Sections 5 and 6, respectively. Finally the Section 7 ends the manual with examples of applications for the moving threads architecture.

2 Overview of moving threads approach

Our RISC-based multicore architectural framework [11, 13] is designed for implementing a PRAM-based (Parallel Random Access Machine; [4]) approach for parallel programming. A goal is 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 pleanty of points in the program, where the programmer can relay 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. Our moving threads architecture follows this approach.

We assume the architecture to consist of homogeneous cores that are connected with an on-chip network, see Figure 1. Besides rather ordinary ALU capabilities, each core maintains a rather large set of hardware-supported threads, has separate memory and instruction caches, and has an on-chip network interface.

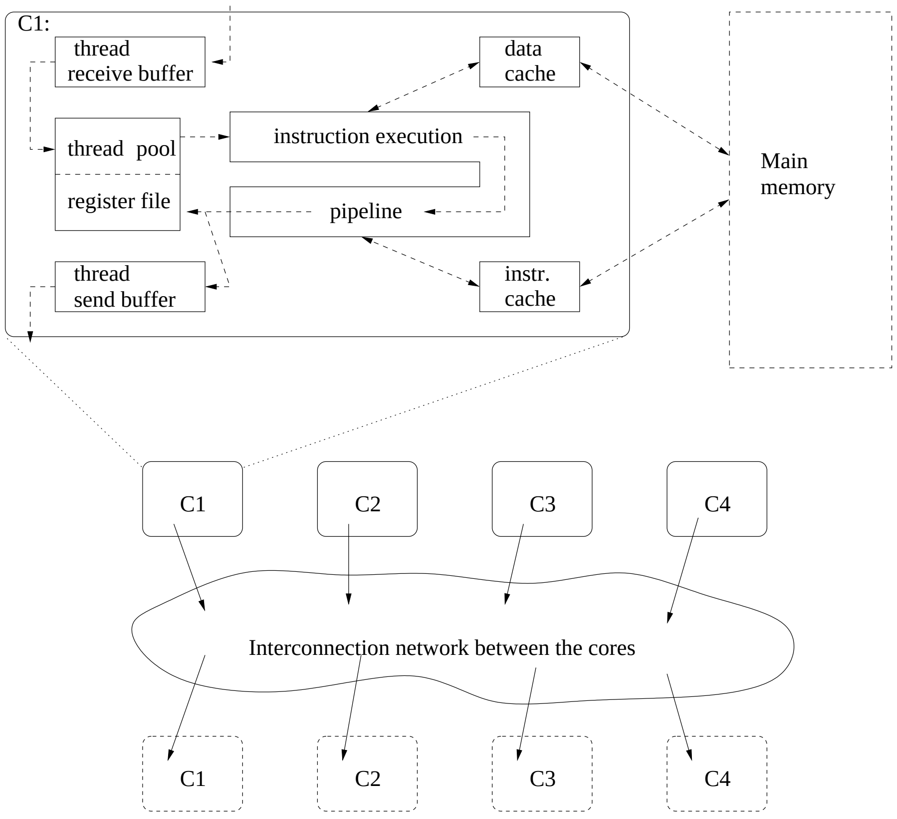


Figure 1: Overview of our multicore system.

The system consists of c RISC-based *cores*, an inter-connection *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.

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 a cache for program instructions.

The idea is that each of the cores has at each logical step a lot of threads (varying number) 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 some of the delays caused by moving the threads. A key idea is to *hide the memory as well as network and other delays* by keeping the average number of threads per core at the same or higher level than the expected delay of executing a single instruction from any thread.

For the creation and termination of threads at 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 of the threads starts by running the same program block. The logical thread id-numbers are independent of physical thread and core numbers. Management of the physical thread identity is completely handled by the hardware. The logical id-numbers are program controlled, with an easy access via register instructions. 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).

2.1 Memory system

The moving threads architecture consists of three types of memory. First, the global main memory is divided into two types: a read-only instruction memory and a read-write data memory. The third memory type consists of the local storage provided by thread local registers. In the current implementation, the word length of 32 bits has been chosen for all registers. Unlike the registers, the global main memory types are shared between all threads.

The data and instruction memories have a flat 32-bit address space. The instruction memory is automatically accessed by the execution engine whenever new instructions need to be fetched, while the data memory is accessed explicitly from the program code via load and store instructions. The local storage provided by the registers can be used via instructions with register parameters. Since all memory types have a dedicated access method, all kinds of memory conflicts can be avoided. The data memory's address space is assumed to be fully populated with physical memory modules, however not necessarily with a unique module for each core since the amount of interconnections with the processor may have strict physical limitations. To simplify the model, we do not consider any hardware exceptions related to memory in this report.

All memory operations (and other instructions) have a unit time cost in the computational model. Physically the memory operations may have varying fetch latencies depending on the exact location of the data in the memory hierarchy. When the latency prevents a thread from executing, the scheduler automatically interleaves the execution with other instructions from other threads that are ready for execution. Obtaining optimal performance requires optimizing both instruction and data memory accesses by using this parallel slackness [8].

Optimizing instruction memory access is very limited. The architecture can provide branch prediction, caching, and prefetching of cache lines as core local optimizations like in contemporary designs. Another optimization can be achieved with multithreading. If threads on the same core execute the same part of the program, the threads can share the same instruction data and decrease instruction memory traffic via core's local instruction cache.

The architecture does not provide any explicit way for optimizing the use of data memory. First, the memory is partitioned into blocks ($2^{16} \ldots 2^{32}$) that are scattered evenly among the cores using some hash function. Currently, the hash function is calculated from the highest n bits of the memory reference. Accessing memory on another core triggers context switch and thread migration to that other core. Optimal usage utilizes the computational power and the memory bandwidth provided by the distributed low level architecture, but also tries to minimize the thread moves, because the interconnection network also introduces limited bandwidth and a communication latency. Locally within the core, when accessing memory, the application can also make use of the memory hierarchy parameters (cache line width, associativity, replacement policy), which is a common coding practice with contemporary systems with hierarchical memory.

The registers provide a constant time, low latency access. Unlike global main memory, which is partitioned between the cores, the register space is local to the thread and migrates along with the thread when the thread is sent to another core or forked.

2.2 Interconnection network

The details of the interconnection network for the moving threads architecture are discussed in another technical report about the implementation of the architecture simulator [9].

2.3 Execution core

Each core (Figure 2) keeps track of the thread context for each active thread executing in the core. The thread scheduling and migration are handled implicitly by the hardware. The table of local threads can be further partitioned into n (e.g. 4 or the number of pipeline stages) subtables to improve thread switching performance and possibly the register file bandwith with register banks. In case the table is divided into subtables, the optimal execution requires populating all subtables with sufficient number of threads to hide the latency introduced by memory accesses and moving of threads, and to avoid pipeline stall if a subtable runs out of executable threads. Currently, we only consider that $n = 1$.

The sufficient number of simultaneous threads depends dynamically on the algorithm, and statically on memory hierarchy properties and the network latencies. Currently we do not have any realistic estimate for these, but the SMASim architecture simulator [10] (Section 6.4) could be used to predict dynamic latencies.

By default a strict PRAM style synchronization is achieved by the synchronization wave technique [7]. The wave implicitly places a synchronization barrier between every two instructions for all active threads. During a single virtual time step of the PRAM, the threads are executed in arbitrary order. The hardware guarantees a serial execution of memory requests during this step. However, the contents of a memory address and the result of memory operations in case of simultaneous writes or simultaneous writes and reads are undefined. This means that the architecture follows the EREW (Exclusive Read, Exclusive Write) PRAM computational model. In fact, the architecture can semantically be seen as an aCRCW (Arbitrary Concurrent Read, Concurrent Write) model, but there is no special support for concurrent operations and thus their cost is larger than on the ordinary CRCW.

An alternative operation for the system (not implemented currently) is to trigger the synchronization wave explicitly with a dedicated synchronization instruction. The idea behind relaxing this model is to decrease the amount of synchronization messages in the network, give more opportunity for latency hiding to work efficiently and improve the performance by removing unnecessary data dependencies with more independent thread local computation.

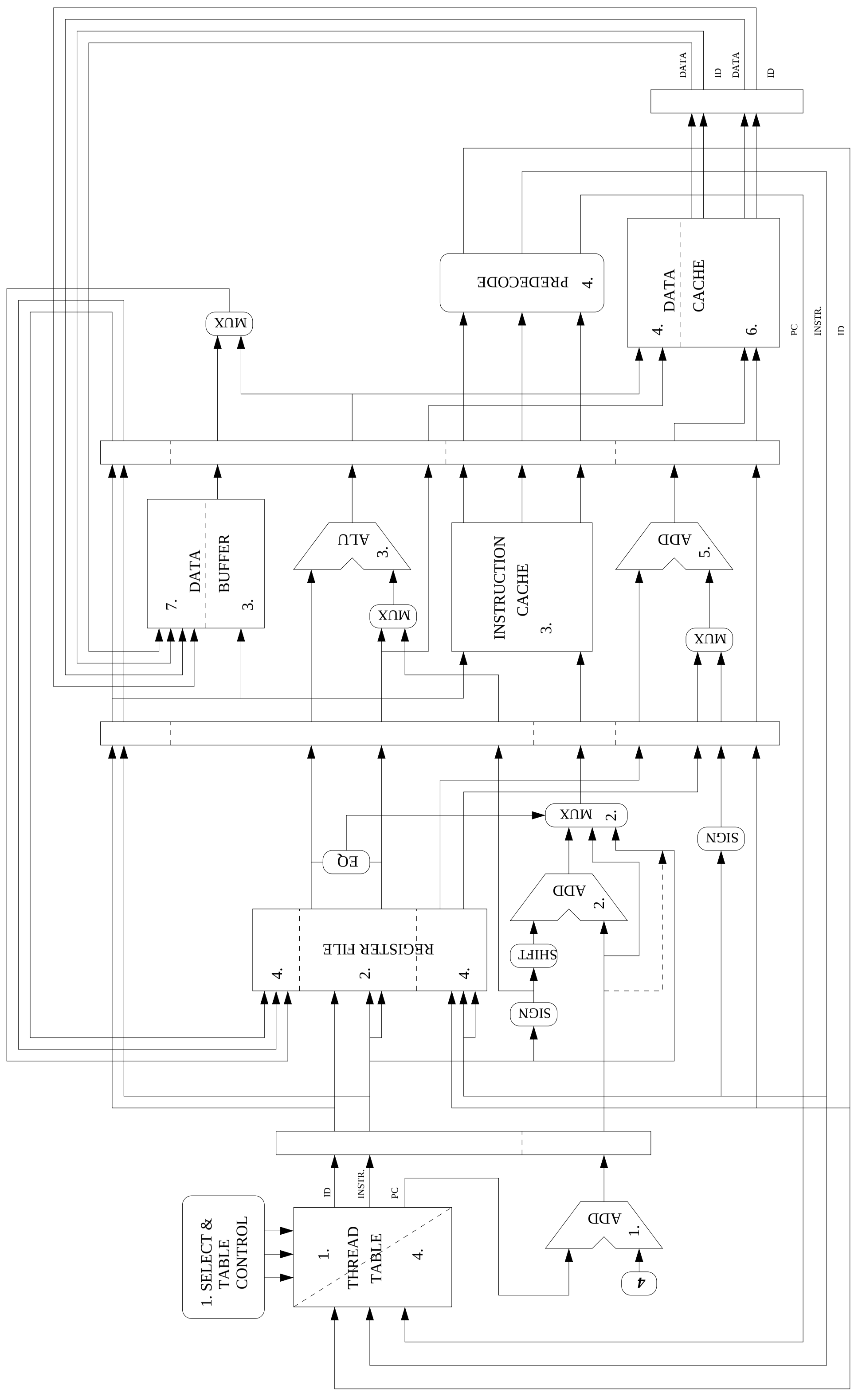


Figure 2: Execution core's datapath.

3 Instruction set

The architecture was designed to be based on the in-order RISC load-store architecture [5]. The rationale behind this is that a RISC architecture does not require complex hardware implementation. Extracting the memory addresses from loads and stores is rather simple (used when migrating threads). In addition, instead of improving a single core's execution performance by a hardware support for out-of-order execution, we rather use the chip space for multiple cores. The first generation of the design does not support SIMD instructions, but SIMD operations could provide a cost-effective way to improve performance of parallel algorithms.

In order to boost application development by making use of existing development toolchain, we settled on MIPS32 ISA [16]. Another plausible alternative would be the ARM ISA [15]. From toolchain's point of view, the instruction set is not hard to switch later on – changes currently only affect the backend of the compiler and the architecture emulator's execution pipeline model.

3.1 MIPS instruction set

The basic set of instructions consists of the MIPS32 instruction set. However, currently floating point, atomic, and trap instructions have no implementation. Because of the execution pipeline design, instructions using pipeline registers `hi` and `lo` can be more freely used in various contexts unlike in the traditional MIPS32. These registers are thread local, but do not migrate with the thread, which forces the compiler to make sure the values are stored to regular registers before a move. In other cases, all instructions can be executed in any sequence without any special cases.

The supported MIPS32 instructions can be divided in four categories:

- **Arithmetic–logic**: ADDI, ADDIU, ADD, ADDU, ANDI, AND, CLO, CLZ, MOVN, MOVZ, MUL, NOP, NOR, ORI, OR, SLL, SLLV, SLTI, SLTIU, SLT, SLTU, SRA, SRAV, SRL, SRLV, SUB, SUBU, XORI, and XOR.

- **Heavyweight Arithmetic**: DIV, DIVU, MADD, MADDU, MFHI, MFLO, MSUB, MSUBU, MTHI, MTLO, MULT, and MULTU.

- **Load and stores**: LB, LBU, LH, LHU, LUI, LWL, LWR, LW, SB, SH, and SW.

- **Branches and jumps**: BEQ, BGEZAL, BGEZ, BGTZ, BLEZ, BLTZAL, BLTZ, BNE, JALR, JAL, JR, and J.

The differences between the first two categories are in the use of `hi` and `lo` registers. The arithmetic instructions in the latter category store the result in two registers. All other instructions write back the result in at most one register.

3.2 Thread instructions

The concurrency model for the moving threads architecture was motivated by the fork–join programming model; the control flow is organized as subsequent blocks of sequential and parallel code (see Figure 3). The blocks can then be recursively split further into a new sequence of these blocks. The architecture forks new threads at the beginning of a parallel block and joins the created child threads at the end of the block. These operations are explicitly handled with two kinds of thread instructions: forks and joins. A detailed explanation of the instructions and their effects is given next in this section.

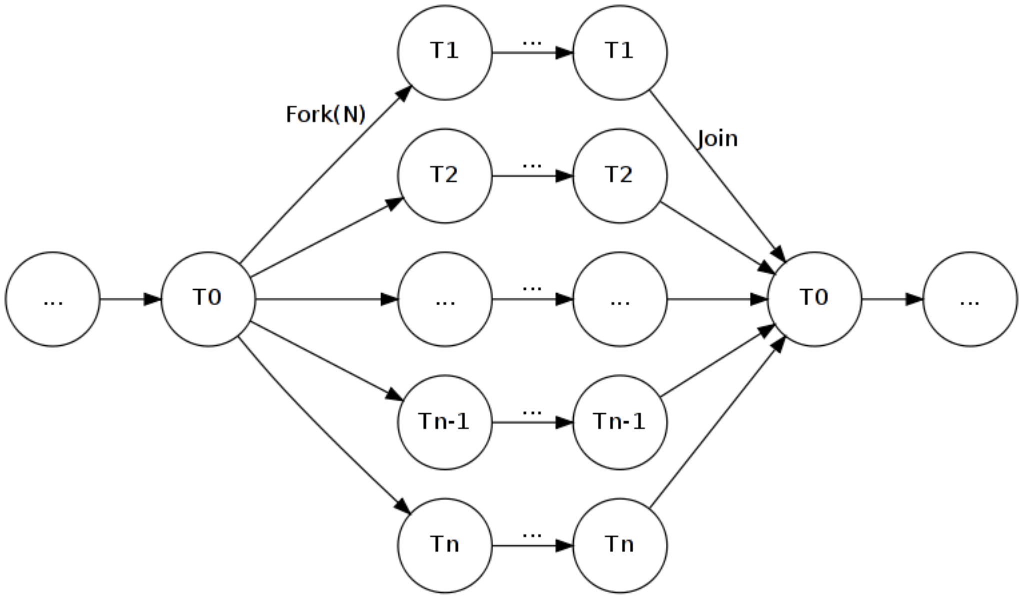


Figure 3: The fork–join pattern.

The architecture does not require a strict symmetry between thread instructions, i.e. the high level fork–join model provides only a subset of the available functionality and is one of the possible parallel programming models; the instruction set uses the forks and joins as primitives. There are also query functions for discovering the current thread index. Multi-level indexes are implementable, but the hardware only keeps track of the topmost index. The indices of the ”outer loops” must be stored to a stack implemented with either registers or a dedicated main memory location.

To simplify the implementation, thread instructions use the same shared general purpose registers for storing their state. The programmer’s responsibility is to keep track of this data (mostly thread index value after a `Fork`), and to restore the thread to a consistent state before calling `Join` (core and thread slot index values). The `Pardo` macro defined in Section 4.3.4 is given as an example of a high level construct designed to automatically keep track of this data for the programmer.

3.2.1 Fork

The `Fork` operation forks new child threads. The amount of new threads (m) to create is given either as an immediate argument (encoded as part of the instruction word) or as a register argument (index of the register containing the number).

The thread creation is split evenly among the cores. That is, on a system consisting of n processor cores, the equation $m = a \cdot n + b$, where $0 < b < n$, defines the numbers of threads to fork on each of the cores. The a represents the amount of new threads on cores $1 \ldots (n-1)$, b on the nth core. If b is zero, all cores fork a new threads. In case the number of threads to fork is smaller than the amount of cores, all child threads are forked locally on the same core.

The new threads get a unique id, starting from 0 and ending with $m - 1$. This enumeration follows the numbering of the cores: the id of the first thread on core i comes after the id of the last thread on core $i - 1$. All the other thread local data (registers, program counter) are copied from the parent thread. Typical ways to achieve parallelism under this assumption is to access different data using the varying id number or use it for branching to make the concurrent control flows diverge.

On top of this, the actual implementation of `Fork` is a bit more complex. A distributed thread forking is used to minimize network traffic; the amount of packets to send is proportional to the amount of cores, not to the amount of threads to fork. To achieve this, the fork is divided to two operations, the remote and local fork. The idea of the remote fork is to propagate the knowledge of the fork to all cores. The cores then locally fork the right number of threads. Instead of a single master thread, this approach also requires per-core coordinator threads. All created threads keep track of the originating core and thread position in the thread table to make the joins later possible in a distributed manner.

Next we list the actual instructions used to implement the described fork functionality. The `FORK` and `FORKI` instructions describe the remote forks, and the `FORKL` describes the version local to every core.

FORKI The execution of `FORKI` broadcasts a thread creation packet to each core (Figure 4).

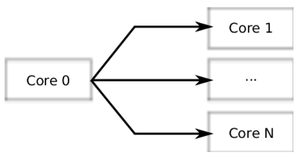


Figure 4: Broadcast of the fork packet to other cores.

The semantics of the instruction are:

1. `pc` ← thread's old `pc` value + 4
2. nextInstruction ← next instruction in the instruction memory
3. register 27 ← number of coordinating threads made
4. state ← async wait

The outgoing network packet consists of the issuing thread's state, with the following differences:

1. nextInstruction ← `FORKL`
2. register 25 ← issuing core's id
3. register 26 ← issuing thread's slot id
4. register 27 ← total number of threads to create

As `FORKI` is a type I instruction in MIPS terminology (see [16]), the number of threads to create is given as an immediate value.

FORK The semantics of `FORK` are identical to those of `FORKI`. However, instead of providing the number of threads to create as an immediate value, the value is indirectly given via a register reference. `FORK` is thus a type R instruction in MIPS terminology.

FORKL The local fork instruction spawns new child threads to the thread table of the local core. The number of threads to fork is read from the register 27. The semantics of the instruction are:

1. `pc` ← thread's old `pc` value + 4
2. nextInstruction ← next instruction in the instruction memory
3. register 27 ← number of child threads made on this core
4. state ← async wait

The child threads are initialized with the following state:

1. `pc` ← thread's old `pc` value + 4
2. nextInstruction ← next instruction in the instruction memory
3. register 25 ← issuing core's id
4. register 26 ← issuing thread's slot id
5. register 27 ← child thread's id

The technique for computing the id values and amount of threads to fork on each core were described in the beginning of this section.

3.2.2 Join

We assume that before calling any `Join`s, the code inside the parallel block has restored all thread related registers (registers 25 and 26) to the original state after a previous `Fork`. Otherwise the operations will have undefined and potentially harmful behavior.

While the child threads are executing, the parent thread and the per-core coordinator threads are permanently waiting. Nothing will trigger these threads until all the child threads have performed the `Join`. This leads to few invariants: the core and the thread slot in the thread table remain constant during the execution of the parallel code block. The child threads use this information to locate the original thread when joining; the `Join` will active thread move if execution is attempted on a wrong core. The coordinating threads contain a counter which is decremented by an associated child thread every time the child thread joins. The last `Join` operation terminates the child thread and activates the coordinating thread. In similar way, the coordinating threads decrease the counter of the parent thread after locating the correct core.

Next we list the actual instructions used to implement the described join functionality. The `JOINC` instruction describes the join of a coordinator thread, and the `JOIN` instruction is directly used by the child threads.

JOIN The execution of instruction `JOIN` performs the following operations:

1. The core id from the register 25 (rid) is compared with the core's id (cid). If $rid \neq cid$, the thread move to core rid is started (Figure 5).
2. If $rid = cid$, the value of the register 27 ($r27$) is decreased. However, the thread, whose register is read from and written back to, is defined by the value of the register 26 ($r26$).
3. The program counter of the thread in slot $r26$ is replaced with the program counter value of the issuing thread.
4. The next instruction value of the thread in slot $r26$ is replaced with *JOINC*.
5. If the newly written register value $r27$ in slot $r26$ becomes zero, the thread in slot $r27$ is activated (state transition to *Ready*).
6. The issuing thread is terminated (state transition to *Free*).

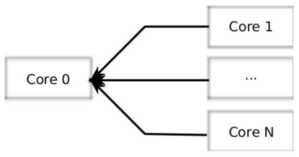


Figure 5: Migration of the join thread packets to the parent core.

As a result, the child thread terminates and the coordinating thread continues its execution with the `JOINC` instruction. Due to the indeterministic execution order of concurrent threads during the same time step (see Section 2.3), a deterministic point for joins might be required if the child threads perform branching.

The child threads created with a `Fork` instruction should end their execution with this instruction. In case the number of forked child threads was less than the number of cores, the indirection caused by the coordinator threads is not used – in that case the child threads directly manipulate the state of the parent thread by using the `JOINC` instruction, which is described next.

JOINC The `JOINC` instruction is used ”internally” by the moving threads processor, when a coordinator thread activates to join to the parent thread. The instruction should be also used directly when no coordinator threads were used.

The execution of instruction `JOINC` performs the following operations:

1. The core id from the register 25 (rid) is compared with the core's id (cid). If $rid \neq cid$, the thread move to core rid is started.
2. If $rid = cid$, the value of the register 27 ($r27$) is decreased. However, the thread, whose register is read from and written back to, is defined by the value of the register 26 ($r26$).
3. The program counter of the thread in slot $r26$ is replaced with the program counter value of the issuing thread + 4.
4. The next instruction value of the thread in slot $r26$ is replaced with *NOP* (no operation).
5. If the newly written register value $r27$ in slot $r26$ becomes zero, the thread in slot $r27$ is activated (state transition to *Ready*).
6. The issuing thread is terminated (state transition to *Free*).

As a result, the coordinator thread terminates and the parent thread continues its execution first with NOP, then from the instruction following the `Join` of the last child thread. Due to the indeterministic execution order of concurrent threads during the same time step (see Section 2.3), a deterministic point for joins might be required if the child threads perform branching. This time the program counter value is chosen from the program counter values propagated to the coordinator threads.

3.2.3 Other thread instructions

HALT In addition, an explicit `HALT` instruction is provided to stop the execution of a thread. `HALT` should be only used in the parent thread since it does no checks to see whether some other thread is waiting for the thread to join. The instruction does not perform any computation, it only sets the thread's status to *Free*. This effectively kills the thread by preventing the thread selection logic from picking the thread anymore for execution.

3.3 An execution example with thread instructions

To demonstrate the use of thread instructions, we use a simple fork–join block with only the NOP (no operation) instruction as its body (Figure 6).

```
FORKI 16
FORKL
NOP
JOIN
JOINC
HALT
```

Figure 6: A simple four instruction fork–join example.

The amount of threads to create, 16 in the above example, is given as an immediate parameter. Using the forementioned semantics, the `FORKI` instruction forks 16 new threads and sets the parent thread in wait mode. The `NOP` instruction is executed by each of the child threads. The child threads finish their execution after issuing the `JOIN`. The system internally proceeds with `JOINC` after the `JOIN` command. Finally, the program halts the execution with `HALT`. The timing of these instructions on each core is listed in Tables 1 and 2. The number of cores in this system is 15.

Thread creation						
Thread id	Core	Instruction	PC	r25 (core)	r26 (slot)	r27 (count)
Time: t (*Fork*)						
0	0	FORKI(16)	100	?	?	-
Time: t+1 (*Local fork*)						
0	0	FORKL *Wait*	104	?	?	1234
1	0	FORKL	104	0	0	16
…						
15	14	FORKL	104	0	0	16

Table 1: Execution of FORK and FORKL.

Thread creation						
Thread	Core	Instruction	PC	r25 (core)	r26 (slot)	r27 (count)
Time: t+2 (*Inside the block*)						
0	0	FORKL *Wait*	104	?	?	1234
1	0	FORKL *Wait*	108	0	0	2
2	1	FORKL *Wait*	108	0	0	1
...						
15	14	FORKL *Wait*	108	0	0	1
16	15	NOP	108	14	15	15
...						
29	1	NOP	108	1	2	2
30	0	NOP	108	0	1	1
31	0	NOP	108	0	1	0
Time: t+3 (*Join*)						
0	0	FORKL *Wait*	104	?	?	1234
1	0	FORKL *Wait*	108	0	0	2
2	1	FORKL *Wait*	108	0	0	1
...						
15	14	FORKL *Wait*	108	0	0	1
16	15	JOIN	112	14	15	15
...						
29	1	JOIN	112	1	2	2
30	0	JOIN	112	0	1	1
31	0	JOIN	112	0	1	0
Time: t+4 (*Coordinator join*)						
0	0	FORKL *Wait*	104	?	?	1234
1	0	JOINC	112	0	0	0
2	1	JOINC	112	0	0	0
...						
15	14	JOINC	112	0	0	0
Time: t+4 (*After migration*)						
0	0	FORKL *Wait*	104	?	?	1234
1	0	JOINC	112	0	0	0
2	0	JOINC	112	0	0	0
...						
15	0	JOINC	112	0	0	0
Time: t+5 (*NOP after JOINC*)						
0	0	NOP	116	?	?	0
Time: t+6						
0	0	HALT	120	?	?	0

Table 2: Execution of JOIN and HALT.

3.4 Ideas for improving the instruction set

Even though the forks and joins are independent instructions in our architecture, the currently implemented semantics do not directly support a lightweight mechanism for creating child threads that do not join, i.e. the parent thread always starts waiting. Some other concurrency models may expect a set of instruction for creating totally independent threads.

The return address of the parallel block is currently propagated from the child threads to the parent thread (possibly via coordinators). The parent thread could also be explicitly provided with a return address.

It seems somewhat unrealistic to expect the thread operations to execute this much functionality in a single time step. A physical prototype of the architecture might split the concurrency instructions into smaller pieces.

4 Programming language

The role of the programming language is to provide a level of abstraction for programming the underlying architecture. In the context of the moving threads architecture, our goal was to design a suitable language for programming the architecture on higher level than the assembler. The current implementation is based on the C programming language and is provided as a 3rd party library extension. The main goal was to provide a way to use the fork–join-model via high level loop constructs with an automatic index tracking (similar to `for()` in C).

4.1 Memory model

The virtual shared memory abstraction can be modeled with a simple linear dense address space. This is similar to C's traditional memory model: We fill the address space from one end with the stack and from the other with the heap. No kind of thread level safety mechanism is provided for the main memory.

The thread local storage in registers needs another mechanism for access. All the concurrency instructions use the same general purpose registers for temporary and return values as the other instructions. It is the responsibility of the programmer or compiler to explicitly protect the registers against accidental overwrites. These must be handled by e.g. the language's flow analysis by calling them via functions (e.g. in a modified C language).

The current runtime implementation does not contain implementation for dynamic heap memory management nor dynamic stack. On architecture level the system does not support virtual memory yet. As a temporary solution, the stack could also be defined statically by dividing a subset of the address space evenly among threads by the compiler.

The reason for omitting these for now are potential performance issues with the memory accesses. For instance, even though some of the adjacent stack addresses reside on the same core, a stack access might incur expensive thread migrations. In worst case, every stack access triggers a thread move. In contemporary systems the stack access is considered an efficient storage (no heap fragmentation, fast allocation and deallocation, works well with caches).

Clearly, the situation is at least as bad when dealing with the heap memory. In addition, dynamic memory management with e.g. a centralized bookkeeping structure might trigger a series of thread moves every time a memory area is allocated and deallocated.

4.2 Concurrency model

In short, our language for the moving threads architecture leaves the safety questions related to the concurrent execution almost completely unsolved. The undeterministic aCRCW hardware model explains how simultaneous reads and writes to the same main memory address only lead to undefined state. In one sense this is compatible with C's single-threaded memory model, because the isolation between threads is handled on hardware, and the application code only needs to guarantee the freedom from these memory conflicts. The C language cannot make this guarantee, but a runtime library or an improved language can be designed to provide a set of parallel constructs that will not break this guarantee. As no single concurrency model or language has become de facto solution, we have left the question unanswered.

4.3 Language constructs

Currently the following language constructs are provided by the runtime library: *thread index query*, *fork*, *join*, and *pardo* (parallel do). The first three provide an explicit way of controlling the thread handling, while *pardo* provides implicit synchronization and thread creation for the provided block of parallel code. Each of the constructs is explained in more detail in the following Sections 4.3.1 . . . 4.3.5. The functions are defined in the file `moth.h`.

4.3.1 Thread index query

The thread index query is provided by the function `int moth_get_index()`. The function returns the thread's number from the reserved register after a `Join` operation (Section 3.2.2). Assigning the value to a variable prevents the programming language from rewriting it. The value is important because it provides the only point of variance between newly created threads.

The definition of the function is listed in Figure 7 and the resulting machine code in Figure 8. The generated machine code varies between contexts. Here it copies the value to another register `$v0` from the original location, register `$k1`.

```
static __attribute__((always_inline))
int moth_get_index() {
  register int idx;
  asm("move %0, $k1" : "=r" (idx) : : "0");
  return idx;
}
```

Figure 7: Listing of the function `int moth_get_index()`.

```
<moth_get_index>:
   0: 03601021  move    v0,k1 // copy index to a
   4: 03e00008  jr      ra    // register variable
```

Figure 8: Listing of the function `int moth_get_index()` implementation.

4.3.2 Fork

The hardware instructions for the fork operations are provided with the function `void moth_fork(int count)`. The function takes one argument, *count*, the number of child threads to create. The parent thread halts its execution until all child threads have joined.

The definition of the function is listed in Figure 9 and the resulting machine code in Figure 10. The machine code assumes that the compiler passes the parameter value via the register `$v0`.

```
static __attribute__((always_inline))
void moth_fork(register int count) {
  register int c asm("v0") = count;
  asm(".word (1879048192+(6<<3))"::"r"(c):"0");
  asm(".word (1879048192+(6<<3)+1)"); // FORK + FORKL
}
```

Figure 9: Listing of the function `void moth_fork(int count)`.

```
<moth_fork>:
   0: 00801021  move    v0,a0  // param as $a0
   4: 70000030  fork    v0
   8: 70000031  forkl
   c: 03e00008  jr      ra
```

Figure 10: Listing of the function `void moth_fork(int count)` implementation.

4.3.3 Join

The hardware instructions for the join operations are provided with the function `void moth_join()`. The function joins the child thread by updating the the counter of the parent thread and by terminating it. When all child threads have joined, the execution continues via the parent thread.

The definition of the function is listed in Figure 11 and the resulting machine code in Figure 12.

```
static __attribute__((always_inline))
void moth_join() {
  asm(".word (1879048192+(6<<3)+2) ");   // JOIN
  asm(".word (1879048192+(6<<3)+3) ");   // JOINC
}
```

Figure 11: Listing of the function `void moth_join()`.

```
<moth_join>:
   0: 70000032  join          // join the child
   4: 70000033  joinc         // join the coordinator
   8: 03e00008  jr      ra
```

Figure 12: Listing of the function `void moth_join()` implementation.

4.3.4 Pardo

The *parallel do* function with the signature `pardo(idx, max, block)` combines the three previous operations. It spawns max child threads, assigns the thread index to a thread local variable idx for each thread and starts executing the code specified in $block$ with all child threads. The end of the block determines an implicit synchronization point where the child threads are joined.

```
#define pardo(idx, max, block) \
  moth_fork(max);\
  {\
    int idx = moth_get_index();\
    block\
  }\
  moth_join();
```

4.3.5 Halt

A `halt` function is also provided as a wrapper for the `HALT` instruction for terminating threads or the whole application.

The definition of the function is listed in Figure 13 and the resulting machine code in Figure 14.

```
static __attribute__((always_inline))
void moth_halt() {
  asm(".word (1879048192+(6<<3)+4)");    // HALT
}
```

Figure 13: Listing of the function `void moth_halt()`.

```
<moth_join>:
   0: 70000034  halt        // halt the thread
   8: 03e00008  jr      ra
```

Figure 14: Listing of the function `void moth_halt()` implementation.

5 Installing the compiler

This section provides step by step instructions for compiling and installing our compiler. The compiler simply consists of the GNU GCC toolchain with the `mips32` target and a simple library for moving threads architecture containing a small set of architecture specific compile time and runtime functionality.

Some operating system distributions are equipped with a GCC compiler with the ability to cross-compile to the `mips32` target. We have tested some of these, e.g. the Embedded Debian Project (`http://www.emdebian.org/`). These packages may or may not work. Worth noting is that the executables from these packages may have different naming conventions (with the custom build the tool names are prefixed with ”mips-”). As the installation instructions for these packages can be found from the websites of the projects, we do not discuss them here.

5.1 Downloading the compiler and dependencies

In these instructions we use the source code releases of GNU Binutils version 2.20.1 (linker, assembler) and GNU GCC version 4.5.1 to build our toolchain. The Binutils release is available from `http://www.gnu.org/software/binutils/` and the GCC compiler from the mirrors listed in `http://gcc.gnu.org/gcc-4.5/`.

The toolchain’s release notes list a set of build dependencies such as GNU binutils and GNU autotools. As we use the Debian Linux distribution in these instructions, at the time of writing, the corresponding command for installing the dependencies in Debian is presented in Figure 15.

```
# apt-get install flex bison libgmp3-dev libmpfr-dev
# apt-get install autoconf texinfo build-essential
```

Figure 15: Installation of gcc’s dependencies on Debian.

Our moving threads library, `moth.h`, and other possibly related files (e.g. the most recent version of this manual) are freely available from `http://staff.cs.utu.fi/research/MOTH/`.

5.2 Compiling and installing the compiler

We assume that the (x86) system contains a fully functional compilation toolchain with the dependencies listed in Section 5.1 successfully installed. The following instructions compile binutils and gcc and install both to `/usr/local/`. The file `moth.h` from our distribution can be installed in many possible directories, for instance in `/usr/local/include/`.

To make building faster, only the MIPS backend and the C frontend are necessary to compile the examples shown in this manual and more generally the programs written in the C programming language. The Figure 16 shows the installation of binutils; the commands for compiling and installing GCC are shown in Figure 17.

```
$ tar xf binutils-2.20.1.tar.bz2
$ cd binutils-2.20.1/
$ ./configure -target=mips
$ make

$ sudo make install
```

Figure 16: Installation of binutils.

```
$ tar xf gcc-4.5.1.tar.bz2
$ cd gcc-4.5.1/
$ mkdir objdir
$ cd objdir
$ ../configure --enable-languages=c
               --enable-targets=mips
               -target=mips
               -disable-threads
               --disable-multilib
               --disable-libmudflap
               --disable-libssp
$ make

$ sudo make install
```

Figure 17: Installation of gcc.

5.3 Testing the compiler

The compiler should now be fully operational. We can test it by constructing and compiling a simple program example (Figure 18). The self-contained example should compile without errors. If the library `moth.h` cannot be found, check the include path settings. However, the resulting executable is not usable on our architecture since we do not have a proper operating system built yet. The way to build useful programs is explained in the next section.

```
$ echo "#include \"moth.h\"" > test.c
$ echo "int main(void) { pardo(i, 42, ) }" >> test.c

$ mips-gcc test.c -o test
```

Figure 18: Testing the toolchain.

6 Using the compiler

Currently our architecture comes without any kind of an operating system or production ready support for one. As a result the ELF type binaries typically produced by GCC cannot be loaded. This forces us to produce raw binaries.

The raw binaries have few issues we need to solve when writing applications. First, the C programs consist of a `main()` function. The location of `main()` varies between raw binaries and we can only set up the memory locations of code at the level of modules (compilation units). We solve this with the linker (linker script in Figure 19) by relocating a short jump code (jump code in Figure 20) to the beginning of `main()` to the hexadecimal address 0x4. Another problem is the manual administration of the various sections (various types of static data, code) in the memory space.

```
SECTIONS {
  outputa 0x4:
    {
    init.o (.text)
    }
  outputb 0x12:
    {
    ${T}.o (.text)
    }
  .sdata 0x1000:
    {
    *(.sdata)
    }
  .bss :
    AT( 0x1000 + SIZEOF(.sdata))
    {
    }
}
```

Figure 19: Linker script for the resulting raw binary.

```
extern int main(void);

void __start() {
  main();
}
```

Figure 20: Jump code to the beginning of main().

The main tools for generating code for out architecture are `mips-as` (GNU assembler), `mips-ld` (GNU linker), and `mips-gcc` (GNU C compiler). The resulting binary can be analyzed with `objdump` (GNU object file disassembler). Since the platform has been highly experimental so far, we provide a simple script for managing the forementioned issues. The script also displays both optimized and unoptimized disassembler output for each function in the object file and the for the resulting binary. The structure and use of this script is described in Section 6.1.

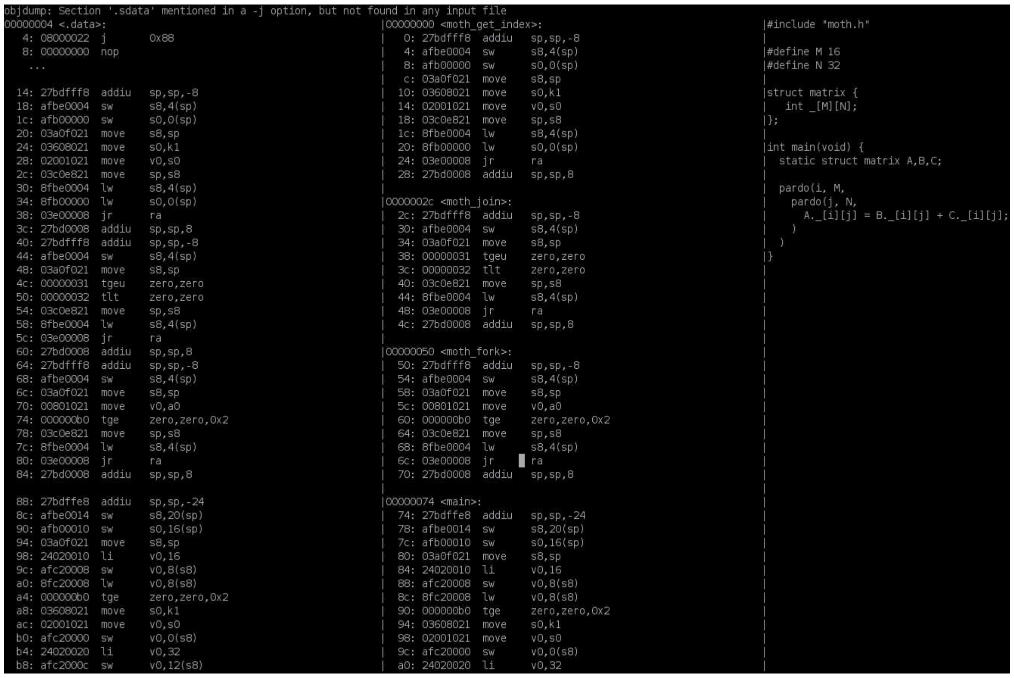
6.1 Using the frontend script

The basic usage of the frontend script is:

```
./compile.sh [source] [compiler parameters]
```

The compile.sh script accepts a single C source code as parameter. If the compilation process needs to be customized further, a list of command line switches for the compiler can be passed after the source file name.

The frontend script compiles the application and also displays the code for final executable, object file, and the source file in three columns. In the first two columns both the unoptimized and optimized versions are displayed in a sequence for debugging purposes. As an example, Figure 21 shows the compilation of output using the example code from Table 3 as its input.



```
objdump: Section '.sdata' mentioned in a -j option, but not found in any input file
00000004 <.data>:                               |00000000 <moth_get_index>:                      |#include "moth.h"
   4: 08000022  j       0x88                    |   0: 27bdfff8  addiu   sp,sp,-8                |
   8: 00000000  nop                             |   4: afbe0004  sw      s8,4(sp)                |#define M 16
    ...                                         |   8: afb00000  sw      s0,0(sp)                |#define N 32
                                                |   c: 03a0f021  move    s8,sp                   |
  14: 27bdfff8  addiu   sp,sp,-8                |  10: 03608021  move    s0,k1                   |struct matrix {
  18: afbe0004  sw      s8,4(sp)                |  14: 02001021  move    v0,s0                   |   int _[M][N];
  1c: afb00000  sw      s0,0(sp)                |  18: 03c0e821  move    sp,s8                   |};
  20: 03a0f021  move    s8,sp                   |  1c: 8fbe0004  lw      s8,4(sp)                |
  24: 03608021  move    s0,k1                   |  20: 8fb00000  lw      s0,0(sp)                |int main(void) {
  28: 02001021  move    v0,s0                   |  24: 03e00008  jr      ra                      |  static struct matrix A,B,C;
  2c: 03c0e821  move    sp,s8                   |  28: 27bd0008  addiu   sp,sp,8                 |
  30: 8fbe0004  lw      s8,4(sp)                |                                                |  pardo(i, M,
  34: 8fb00000  lw      s0,0(sp)                |0000002c <moth_join>:                           |    pardo(j, N,
  38: 03e00008  jr      ra                      |  2c: 27bdfff8  addiu   sp,sp,-8                |      A._[i][j] = B._[i][j] + C._[i][j];
  3c: 27bd0008  addiu   sp,sp,8                 |  30: afbe0004  sw      s8,4(sp)                |    )
  40: 27bdfff8  addiu   sp,sp,-8                |  34: 03a0f021  move    s8,sp                   |  )
  44: afbe0004  sw      s8,4(sp)                |  38: 00000031  tgeu    zero,zero               |}
  48: 03a0f021  move    s8,sp                   |  3c: 00000032  tlt     zero,zero               |
  4c: 00000031  tgeu    zero,zero               |  40: 03c0e821  move    sp,s8                   |
  50: 00000032  tlt     zero,zero               |  44: 8fbe0004  lw      s8,4(sp)                |
  54: 03c0e821  move    sp,s8                   |  48: 03e00008  jr      ra                      |
  58: 8fbe0004  lw      s8,4(sp)                |  4c: 27bd0008  addiu   sp,sp,8                 |
  5c: 03e00008  jr      ra                      |                                                |
  60: 27bd0008  addiu   sp,sp,8                 |00000050 <moth_fork>:                           |
  64: 27bdfff8  addiu   sp,sp,-8                |  50: 27bdfff8  addiu   sp,sp,-8                |
  68: afbe0004  sw      s8,4(sp)                |  54: afbe0004  sw      s8,4(sp)                |
  6c: 03a0f021  move    s8,sp                   |  58: 03a0f021  move    s8,sp                   |
  70: 00801021  move    v0,a0                   |  5c: 00801021  move    v0,a0                   |
  74: 000000b0  tge     zero,zero,0x2           |  60: 000000b0  tge     zero,zero,0x2           |
  78: 03c0e821  move    sp,s8                   |  64: 03c0e821  move    sp,s8                   |
  7c: 8fbe0004  lw      s8,4(sp)                |  68: 8fbe0004  lw      s8,4(sp)                |
  80: 03e00008  jr      ra                      |  6c: 03e00008  jr      ra                      |
  84: 27bd0008  addiu   sp,sp,8                 |  70: 27bd0008  addiu   sp,sp,8                 |
                                                |                                                |
  88: 27bdffe8  addiu   sp,sp,-24               |00000074 <main>:                                |
  8c: afbe0014  sw      s8,20(sp)               |  74: 27bdffe8  addiu   sp,sp,-24               |
  90: afb00010  sw      s0,16(sp)               |  78: afbe0014  sw      s8,20(sp)               |
  94: 03a0f021  move    s8,sp                   |  7c: afb00010  sw      s0,16(sp)               |
  98: 24020010  li      v0,16                   |  80: 03a0f021  move    s8,sp                   |
  9c: afc20008  sw      v0,8(s8)                |  84: 24020010  li      v0,16                   |
  a0: 8fc20008  lw      v0,8(s8)                |  88: afc20008  sw      v0,8(s8)                |
  a4: 000000b0  tge     zero,zero,0x2           |  8c: 8fc20008  lw      v0,8(s8)                |
  a8: 03608021  move    s0,k1                   |  90: 000000b0  tge     zero,zero,0x2           |
  ac: 02001021  move    v0,s0                   |  94: 03608021  move    s0,k1                   |
  b0: afc20000  sw      v0,0(s8)                |  98: 02001021  move    v0,s0                   |
  b4: 24020020  li      v0,32                   |  9c: afc20000  sw      v0,0(s8)                |
  b8: afc2000c  sw      v0,12(s8)               |  a0: 24020020  li      v0,32                   |
```

Figure 21: Compilation example.

6.2 Known issues with compiler optimizations

The operation of the moving threads architecture is somewhat incompatible with the standard MIPS32 code. The first difference is that the instruction set is only a simple subset of all MIPS32 instructions. The compiler needs to work conservatively with optimizations to avoid emitting incompatible opcodes. The second issue is with delay slots. The moving threads architecture does not support delay slots and the feature has to be switched off with `-fno-delayed-branch`. Unfortunately some general optimizations such as `-O3` turn this on automatically.

The list of architecture flags set by the frontend script is shown in Figure 23 and the list of optimizations and deoptimizations respectively in Figure 22.

```
-funswitch-loops -foptimize-sibling-calls -fsee
-fforward-propagate -fmerge-constants -fregmove
-fmerge-all-constants -fmodulo-sched-allow-regmoves
-fthread-jumps -fmodulo-sched -fsplit-wide-types
-fcse-follow-jumps -fcse-skip-blocks -fgcse-las
-fgcse -fgcse-lm -fgcse-sm -frerun-cse-after-loop
-fgcse-after-reload -funsafe-loop-optimizations
-fcrossjumping -fauto-inc-dec -fdce -fdse
-fif-conversion -fif-conversion2 -fschedule-insns
-fdelete-null-pointer-checks -fcaller-saves
-fexpensive-optimizations -ftree-reassoc
-ftree-pre -ftree-fre -ftree-copy-prop -ftree-salias
-fipa-pure-const -fipa-reference -fipa-pta
-fipa-cp -fipa-matrix-reorg -ftree-sink -ftree-ccp
-ftree-sra -ftree-store-ccp -ftree-ch -ftree-loop-im
-ftree-dominator-opts -ftree-dse -ftree-dce
-ftree-loop-optimize -fwhole-program
-ftree-loop-ivcanon -fivopts -ftree-copyrename
-ftree-ter -ftree-vectorize -ftree-vect-loop-version
-ftree-vrp -ftracer -fvariable-expansion-in-unroller
-fpredictive-commoning -fprefetch-loop-arrays
-fpeephole -fpeephole2 -freorder-blocks
-freorder-blocks-and-partition -freorder-functions
-fstrict-aliasing -fstrict-overflow -falign-loops
-falign-functions -falign-labels -falign-jumps
-fcprop-registers -fguess-branch-probability
-fmove-loop-invariants -fdefer-pop -funit-at-a-time
-fomit-frame-pointer -finline-small-functions
-finline-functions -Os -fno-delayed-branch
```

Figure 22: List of optimization flags set by the frontend script.

```
-mabi=32 -mips32 -mno-shared -mno-llsc -mno-dsp
-mno-fused-madd -mno-check-zero-division -mno-mt
-mno-mips3d -mno-mdmx -mno-paired-single -mno-dspr2
```

Figure 23: List of architecture flags set by the frontend script.

6.3 Issues left for future development

Since the standard C compiler is not well suited for multi-threaded programming on our architecture, some problematic artefacts and bugs produce inefficient and confusing binaries. For example standalone thread function implementations have bugs with backing up registers to the stack (e.g. `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43700`). This also affects inlined functions, which have no need for stack. The GCC toolchain also does not seem to support the fact and instruction and data memory can overlap without any conflicts.

The thread local state could be better used if the MIPS calling convention used by the GCC was replaced with something more appropriate to save registers and/or to prevent unnecessary use of stack. The thread local storage also does not support any looping constructs or memory mapping. Each register has to be explicitly controlled by the user code. A higher level custom language or metaprgoramming library would eliminate this problem. The language could also support other higher level parallel programming idioms such as skeleton libraries and also provide a set of parallel (and perhaps cache-oblivious) algorithms.

Our runtime system does not implement any kind of dynamic memory management in form of stack or heap. We cannot really build any big applications without dynamic memory management.

6.4 Simulating the applications with SMASim

The simulation of applications on the moving threads architecture is described in the technical report *SMASim manual 1.0* [12], which can be obtained from `http://tucs.fi/publications/insight.php?id=tMaPaLe10a`.

7 Application examples

7.1 Sum of matrices

This example demonstrates the proper use of our platform using the functionality provided by the library `moth.h`. In this program (Table 3) a naive algorithm for computing the sum of two matrices of the same size, 16x32, is shown.

The example begins with the definition of a matrix structure. We then initialize the input matrices M and N, and the result matrix O. The matrix elements are summed in a two-dimensional loop done with nested `pardo` statements. The dimensions are given as bounds to `pardo`s and an index variable is provided by the mechanism. These indexes are then used to access the correct element in the three matrices. Since the pardo is a macro in C, the preprocessed and unrolled output is shown as a comment below the code. The resulting disassembler output is given in the right column.

#include "moth.h"	**li** v1,16
#define m 16 *// dimensions*	fork v1 *# pardo #1*
#define n 32	forkl
	move v1,k1
struct matrix { **int** _[m][n]; }; *// definition*	**li** a0,32
	fork a0 *# pardo #2*
int main(**void**) {	forkl
struct matrix M,N,O; *// inputs, result*	**sll** v0,v1,0x5
	addu v1,v0,v1
pardo(i, m,	**lui** a0,0x0
pardo(j, n,	**lui** v0,0x0
M._[i][j] = N._[i][j] + O._[i][j];	**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*
fork(16);	**lw** v0,0(v0) *# load C's element*
int i = moth_thread_id();	**lui** a1,0x0
fork(32);	**addiu** a1,a1,4232
int j = moth_thread_id();	**addu** v1,v1,a1
M._[i][j] = N._[i][j] + O._[i][j];	**addu** v0,a0,v0 *# matrix A index*
join();	**sw** v0,0(v1) *# store A's element*
join();	join
*/	joinc *# join #1 (implicit)*
	join
	joinc *# join #2 (implicit)*

Table 3: Program code of the matrix sum in C and MIPS assembly.

References

[1] M. Forsell and V. Leppänen. Supporting Concurrent Memory Access and Multioperations in Moving Threads CMPs. In *Proceedings of PDPTA 2010*, pages 377–383, 2010.

[2] M. Forsell and V. Leppänen. Moving Threads Processor Architecture. *Journal of Supercomputing*, page To appear, 2011.

[3] Martti Forsell and Ville Leppänen. MTPA - A Processor Architecture for MP-SOCs Employing the Moving Threads Paradigm. In *PDPTA*, pages 198–204, 2009.

[4] 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.

[5] J.L. Hennessy, D.A. Patterson, D. Goldberg, and K. Asanovic. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[6] ISO. ISO C Standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[7] V. Leppänen. *Studies on the Realization of PRAM*. PhD thesis, University of Turku, TUCS, Lemminkaisenkatu 14, FIN-20520 Turku, Finland, nov 1996. TUCS Dissertions No 3.

[8] V. Leppänen and M. Penttonen. Sparse networks: Balance of processing and communication. 2008.

[9] J.M. Mäkelä and V. Leppänen. Simulation platform for the moving threads architecture. Technical report, 2010. `www.tucs.fi`.

[10] J.M. Mäkelä and V. Leppänen. SMASim: A Cycle-accurate Scalable Multicore Architecture Simulator. In *Proceedings of the World Congress on Engineering*, volume 1, 2010.

[11] J.M. Mäkelä and V. Leppänen. Towards programming on the moving threads architecture. In *CompSysTech '10: Proceedings of the International Conference on Computer Systems and Technologies*, pages 137–142. ACM Press, 2010.

[12] J.M. Mäkelä, J. Paakkulainen, and V. Leppänen. Smasim manual, version 1.0. Technical Report 972, April 2010. `www.tucs.fi`.

[13] 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*, pages 1–6. ACM Press, 2009.

[14] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8:25–33, October 1980.

[15] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[16] MIPS Technologies. *MIPS32® Architecture for Programmers Volume II: The MIPS32® Instruction Set*. 2001.

Turku

Centre *for*

Computer

Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2553-6
ISSN 1239-1891