International Journal of High Speed Computing, Vol. 7, No. 3 (1995) 465–480
© World Scientific Publishing Company

MULTITHREADED DECOUPLED ARCHITECTURE

MIKHAIL N. DOROJEVETS
Parallel Systems Laboratory
Institute of Informatics Systems SO RAN
6 Lavrentiev Pr., Novosibirsk 630090 Russia
E-mail: *midor@isi.itfs.nsk.su*

VOJIN G. OKLOBDZIJA
Advanced Computer Systems Engineering Laboratory
Department of Electrical and Computer Engineering
University of California, Davis, CA 95616, USA
E-mail: *vojin@ece.ucdavis.edu*

Received July 5, 1994

ABSTRACT
A new computer architecture called the Multithreaded Decoupled Architecture has been proposed for exploiting fine-grain parallelism. It develops further some of the ideas of parallel processing implemented in the Russian MARS-M computer in the 1980s. The MTD architecture aims at enhancing both total machine throughput and a single thread performance. To achieve this goal, we propose a two-level parallel computation model. Its low level defines the decoupled parallel execution of instructions within program fragments not containing branches. We will be referring to these fragments as basic blocks. The model's high level defines the parallel execution of multiple basic blocks representing a function or procedure. This scheduling hierarchy reflects the MTD storage hierarchy. Together the scheduling and storage models allow a processor with multiple execution units to exploit several forms of parallelism within a procedure. The compiler provides the hardware with thread register usage masks to allow run-time enforcing of control and data dependencies between the high level threads. We present a possible implementation of the MTD-processor with multiple execution units and two-level distributed register memory.

Keywords: Fine-grain parallelism, superscalar, multithreading, decoupled architecture.

1. Introduction. In order to exploit instruction level parallelism, superscalar and VLIW computers are capable of issuing multiple instructions per cycle to multiple functional units that can operate concurrently. New technology provides ways to increase the number of functional units and registers within the chip. The question is how much of the potential parallelism existing in a program can be really exploited by superscalar computers with multiple functional units, and how efficiently hazards and contentions can be resolved.

The exploitation of parallelism within basic blocks by hardware began almost thirty years ago. Two algorithms currently used to solve the problem of data dependencies between multiple instructions were originally developed in the 1960's: Tomasulo's algorithm [9] and Thornton's algorithm [10]. The Thornton's algorithm, also known as "scoreboarding", was used again in Intel's and Motorola's RISC processors, the I960CA and MC88110, respectively. Recently a register-renaming algorithm (which essentially supersedes Tomasulo's one) has been developed and implemented in the IBM RS/6000 superscalar processor [11].

The effectiveness of all the algorithms is limited by the complexity of the control hardware that handles contention for data between instructions. The question is whether such hardware solutions can adequately cope with the increase in the number of execution units and registers in next-generation superscalar computers. Taking into consideration the small amount of parallelism within basic blocks, superscalar architectures with VLIW ones crossed block boundaries and used a speculative execution concept in order to exploit the higher-level, inter-block parallelism. This paper describes a new architecture for exploiting fine-grain parallelism in sequential programs. This Multithreaded Decoupled (MTD) architecture allows multiple threads and multiple operations within threads to proceed in parallel in a processor with multiple execution units and distributed register memory.

In Section 2 we describe the main features of the MTD execution model. Section 3 makes comparison to related work. A possible implementation of the MTD architecture is presented in Section 4. Concluding remarks are made in Section 5.

2. Model of execution.

The multithreaded decoupled architecture is based on a two-level parallel computation model. Its low level defines the decoupled parallel execution of instructions within program fragments not containing branches. We will be referring to these fragments as basic blocks. The model's high level defines the parallel execution of multiple basic blocks representing a function or procedure. This scheduling hierarchy reflects the MTD storage hierarchy. Together the scheduling and storage models allow a processor with multiple execution units to exploit several forms of parallelism within a procedure.

(a) Low level

From the programmer's point of view, a basic block is a sequence of instructions which execute sequentially from the beginning to end without branching. The low level scheduling model represents a basic block as a partially ordered graph of subblocks, where a subblock is a fully ordered sequence of instructions to be executed by a separate unit. For each unit, the order of instructions in the unit's subblock strictly corresponds to the one in which the unit's instructions occur in the logical, non-split basic block. The splitting of a basic block in multiple instruction subblocks occurs either at the compile or run time, depending on the implementation of the MTD model.

We will be referring to the execution of these unit's specific instruction streams by separate execution units as USI-threads. Also, the set of USI-threads representing the execution of a basic block is referred to as a BB-thread.

Each execution unit is a simple state machine that can work independently of other units. The MTD decoupled model of execution allows multiple USI-threads to proceed in parallel at their own rate depending on the availability of their instructions' operands.

From the issue/decode logic's point of view, any instruction within a basic block can work with register data of two types: local (USI-thread level) and global (BB-thread level). The local data are the results of execution of the basic block's instructions. A local memory distributed over the execution units keeps these local data accessible to all instructions which need them until the BB-thread terminates. The global data are ones computed by other BB-threads preceding to the current one. If a value generated by any USI-thread's instruction is expected to be used outside the basic block, it is to be written in a common register file as well. Once written in the common register file, the global data can live until the execution of the procedure completes.

The MTD model assumes direct communication between all USI-threads within the same BB-thread. For any USI-thread's instruction, the corresponding unit's part of the local memory always has a place to hold the instruction's result. Its address corresponds to the instruction number within the USI-thread. (No instruction contains an explicit destination address field specifying where its outcome is to be written in the local memory.) In further, this locally-generated value can be addressed by a combination of the number of the unit executing the USI-thread and the producer-instruction number within the USI-thread. Since this information is known at the compile time, the compiler generates the correct operand access code of any instruction which uses the local data as its input operand.

In the MTD architecture, to determine whether a local value has been computed means to check whether the corresponding instruction has been completed. Since instructions are to be executed in a sequential order without branching within any USI-thread, this check can be done easily because a number of instructions completed within each unit is always known to all units. After completing an instruction, each unit broadcasts a one-bit completion message to all units including itself. The issue/decode logic of each unit uses a set of completion counters (one per USI-thread) to keep the up-to-date information about a number of instructions executed by each unit. Before issuing an instruction that uses local data as its input operand, the issue/decode logic compares the producer-instruction number taken from its operand access field with the corresponding count. Only if the count is less then the number, the issuing of this and other instructions of the USI-thread within the unit is to be stopped until the count reaches the number.

The USI-level scheduling model allows multiple instructions to be issued each cycle, while keeping the issue/decode hardware quite simple.

(b) High level

The MTD model defines the execution of a procedure as a partially ordered tree of BB-threads, where each BB-thread is a partially ordered graph of USI-threads. (Below by "threads" we mean "BB-threads.")

In the model, classic branch operations have been replaced with thread-based control operations like Fork, Switch, and Stop. Any thread can initiate eager execution of other basic blocks, i.e., create child threads having data and control dependencies with the parent thread through the common register file and main memory. These child threads can begin their execution even though not all of their input operands (including control ones) have been computed yet. The model allows speculative execution of threads, treating conditional values as operands of yet another type.

The MTD processor has a multiple-context architecture, one context per active thread. A thread context includes low level activation frames in the local data and instruction memories (plus a set of the completion counters) within each execution unit, high-level frames in the common register (CR) and condition code register (CCR) files, and one thread status register (TSR). The CR and CCR hardware allocates registers for the contexts in a way that allows the threads to communicate in the MTD activation tree. A TSR holds a thread status block containing the thread's entry and several bit masks which specify the thread's input and output dependencies. We will describe these masks later in detail.

After loading a thread status block, the thread can start its execution without having to wait for the calculation of its input condition value. When/if the condition value in the thread's input CCR turns out to be false, the hardware will cancel the thread.

While executing, threads share a common set of the execution units. In each cycle, every unit's issue/decode logic can deal with multiple USI-threads working in different contexts. The logic checks the availability of operands of the threads' current instructions and then issues one of the instructions whose operands are available. A multi-instruction word issued to the units in each cycle is a run-time composition of instructions of one or several threads.

The task of resolving data dependencies for such a parallel computation model as the MTD one is a real challenge to the processor architecture using the common register file for inter-thread communication. The several known-to-date hardware algorithms [9-11] cannot be of help since they issue instructions to execution units and allocate registers to hold their results in a strict program order. For example, they cannot deal with the situation when the decode logic encounters the consumer-instruction's (e.g. with R1 as its input operand), while the producer-instruction's (with R1 as its destination register) which logically precedes to the consumer-one's has not been issued yet. Simple solutions like providing each common register with one full/empty status bit will not be working because the MTD model allows multiple threads in the dynamic activation tree to communicate via the same common registers simultaneously. Besides these flow dependencies, there can be anti- and output inter-thread data dependencies to be resolved correctly as well.

Enforcing data dependencies in the MTD architecture is achieved by joint efforts of the compiler and hardware. The model assumes that any architected (i.e., visible to the compiler) register in the CR and CCR files may have as many different live instances (versions) as a number of contexts implemented in the MTD processor. The hardware maps architected registers onto physical ones via map tables (MTs) at run time. Each thread context has two MTs, one for the CR and one for the CCR files. Each physical register has an associated full/empty bit. An actual number of physical registers is implementation dependent.

A thread has two pairs of source and destination bit masks (one bit per register) specifying which architected registers in the common and condition code register files are to be used as its input and output operands. The goal of introducing these masks is to provide the register renaming hardware with the compiler's information specifying the thread register usage pattern. After loading a TSR and before initiating the thread, the hardware allocates free physical registers (with OFF-values of their full/empty bits) to the architected ones specified in the destination masks. Simultaneously, the hardware fetches from the parent thread's MT the numbers of the physical registers which hold the current values of the architected ones specified in the source masks.

For each architected register, there is a MT word containing two fields: one for reading and one for writing into this register. The read field points out the physical register from which the current value of the architected register can be fetched after writing the value by one of the preceding threads. The write field specifies the physical register pre-allocated by the renaming hardware to hold a new value to be computed by the current thread.

Simultaneously with writing the new value, the unit hardware sets ON in the physical register's full/empty bit.

The hardware uses source and destination masks to calculate a reference count of any physical register and to perform garbage collection in the register files as well. Also, the hardware keeps the information about register usage by any active thread to restore the correct state of computation when a thread is cancelled. There are two global MTs (one for the CR and one for the CCR files), reflecting the state of computation of the topmost (root) thread in the dynamic activation tree. A reorder buffer updates these global MTs in the logical order in which basic blocks occur in the program. Whenever an exception occurs, the hardware is able to specify the thread's instruction which caused the exception and to restore the state of computation at the point preceding the initiation of the thread.

To decrease the overhead of applying the MTD thread control to small basic blocks, the MTD architecture uses a basic block enlargement technique similar to the El'brus-3's one [3]. The compiler combines two such blocks in a way to allow both possible paths of computation to be executed in parallel within one thread. Then, after calculating the condition value, a merge operation uses the value as its predicate to select the true path's results. This technique also allows to remove all branches within a loop body. After eliminating internal control dependencies, as many iterations as a number of contexts implemented in the MTD processor can run in parallel, while resolving inter-iteration dependencies at run rime. In the mean time, all the iterations can share the only copy of the loop body code loaded in the low level local instruction memory, while having separate thread instruction pointers in each unit.

2.1. An example of applying the MTD approach to a program fragment. To illustrate the main features of the approach, let us consider an example of exploiting parallelism within a program fragment consisting of three basic blocks: the parent BB0 and two alternatives, BB1 and BB2. Depending on the condition value produced by BB0 (and loaded into the control register CCR1), the results of only one of the blocks (BB1 or BB2) is to be taken into account.

The dynamic data-flow graph of the fragment is shown in Fig. 1(a). Suppose that a MTD-processor contains four execution units, U1-U4, eight common registers R0-R7, and two control registers CCR0-CCR1. Assume both U1 and U2 have a unit delay of one cycle to fetch operands, perform operations, and then write the results in the common registers or their local data buffers, U3 has a 3 cycle delay, and U4 has a 4 cycle delay. (Delays due to fetching of instructions are ignored.) Further, suppose that thread status registers (not shown here) are loaded by U1. The numbers of instructions (I1-I12) are shown within the corresponding nodes of the graph. Suppose the execution of Thread 0 is to be committed if the CCR0's value is 1, and Thread 1 and Thread 2 are to be committed if CCR1=1 and CCR1=0 respectively. For the example presented, Fig. 1(b) shows the threads' instructions that are generated by the compiler and then loaded by hardware into instruction buffers of each of the units. Figure 1(c) shows the source and destination masks to be formed by the compiler for each of the threads, and Fig. 1(d) shows a cycle-by-cycle diagram of issuing instructions to the units by the threads. (We supposed that Thread 0 has the highest and Thread 2 the lowest scheduling priorities.)

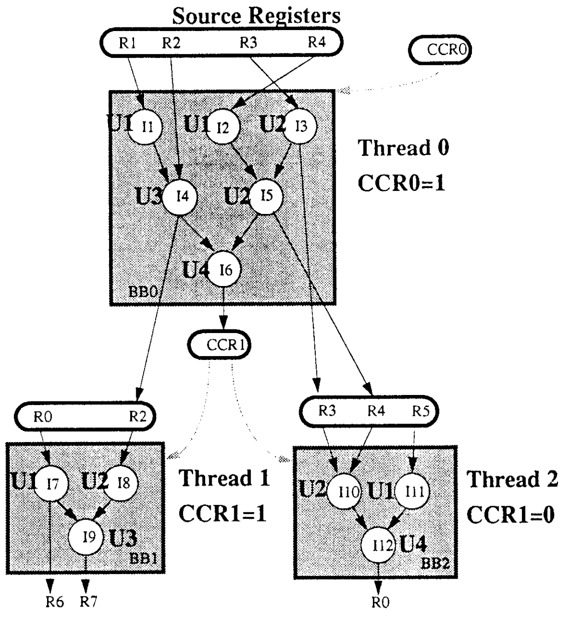

FIG. 1(a). *Dynamic data-flow graph.*

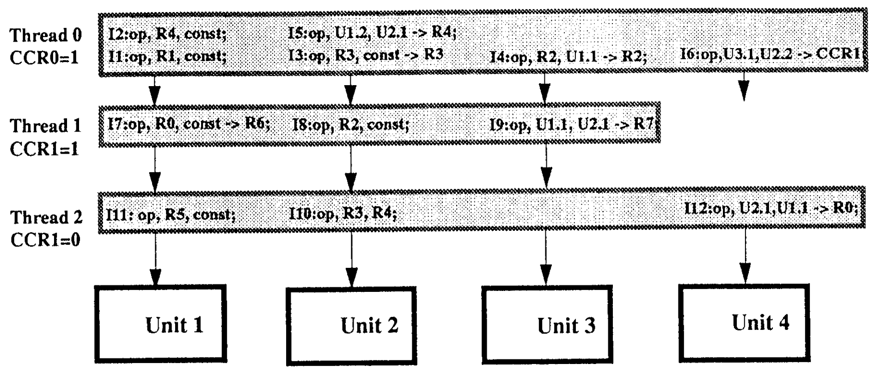

FIG. 1(b). *Distribution of instructions among the units.*

		General data registers								*Control registers*	
		R0	R1	R2	R3	R4	R5	R6	R7	CCR0	CCR1
Thread 0:	SR-mask	0	1	1	1	1	0	0	0		
	DR-mask	0	0	1	1	1	0	0	0		
	SCCR-mask									1	0
	DCCR-mask									0	1
Thread 1:	SR-mask	1	0	1	0	0	0	0	0		
	DR-mask	0	0	0	0	0	0	1	1		
	SCCR-mask									0	1
	DCCR-mask									0	0
Thread 2:	SR-mask	0	0	0	1	1	1	0	0		
	DR-mask	1	0	0	0	0	0	0	0		
	SCCR-mask									0	1
	DCCR-mask									0	0

SR-mask is a 8-bit source data registers mask,
DR-mask is a 8-bit destination data registers mask,
SCCR-mask is a 2-bit source control registers mask,
DCCR-mask is a 2-bit destination control registers mask.

FIG. 1(c). *The source and destination register masks.*

Cycles	1	2	3	4	5	6
Unit 1:	I1	I2	I7	I11		
Unit 2:	I3		I5	I8	I10	
Unit 3:		I4			I9	
Unit 4:				I6		I12

FIG. 1(d). *The instruction issue diagram.*

3. Related work. A set of ideas proposed and implemented earlier in in other architectures have influenced the MTD model of execution. We point out the most significant of the architectures: the data-flow, VLIW, super-scalar, and multithreaded architecture.

3.1. The data-flow model. A pure data-flow paradigm examines the entire program graph to find the nodes having all required operands. All such nodes can be executed in parallel. The model also assumes direct communication of all program nodes. In static data-flow architectures, however, code replication is necessary for such program entities as loops and function bodies. In addition, there are other problems concerning the implementation of actions that are sequential in nature.

In the MTD case, only a part of the program consisting of several basic blocks will be considered at run time. The data-flow mechanism is used only for passing values between basic blocks (i.e., only at the inter-thread level). The MTD model also includes sequentiality in issuing instructions within each USI-thread. The model employs the data-flow idea of direct communication of nodes within basic blocks, while avoiding code replication for loops and functions.

3.2. VLIW-architectures. This approach relies upon the compiler to resolve the problem of enforcing control dependencies in order to enable multiple operations from different basic blocks to proceed in parallel. There are several well-known approaches implemented in real VLIW-computers: trace scheduling, directed dataflow, and basic block enlargement implemented in Multiflow Trace, Cydrome Cydra 5, and Russian El'brus-3 computers, respectively [1–3]. In trace scheduling, the compiler selects the most likely path and provides compensation code to restore program correctness if some predictions were wrong. The directed dataflow approach enlarges each basic block with operations from other blocks, while equipping all the operations with predicate operands. The hardware issues only those operations whose predicate values are true at run time.

The El'brus-3 approach enables both possible paths to be executed in parallel, while expanding them when necessary with operations from succeeding basic blocks. To preserve program correctness in this case, conditional stores as well as merge operations are provided. However, VLIW architectures find it very difficult to schedule statically the events that are essentially dynamic in their nature, like interactions of a processor with dynamic memory, interrupt handling, etc.

We consider the VLIW paradigm a radical attempt to address the question of how the compiler can help the hardware in resolving data and control dependencies in a program. In the MTD model, there is no static instruction scheduling, although the compiler uses the basic block enlargement of small basic blocks. The compiler provides the hardware with information which allows the hardware to enforce data and control dependencies at run time.

3.3. Superscalar architectures. Superscalar architectures which use hardware to identify parallelism at run time enable multiple instructions to be issued simultaneously [4–8]. To exploit the fine grain parallelism and support out-of-the order execution within basic blocks, several powerful techniques have been developed and implemented [9–11]. However, the further expansion of these and other dynamic techniques on processors with a larger number of execution units requires much more complex logic.

In most cases, the superscalar computers use centralized multi-ported register files. Supporting simultaneous access to the centralized register file from multiple execution units is a real challenge for future superscalar architectures.

To address both the questions of the logic complexity and register file bandwidth, the MTD architecture decentralizes both instruction issue logic and register memory, distributing them between execution units. At the compile or run time, the whole computation flow is split into multiple USI-threads (streams below) to be processed by execution units independently. A centralized control unit typical for superscalar computers is replaced with multiple, simpler decode/issue control units, each dealing with one of the streams only. Meanwhile, a program ordering of instructions within each stream is preserved. Instead of the centralized register file, a two-level register memory is used. The low level consists of multiple local data buffers, each of which is tightly-coupled with its execution unit. The high level is a common register file for inter-thread communication.

3.4. Multithreaded architectures. The technique of multithreading (sometimes called virtual multiprocessing) has been used in several architectures to date. To hide long memory latencies, some architectures such as the CDC 6600 peripheral processors [10], HEP [12], MASA [13], Horizon [14], Tera [15], and P-RISC [16], using simple multithreading to switch between threads in every cycle. Other architectures such as the MARS-M control processor [17] and APRIL [19] switch to another thread only when a long-latency memory access occurs. Our approach grows primarily from the MARS-M architecture [18,3]. The MARS-M approach looks the most aggressive among other multithreaded or multiple context processor architectures, such as [20–26] that have been proposed to enhance machine throughput via multithreading.

First, the MARS-M uses decoupling and multithreading to tolerate memory delays and increase machine throughput, allowing multiple threads to run simultaneously within the MARS-M's VLIW processors. At the same time, these threads share a common set of address and execution units on a cycle-by-cycle basis. Second, the MARS-M multithreading is merged with multiple-instruction issuing and pipelining within threads in a way that enables machine throughput and single thread performance to be enhanced simultaneously. Third, in the MARS-M, several threads can run at the full speed of pipelines available while issuing multiple operations each, and thread switching does not result in any bubbles in the pipelines. (Another attractive feature is that the MARS-M architecture was implemented in a real computer.)

However, the MARS-M architecture has several weaknesses. First, control dependencies between fragments are obstacles to parallel execution of the fragments (i.e., speculative execution is not implemented). Second, at the first stage of the project, each fragment represented a complex memory access or execution operation. Before starting a program, each program fragment written in the MARS-M assembler was to be scheduled statically by the compiler and then loaded in the distributed instruction memory. The MARS-M goal of expanding the approach to programs written in conventional high-level languages has not been implemented. Third, the MARS-M used an expensive crossbar switch to transfer data between instruction and multiple hardware queues for inter-thread communication.

Two MARS-M ideas have influenced the MTD architecture: 1) the partitioning of a whole program into a set of different type fragments having dependencies that can be resolved at run time. This is being done in the process of overlapped execution of the fragments by multiple thread processors sharing a common set of functional units and 2) representing of the execution of each fragment as a set of a unit's specific instruction streams to be executed by separate functional units.

The ESW model's ideas of splitting a large instruction window into smaller windows and using special masks to point out data dependencies between threads [26] are similar to the MTD ideas concerning the first level of thread scheduling. In contrast to ESW-threads, MTD threads are simpler because there are no control dependencies within them. This restriction, however, makes possible for the MTD architecture to enhance a single thread performance via decoupling and multithreading (i.e., to address the question that the ESW model pays a little attention to). Another difference lies in the method of communication between threads. Similar to Hirata's architecture [23], ESW proposes the use of forwarding queues to provide communication between logically-adjacent iterations (threads) only.

4. Implementation of the MTD architecture. The basic organization of the MTD processor is shown in Fig. 2. Logically the processor consists of four independent logical processors (called slots) sharing one common data path, which consists of multiple functional units, buses, and caches. Each slot executes threads in parallel with other slots, receiving and passing values (when necessary) to/from other slots via floating-point, fixed-point, and condition code registers (FPRs, FXRs, and CCRs, respectively).

Slot control logic is distributed between identical control modules (CM), each of which deals with a single functional unit. To provide independent execution of four threads, there are four sets of instruction (IB) and data (DB) buffers and registers within each of the control modules.

When a thread management unit instructs a slot to execute a thread, the unit sends the request to the instruction cache unit (ICU) to fetch the thread's code. The instruction cache units fetches the code into its instruction cache and then transfers up to four 32-bit instructions per cycle along the four-word instruction bus to the control modules. Each instruction has a unit field specifying the unit where the instruction is to be executed. The

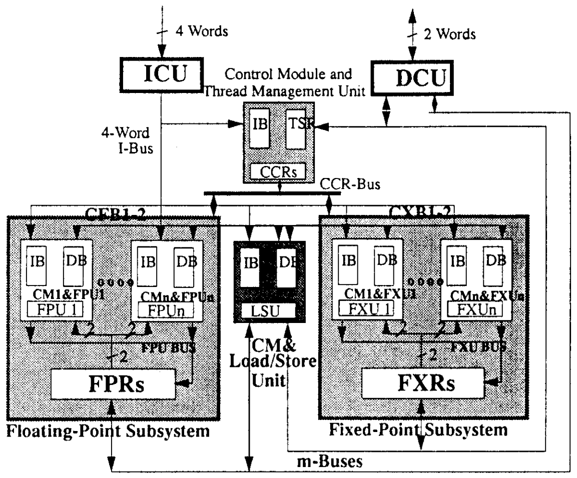

FIG. 2. *Basic organization of the MTD processor.*

field, together with the slot-number assigned to the thread by the thread unit, forms the exact address of the instruction buffer (IB) where the instruction is to be written. Instruction buffers within each of the control modules are interleaved and are capable of writing up to four instructions per cycle. When all or some of the slots are assigned to execute different iterations of the same loop, no replication of the loop's instruction code is needed. The code is shared by the threads.

The issue logic of each of the slots within a control module fetches instructions from the instruction buffers, checks the availability of their operands within the module's data buffers, and when necessary requests operands located in other control modules. Communication buses (CFB1 and CFB2, CXB1 and CXB2, for the floating-point and fixed-point subsystems, respectively) are provided to transfer data between control modules.

When the operations' results are used exclusively within the thread, they are written in the module's destination buffer only. However, when the results are input operands for other threads, it is necessary to write them into floating or fixed-point registers as well, using the common FPU- or FXU-buses. The compiler provides a Write Register bit for each instruction to control such writes. For any of the registers, including the CCRs, a single assignment rule is used within any thread. The storing of all intermediate (internal consumption only) values within the registers is eliminated due to the use of destination buffers for these purposes.

Besides a destination buffer, each control module contains an external data buffer (EXB) which holds the values computed by other units that are to be used by more than one of the instructions within the control module. Since there are no branches within a thread, the compiler knows precisely how many times and where (by which units), each value is to be used within the thread. Special write-bits also exist for each input operand field within an instruction to store such multiply used operands within the module's EXB. A FIFO-discipline is used for writing into, and a random access one for reading from the external data buffers.

For any instruction requiring the operand(s) that have been previously loaded in the external data buffer of the same module, the compiler (since it knows how many operands have been written into each buffer at each moment) provides the address of the operand(s) within the buffer. The values to be written/read to/from destination buffers are addressed by the number of the instructions that produced the values.

There are 16 floating-point, 16 fixed-point, and 4 condition code architectured registers organized into corresponding register files common for all slots. Up to two registers can be read and one written from/to the register files in each cycle. Because of the speculative execution of four threads in parallel, there are output data dependencies between the threads aliased to the same architectured registers. To resolve these dependencies, an extended register-renaming mechanism is used, which enables multiple physical registers assigned to a thread to be released when the thread is squashed.

The data cache unit (DCU) transfers data to/from the floating-point and fixed-point register files and the load store unit along two M-buses. However, memory data dependencies are a much more serious challenge to the architecture than register data dependencies, since a large size of main memory and run-time calculation of memory addresses makes it impossible to use a memory-renaming mechanism based on memory usage masks.

There are four main issues to be addressed:

(a) keeping the proper state of memory for speculative execution of four threads in parallel;
(b) enforcing memory data dependencies both within a slot and across slots;
(c) forwarding memory values across slots (or chaining load and store operations);
(d) recovering the proper state when forwarding/loading of incorrect values has been identified.

The load/store unit together with the thread management unit provide the execution of the four tasks at run time.

In performing load operations, the load/store unit considers the data fetched from memory as results of the unit's operations and loads them into its destination buffer. (The data are loaded into floating- and fixed-point registers only when they are input operands for other threads too.) Thus, overlapping of load and arithmetic operations for any thread's internal values is achieved without using any register renaming mechanism.

Each store operation is committed only if the condition code value assigned to the thread turns out to be true. Conditional store operations are provided to support the basic block enlargement technique within a thread. Such store operations can be committed only when the values of their condition code operands are true (in addition to the previously mentioned requirement concerning the thread's condition value).

A special hardware mechanism is used to forward data across the slots when it is unknown whether these data will be used by other threads. In performing any load operation, the hardware compares the load address with the store addresses of the store operations that have not been committed yet. This comparison takes place only for the store operations of the threads logically preceding to the current one in the activation tree. In case of coincidence in addresses, the value from a store buffer will be transferred as a result of the load operation. In fact, the store buffer within the load/store unit operates essentially as a small cache designed to speed up the process of communication of threads via memory. Since it is known exactly how many load and store operations are within each thread, and threads are not large in size, it is possible to implement such an address-compare operation with a reasonable hardware cost. In performing any store operation, the hardware can find out that some of the load operations executed previously in the succeeding threads have used old (i.e., wrong) memory values. In this case, the thread management unit squashes the threads and starts them again.

5. Concluding Remarks. In this paper we have proposed an architecture for exploiting fine-grain parallelism through concurrent speculative execution of multiple basic blocks and decoupling operations within blocks. In the MTD architecture, the hardware resolves control and data dependencies between threads at run time, using thread register usage masks generated by the compiler. The architecture allows to implement a decentralized instruction issue mechanism for a processor with multiple execution units and exploit multiple forms of parallelism within a procedure.

What is left is to evaluate the feasibility of the MTD architecture. In particular, we are going to study the correlation between processor characteristics, such as a number of contexts, functional units, buses, and registers, on the one hand, and the level of available parallelism within programs together with cost of its exploiting in the MTD architecture, on the other hand.

REFERENCES

[1] R. P. Colwell, J. J. O'Donnel, R. D. Papworth and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In The Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II), Palo Alto, Calif., Oct. 5–8, 1987: 180–192.

[2] B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle. The Cydra 5 Departmental Supercomputer. IEEE Computer 22 (1), Jan. 1989: 12–35.

[3] M. N. Dorozhevets and P. Wolcott. The El'brus-3 and MARS-M: Recent Advances in Russian High-Performance Computing. The Journal of Supercomputing 6, 1992: 5–48.

[4] G. F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor. IBM Journal Research and Development 34 (1), 1990: 37–58.

[5] E. Edina, W. Walker, J. Yetter and M. Forsyth. A High Speed Superscalar PARISC Processor. In COMPCON SPRING '92, San Francisco, Calif., 1992: 116–121.

[6] K. Diefendorf and M. Allen. The Motorola 88110 Superscalar RISC Processor. In COMPCON SPRING '92, San Francisco, Calif., 1992: 157–162.

[7] G. Blanck and S. Krueger. The SuperSPARC Microprocessor. In COMPCON SPRING '92, San Francisco, Calif., 1992: 136–141.

[8] R. L. Sites. Alpha AXP Architecture. Communications of the ACM 36 (2), 1993: 33–44.

[9] D. W. Anderson, F. J. Sparacio and R. M. Tomasulo. The IBM 360 Model 91: Machine Philosophy and Instruction Handling. IBM Journal of Research and Development 11 (1), 1967: 8–24.

[10] J. E. Thornton. Design of a Computer — The Control Data 6600. Glenview, IL: Scott, Forersman and Co., 1970.

[11] J. Cocke, G. F. Grohoski and V. G. Oklobdzija. Instruction Control Mechanism for a Computing System with Register Renaming, MAP Table and Queues Indicating Available Registers. U.S. Patent 4 (992, 938). February, 1991.

[12] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In 1978 Int. Conf. on Parallel Processing, 1978: 6–8.

[13] R. H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In The 15th Annual Int. Symp. on Computer Architecture, June 1988: 443–451.

[14] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. In 1988 Supercomputing Conf., November 1988: 35–41.

[15] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith. The Tera Computer System. In Int. Conf. on Supercomputing, June 1990: 1–6.

[16] R. S. Nikhil and Arvind. Can Data Flow Subsume von Nuemann Computing. In The 16th Annual Int. Symp. on Computer Architecture, June 1989: 262–272.

[17] M. N. Dorojevets. The MARS-M Control Processor, Theoretical and applied problems in parallel processing, Computing Center of SD of the USSR Academy of Sciences, Novosibirsk, 1984: 150–160.

[18] Yu. L. Vishnevsky. Architectural Features of the Mini-MARS Processor, High-performance Systems for Data Array Processing, Computing Center of SD of the USSR Academy of Sciences, Novosibirsk, 1982: 5–33.

[19] A. Agarwal, B. H. Lim, D. Kranz and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In The 17th Annual Int. Symp. on Computer Architecture, 1990: 104–114.

[20] G. E. Daddis Jr and H. C. Torng. The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors. In The 20th Int. Conf. on Parallel Processing, August 1991: 76–83.

[21] R. G. Prasadh and C. Wu. A Benchmark Evaluation of a Multi-Threaded RISC Processor Architecture. In The 20th Int. Conf. on Parallel Processing, August 1991: 84–91.

[22] A. Wolfe and J. P. Shen. A Variable Instruction Stream Extension to the VLIW Architecture. In The 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems. ACM Press, April 1991: 2–14.

[23] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase and T. Nishizava. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In The 19th Annual Int. Symp. on Computer Architecture, 1992: 136–145.

[24] S. W. Keckler and W. J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In The 19th Annual Int. Symp. on Computer Architecture, 1992: 202–213.

[25] P. Lenir, R. Govindarajan and S. S. Nemawarker. Exploiting Instruction-Level Parallelism: The Multithreaded Approach. In The 25th Annual Int. Symp. on Microarchitecture, 1992: 189–192.

[26] M. Franklin and G. S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In Int. Conf. on Computer Architecture, 1992: 58–67.