A Register Communication Mechanism for Speculative Multithreading Chip Multiprocessors

Niko Demus Barli,† Daisuke Tashiro,† Chitaka Iwama,†
Shuichi Sakai† and Hidehiko Tanaka†

Speculative multithreading on chip multiprocessors has drawn great attention as a technique for exploiting thread level parallelism from sequential applications. This paper proposes a register communication mechanism required to handle inter-thread register dependencies during speculative multithreading execution. The key issues in designing this mechanism are, ensuring the correctness of execution and tolerating communication latency important to the performance. This paper first describes a synchronization method for maintaining a consistent architectural view of the registers. It then presents a design of a ring-topology communication datapath for synchronizing register values. The communication mechanism we propose tolerates communication latency by eagerly moving register values closer to the consumer and by employing a simple producer-initiated communication protocol. It also avoids substantial increase in number of ports of register file and register rename map. Evaluation results show that, for a practical configuration the average performance achieved is within 6% margin compared to an ideal datapath.

1. Introduction

Chip multiprocessor (CMP) architecture is becoming increasingly accepted as a platform for high performance microprocessors [3, 7, 15]. The CMP architecture is very suitable for multiprogrammed or multithreaded workloads generally found in servers. However, to be fully accepted as a general purpose platform, it must also deliver competitive performance for a wide range of sequential applications usually found in desktops. To exploit thread-level parallelism from these applications, speculative multithreading is proposed. In a speculative multithreading execution, the originally sequential execution stream is partitioned into chunks, executed in parallel as distinct threads. These threads may exhibit some sort of control or data dependencies and a combination of software and hardware supports is added to ensure the correctness of execution [1, 5, 6, 8, 10, 11, 14, 16].

In an execution model that allows inter-thread register dependencies, a register communication mechanism that maintains a consistent view of registers during speculative multithreading execution is required. The implementation of this mechanism is challenging. First, since the register files in CMP are physically distributed the implementation is not straightforward. Second, in a CMP that comprises dynamically scheduled superscalar cores, additional complication arises from the requirements to correctly handle register mapping during communication. Finally, the mechanism must be able to tolerate communication latency critical to the performance.

This paper proposes a solution for these problems. We specify a register synchronization method to ensure the correctness of execution. Information on which registers may be redefined inside a thread and the timing to safely communicate their values to succeeding threads is inserted into binaries using compiler assistance. Then we design a communication datapath necessary for the synchronization process. Observing that most of the inter-thread register dependencies exist between two adjacent threads, we employ a ring-topology communication datapath. The datapath is optimized to tolerate communication latency by eagerly moving generated register values closer to the consumer even when it is still uncertain whether the values are the final values needed for the communication. Here, we employ a producer-initiated communication protocol in order to simplify the communication process. Our design also avoids substantial increase in number of ports of register file and rename map. For a minimal configuration, it only requires one additional write port of register file, and one additional read port of rename map. Evaluation results show that the configuration achieves performance within a 6% margin relative to an ideal communication datapath.

The rest of this paper is organized as follows. Related work on the design of register communication mechanism is summarized in section 2. Register synchronization method and the role of compiler are described in section 3. Section 4 presents the communication datapath and discusses choices

† Graduate School of Information Science and Technology, The University of Tokyo

and policies made during the design process. The evaluation results are then presented in section 5. Finally, section 6 concludes the paper.

2. Related Work

A number of speculative multithreading architectures whose thread model allows inter-thread register dependencies have been proposed. Multiscalar [4, 14] employs a ring-topology communication controlled by six sets of masks. However, it does not describe how it synchronizes register values in the presence of register renaming without introducing significant additional latency.

CMP architecture proposed in [8] uses a scoreboarding mechanism on a shared-bus datapath. It requires only small additional area at the expense of longer communication latency when the communication is initiated by a consumer. Although the evaluations showed insignificant impact caused by this additional latency [9], it is possibly because the threads are only created at innermost loop iterations. In this case, the probability of a thread to be restarted due to control misspeculation, thus igniting a consumer-initiated communication, is low. It should also be noted that targeting only innermost loops for threads is not sufficient for integer applications. In these applications, innermost loops only account for 30% of total executed instructions [10] and the average number of iterations per execution is very low [17].

Speculative Multithreaded processor proposed in [10] employs an hardware mechanism using a form of value prediction to handle inter-thread register dependencies. Threads are also created only at loop iterations. The mechanism is tailored heavily to exploit the characteristics of loop iteration threads. Therefore, it is not applicable to thread models that include non loop-iteration threads.

Finally, Trace Processor [12], Superthreaded architecture [16], and MP98 processor [5] use a global register file for synchronizing register values. Although this approach is more straightforward, the port requirements and the latency of long wires when accessing this centralized structure are likely to limit the scalability and the performance achieved by these architectures.

3. Register Synchronization

Before describing our register synchronization method, we first explain thread execution model we used throughout this paper. In this model, a sequential program is first partitioned into threads by a compiler. We define *thread* as a connected subgraph of a control flow graph with exactly one entry point. Overlapped regions shared by two or more threads may not exist. Thread boundaries are put at function invocations, innermost loop iterations, and at remaining points necessary to satisfy the thread definition [2]. During the execution, threads are scheduled to the CMP's processing units in a round-robin fashion. A thread control unit is responsible to predict and validate the execution of the threads. Memory *load*s are speculatively executed and a mechanism to detect memory dependency violation similar to the one described in [8] is provided. In case a violation occurs, the violating thread and all of its successors are flushed and restarted. Finally, inter-thread register dependencies which is the main focus of this paper are synchronized.

3.1 Compiler Support

To relax the hardware requirements, compiler is used to identify inter-thread register dependencies and insert necessary timing information to safely synchronize values between dependent threads. Specifically, for each statically defined thread, the following information is generated by the compiler:

- *create mask* : inserted into thread headers; specifies set of registers possibly redefined by the thread; depending upon the control flow, a register in the mask may be released without modification.
- *send flags* : a 1-bit extension in instruction opcode; if set, indicates that the instruction's destination register is ready for communication when the instruction retired.
- *send instructions* : explicitly encode registers ready for communication when the instruction retired.

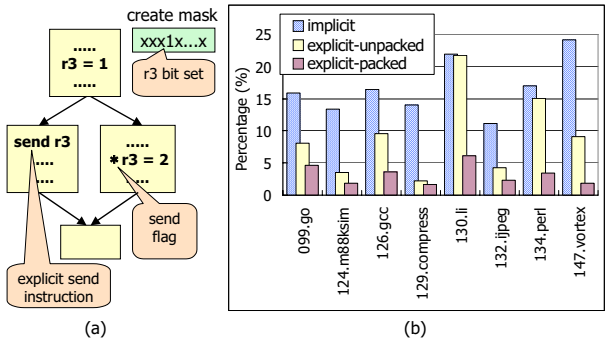

Fig. 1 Static information generation

Figure 1(a) illustrates an example on how this information is generated. For a static thread shown in the figure, we first set `r3`'s entry in the create mask since `r3` is redefined inside the thread. Then,

we identify points to safely send the value of `r3`. In case the control flow follows the right path, the safe point is when the second instruction writing to `r3` is retired. Thus, we set the instruction's send flag. On the other hand, if the control flow follows the left path, the safe point is at the beginning of the left-side basic block. Here, we need to explicitly insert a send instruction to send the value of `r3`.

Explicitly inserted send instructions may substantially increase the total number of instructions fetched. Figure 1(b) shows: the percentage of executed instructions that have their send flag set (*implicit*), the percentage of send instructions when only one register can be specified in an instruction (*explicit-unpacked*), and the percentage of send instructions when we can pack some registers into one instruction (*explicit-packed*). For *explicit-packed* we provide four new opcodes, each can pack int register 0 to 15, int register 16 to 31, fp register 0 to 15, and fp register 16 to 31, respectively.

The figure shows that more than half of the information takes the form of send flags which incur no additional overhead. However, it also suggests the necessity to encode more than one registers into one send instruction otherwise the overhead becomes significantly large. The four new opcodes we adopted helped to suppress the overhead to less than 6%. It might not be an optimal solution, but is acceptable for our research purpose and used throughout this paper.

3.2 Synchronization

Instructions whose operand value is produced by a predecessor thread must be stalled in order to maintain correct execution semantics. We accomplish this by maintaining wait and redefined masks of logical registers. *Wait mask* is created by ORing create masks of predecessor threads. Thread control unit, which maintains create masks of currently active threads, creates the wait mask at the beginning of a thread execution. If wait bit of a register in the mask is set, it indicates that the register may be produced by one or more of the predecessor threads. Consequently, the execution of instructions that use the register value should be stalled.

However, not all instructions whose operand register has its wait bit set need to be stalled. If the register is redefined by another instruction, succeeding instructions from the same thread that use the register are not necessarily be stalled. To identify which registers have been redefined inside a thread, we accommodate each thread with a *redefined mask*. At the beginning of execution, all of the bits are cleared. When an instruction is decoded the corresponding redefined bit of its destination register is set. If redefine bit of a register is set, successive instructions that use the same register as source operands, do not need to wait for the values from predecessor threads even if the wait bit of the register is set. In other words, a source register of an instruction needs to be synchronized only if the following condition, $wait(reg\_id)\ \&\ !\,redefined(reg\_id)$, is asserted.

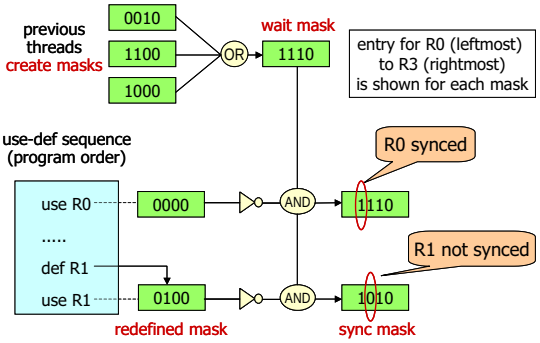

Fig. 2 Synchronization method

Figure 2 illustrates an example of synchronization process in our design. Entries for R0 to R3 are shown from left to right for each mask. Suppose the previous three predecessor threads have create masks of 0010, 1100, and 1000 respectively, then by bitwise OR-ing these masks, a wait mask of 1110 is created. This bit pattern indicates that the previous threads produced R0, R1, and R2. Suppose the first instruction in the current thread uses R0. At this point, the corresponding redefined bit of R0 is 0, thus sync bit of R0 is set to 1. Then we know that the instruction should be stalled until R0 is made available by the latest producer among the predecessor threads. Now suppose the other instruction in the current thread use R1. This time, also suppose one of preceding instructions has redefined R1. In this case, since the redefined bit of R1 was set to 1, sync bit of R1 is 0. Consequently, we do not need to synchronize R1 value for this instruction.

4. Datapath Design

In this section, we present datapath design for implementing synchronization method described in the previous section. The design includes extension of existing datapath and integration of a register communication datapath. The extension of existing datapath includes:

- Redefined mask in rename table: the status of this mask must be correctly rolled-back in case of branch mispredictions.

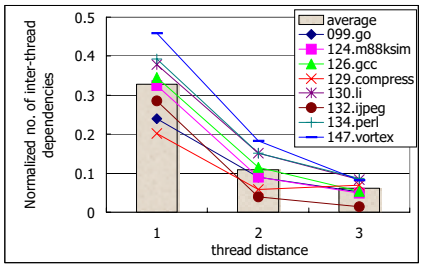

Fig. 3 Thread distance statistics of inter-thread register dependencies

- Additional bank of registers: required to recover register state to its initial value when a thread misprediction or a memory violation occurs.
- Logical id fields for source registers in reservation station: necessary to track the status of synchronized registers. When a register value is received from a predecessor thread, its logical identifier is broadcasted to the reservation station. Instructions waiting for the register clear their corresponding synchronization bits and become dispatchable.
- Physical and logical id field for destination register in reorder buffer: necessary to map back physical id to logical id when sending the register.

Although implementation issues of the above extension are not less important, due to the limitation of space, for the rest of this section we focus our discussion on the design and implementation issues of the communication datapath.

4.1 Preliminary Analysis

We performed an analysis to find thread distance characteristics of inter-thread register dependencies. The results for eight integer applications from spec95 benchmark are shown in figure 3. The number of inter-thread register dependencies in the figure is taken by counting the number of producer-consumer instruction pairs for every synchronized register. This number is normalized to the number of intra-thread register dependencies. An important characteristic to be noted here is that the largest part of inter-thread register dependencies exists between two adjacent threads (i.e. when thread distance = 1). Consequently, it is very important to provide fast communication datapath between neighboring PUs to benefit from this characteristics.

We also estimated latency and bandwidth requirements for communication datapath. For each consumer thread we assumed a datapath as shown

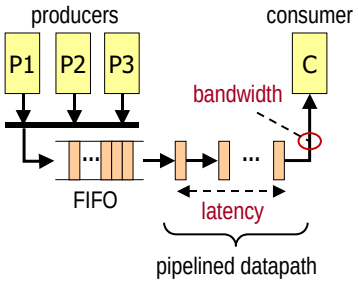

Fig. 4 Datapath used for studying latency and bandwidth requirements

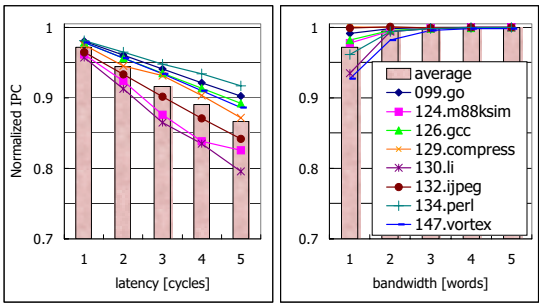

Fig. 5 Latency and bandwidth sensitivity

in figure 4. We assumed an ideal FIFO (zero latency, unlimited capacity, no insertion congestion) for the analysis. Figure 5 shows how the performance (IPC) is affected when we varied the latency and bandwidth of the datapath for a 4-PU CMP configuration. The IPC is normalized to the IPC when the latency is zero and the bandwidth is unlimited. The results indicate that the performance is particularly sensitive to the increase in communication latency. A one cycle increase in latency leads to a 2.6% decrease in performance. On the other hand, the bandwidth requirement is moderate. A bandwidth of one word per cycle achieved 97% of performance achieved when the bandwidth is unlimited.

4.2 Communication Datapath

The results of preliminary analysis suggested that it is important to provide low latency communication, especially between two adjacent threads which contribute to most of the inter-thread register dependencies. Therefore, we chose to employ a ring-topology datapath with a simple producer-initiated communication protocol. The organization of the datapath is shown in figure 6. We integrated and replicated the following datapath elements into each processing unit (PU):

- *Intermediate Register Buffer (IRB)* : indexed by physical register ID; holds speculative (not yet committed) register values produced by the PU.

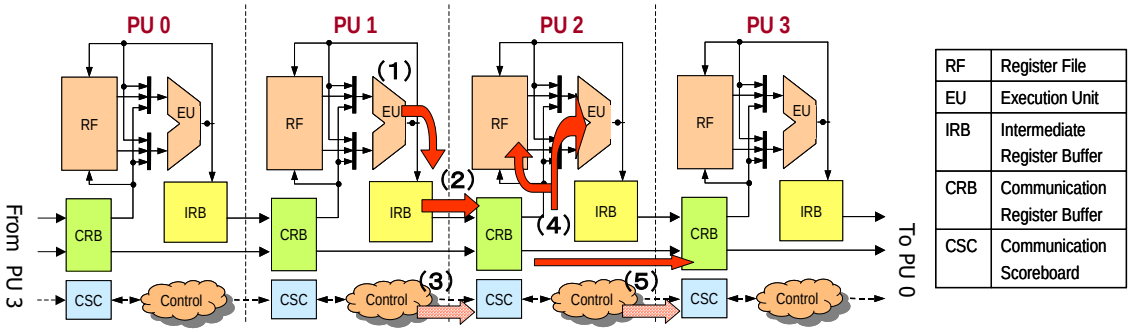

Fig. 6 A ring-topology datapath and an example of communication process

- *Communication Register Buffer (CRB)* : indexed by logical register ID; holds register values either committed or propagated by previous PU.
- *Communication Scoreboard (CSC)* : indexed by logical register ID; holds status of registers in communication.

The organization and access control of IRB and CRB are similar to renamed register file and architectural register file in split rename register file architecture [13]. IRB holds speculative register values whereas CRB holds committed register values. However, in our design IRB and CRB are used for different purpose, that is to provide buffers when register values moved closer to the consumer.

Communication control is implemented by monitoring the status of registers in Communication Scoreboard (CSC). Each entry in the scoreboard holds the following information:

- *Producer ID (PID)* : ID of a producer PU that initially generated or released the register; required to determine when to terminate propagation.
- *Ready-CRB (RC) bit* : if set, indicates the register value generated by a producer thread is ready in CRB.
- *Ready-Released (RR) bit* : if set, indicates the register value is released; A released register means that no new value is generated and the waiting instructing may proceed the execution using the locally available register value.
- *Updated (UPD) bit* : if set, indicates the register value and status has been updated to local register file.
- *Propagated (PRO) bit* : if set, indicates the register value has been processed for propagation to the next PU; Only registers whose entry in the create mask not set (i.e. not redefined inside the thread) are propagated.

Figure 6 also illustrates the communication process occurs on the datapath. Suppose PU 1 generates a register value and sends it to the rest of the PUs. The communication process can be described as follows:

(1) PU 1's Execution Unit (EU) executes an instruction and generates a value for the instruction's destination register. When the value is written back to Register File (RF), it is also written to Intermediate Register Buffer (IRB).

(2) When the instruction is retired, the generated register value is copied from the IRB to PU 2's Communication Register Buffer (CRB). Logical id stored in reorder buffer (see discussion at the beginning of section 4) is used to map back physical id to logical id during the process. Storing logical id in reorder buffer is useful to avoid accessing rename map which is already heavily ported and probably far in distance.

(3) Let us assume that the instruction has its send flag set. When it retired, we set Ready-CRB (RC) bit of the corresponding register in PU 2's Communication Scoreboard (CSC).

(4) Update logic of PU 2 is monitoring the CSC. When it sees the RC bit of the register is set, it knows that the value has been made available in its CRB. It then reads the corresponding value from CRB and updates its local register file. It also forwards the values to execution units. We define the datapath from CRB to register file and forwarding path as *update datapath*. Finally, PU 2 also sets Updated (UPD) bit of the register in its scoreboard, indicating that the local register file has been updated.

(5) PU 2 also propagates the value to PU 3. The propagation logic checks the register entry in create mask and if not set (i.e. PU 2 does not redefine the register), moves the register value

from its own CRB to PU 3's CRB. We define this datapath as *propagate datapath*. PU 2 also sets Ready-CRB (RC) bit of PU 3's CSC. Consequently, PU 3 knows the value is now available in its CRB. Finally, PU 2 sets the Propagated (PRO) bit of the register in its scoreboard.

In the above example, we did not describe the procedure to handle explicit send instructions. If the register is redefined inside a thread, the procedure is basically identical. It differs in that, it is not necessary to copy register values from IRB to CRB since the values have already resided in CRB (the values were copied when the instructions that created them retired). The control logic then only needs to set the RC bit of the register in next PU's CSC.

Complication arises when a register that initially has its create bit set, happens not to be redefined inside the thread. The register is then *released* by a send instruction. In this case, we should propagate value generated from a preceding PU, if any, or let the consumer use locally available value. This can be done as follows. If wait bit for the corresponding register is set, then we know the register may be generated by a preceding PU. In this case, we clear the register entry in create mask and let the propagation logic handles the rest. If the wait bit is not set, then we set the Ready-Released (RR) bit in the next PU's CSC. When the next PU's update logic sees RR bit is set, it sets UPD bit and broadcast the register ID to reservation so that waiting instructions may be dispatched and acquires register values from local register file. Subsequently, the propagation logic checks the PU's create mask and if not set, propagate the release status by setting RR bit of CSC in the succeeding PU.

A distinctive feature of our communication datapath is that register values are moved closer to the consumer as soon as they become available. At the time we know the values are ready to consume, they are already residing in the consumer's CRB. This technique considerably reduces latency compared to if we have to reread the value from the producer's register file. Our design also relieves pressure on number of ports of register file and rename map. In a minimal configuration, it only add one write port to the register file and one read port to rename map.

4.3 Update Pipeline

Figure 7 depicts a pipeline view of the update process. The process involves a scheduling stage, in which, the selected register ID is broadcasted to reservation station so that waiting instructions, if any, can be dispatched. Then, if RC is set, the register map is read and the register file is updated. At the same time, the register value is forwarded to execution units, so that an immediately dispatched instruction can receive the value. Assuming a minimum one cycle latency for scheduling, using this datapath, there is a two-cycle minimum latency after a producing instruction retired until a consumer can retrieve the register value.

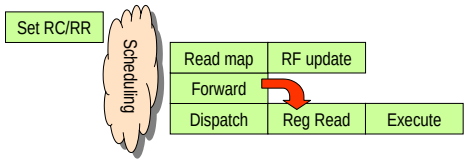

Fig. 7 Pipeline view of communication process

4.4 Misspeculation Recovery

A question may arise on how the mechanism deals with misspeculation, either initiated by thread misspeculation or by memory dependency violation. Since in case of a misspeculation, the local register file in the PU and the CRB in the succeeding PUs may have been corrupted, we must reset the status of communication and re-initiate the process of updating and propagating register values. Specifically, for each PU whose execution must be restarted, it has to clear UPD and PRO bits in its scoreboard. It also has to clear RC and RR bits in its succeeding PU's scoreboard. The resetting mechanism is however simple since it can be accomplished locally without any handshaking process among the PUs.

5. Evaluations

To evaluate the efficiency of the design, we performed simulations using a trace-based speculative multithreading CMP simulator whose configuration is shown in table 1. It simulates a 4-PU CMP with an ideal thread prediction. Each PU is a 4-issue 10-stage out-of-order superscalar core with a 32-kB L1 data and instruction cache (2-cycle latency). An infinite size L2 cache (always hit, 6-cycle latency) shared by all the PUs is assumed. Eight integer applications from spec95 benchmark are used for the simulations. The input parameters are adjusted so that the execution finishes between 100-300 million instructions.

We collected data on three communication datapath configurations shown in table 2. *static-bw1* is the most simple configuration with one cycle update latency, two cycles propagate latency, and one

Table 1 Simulation parameters

No. of Proc. Units	4 PUs
Thread Prediction	ideal
Pipeline stages	10 stages
Fetch/Decode/Rename/Retire	4 insts/cycle
No. of ALUs	2
No. of Address Units	2
No. of ROB entries	64
No. of LSQ entries	20
L1 ICache (2-way assoc.)	32 KB
L1 DCache (2-way assoc.)	32 KB
L1 cache access latency	2 cycles
L2 cache access latency	6 cycles
L2 Unified Cache	ideal

Table 2 Datapath configurations

Comm. Params	static-bw1	static-bw2	stall-first
Update Lat	1	1	1
Update BW	1	2	1
Propagate Lat	2	2	2
Propagate BW	1	1	1
Scheduling	static	static	stall-first

word per cycle update and propagate bandwidth. *static-bw2* increases the update bandwidth to two words per cycle. This implies a total of additional two write ports in register file and two read ports in rename map. Both configurations employ a statically assigned priority for scheduling update and propagation. In contrast, the third configuration, *stall-first*, employs a more sophisticated scheduling in which registers whose consumer has already waiting in the reservation station are scheduled first.

Figure 8 shows the performance achieved normalized to when the datapath is ideal. Ideal datapath here is the datapath previously shown in figure 4, with zero latency and unlimited bandwidth. For the most simple configuration (*static-bw1*), our design achieved a fairly high performance, degraded only by 6% in average compared to the ideal datapath. Looking into the applications individually, *vortex* and *li* suffer a larger impact in performance. This is mainly because the number of inter-thread register dependencies and the latency and bandwidth requirements in these two applications are substantially larger (see fig. 3 and fig. 5). Increasing update datapath bandwidth (*static-bw2*) or providing a better scheduling (*stall-first*) achieved similar amount of improvement over *static-bw1* configuration. Considering that increasing update bandwidth requires additional ports in register file and rename map, implementing a better scheduling might be a better approach.

Figure 9 shows sensitivity of performance to variation in latency and bandwidth of update and propagate datapath. The base configuration is

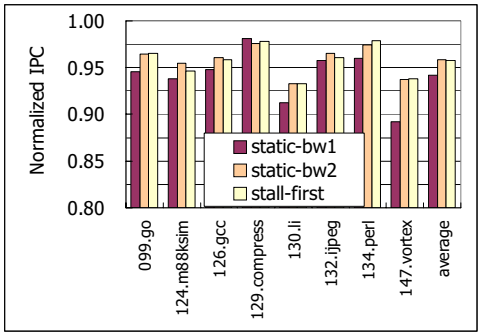

Fig. 8 Performance relative to when the communication datapath is idealized

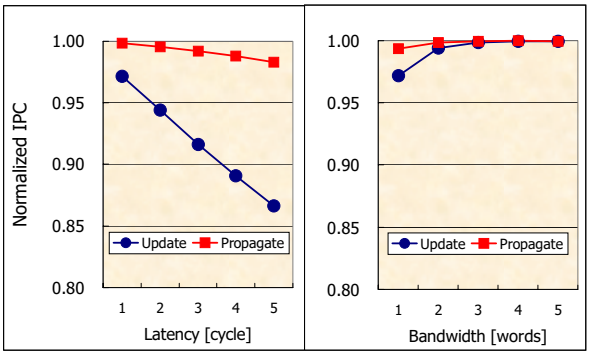

Fig. 9 Latency and bandwidth sensitivity of update and propagate datapath

identical as *static-bw1*. For each evaluation, we varied the parameter in consideration and set the other parameters to their ideal values (zero for latency and unlimited for bandwidth). The results show that the performance is more sensitive to latency and bandwidth of update datapath than to latency and bandwidth of propagate datapath. In particular, it is very sensitive to the latency of update path. These results are consistent with the analysis results previously described in section 4. Since most of inter-thread register dependencies exist between two adjacent threads, the performance is particularly sensitive to the latency of update datapath.

Finally, from the fact that the performance is insensitive to latency and bandwidth of propagate datapath, it can be implied that ring-topology is sufficient for handling inter-thread register communication. The advantages of ring network is that the communication protocol is simpler and the datapath is easier to optimize to tolerate communication latency. In our case, we can focus the design and optimization effort to reduce the latency of update datapath, and make trade-off with the propagate datapath when necessary.

6. Conclusions

This paper proposed a register communication mechanism for speculative multithreading chip multiprocessors. The elements of the mechanism are a register synchronization method and a communication datapath necessary for the synchronization process. Compiler assistance is used to relax hardware requirements for the synchronization process. The compiler identifies inter-thread register dependencies and determines timing to safely communicate register values.

Observing that most of inter-thread register dependencies exist between two adjacent threads, we designed a ring-topology communication datapath. Our design tolerates communication latency by eagerly moving register values generated by a producer closer to the consumer even when it is still uncertain whether the values are the final values needed for the communication. Furthermore, we employed a producer-initiated communication protocol to simplify the communication process and minimize communication latency. Finally, our design also avoids substantial increase in number of ports of the register file and rename map. A minimal configuration requires an additional write port to register file and an additional read port to rename map. Evaluation results showed that for the configuration, our design achieved performance degraded only by 6% compared to an ideal datapath.

Acknowledgements

This research is partially supported by Grant-in-Aid for Fundamental Scientific Research B(2) #13480077 from Ministry of Education, Culture, Sports, Science and Technology Japan, Semiconductor Technology Academic Research Center (STARC) Japan, CREST project of Japan Science and Technology Corporation, and by 21st century COE project of Japan Society for the Promotion of Science.

We are also thankful for anonymous reviewers for their constructive critics and suggestions.

References

[1] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st International Symposium on Microarchitecture*, 1998.

[2] N. D. Barli, H. Mine, S. Sakai, and H. Tanaka. A Thread Partitioning Algorithm using Structural Analysis. *ARC-2000-139*, 2000(24):37–42, 2000.

[3] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proc. of the 27th International Symposium on Computer Architecture*, pages 282–293, 2000.

[4] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. The Anatomy of the Register File in a Multiscalar Processor. In *Proc. of the 27th International Symposium on Microarchitecture*, pages 181–190, 1994.

[5] M. Edahiro, S. Matsushita, M. Yamashina, and N. Nishi. A Single-Chip Multiprocessor for Smart Terminals. *IEEE Micro*, 20(4):12–20, 2000.

[6] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of the 8th International Symposium on Architectural Support for Parallel Languages and Operating Systems*, pages 58–69, 1998.

[7] J. Kahle. Power4: A Dual-CPU Processor Chip. In *Proc. Microprocessor Forum '99*, 1999.

[8] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[9] V.Krishnan and J.Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. *International Journal of Parallel Programming*, 29(1):3–33, 2001.

[10] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative Multithreaded Processors. In *Proc. of the 12th International Conference on Supercomputing*, pages 77–84, 1998.

[11] K. Olukotun, L. Hammond, and M. Willey. Improving The Performance of Speculatively Parallel Applications on the Hydra CMP. In *Proc. of the 13th International Conference on Supercomputing*, pages 21–30, 1999.

[12] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proc. of the 29th International Symposium on Microarchitecture*, pages 24–35, 1996.

[13] D. Sima. The Design Space of Register Renaming Techniques. *IEEE Micro*, 20(5):70–83, 2000.

[14] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd International Symposium on Computer Architecture*, pages 414–425, 1995.

[15] M.Tremblay. MAJC-5200: A VLIW Convergent MPSOC. In *Proc. Microprocessor Forum '99*, 1999.

[16] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[17] J. Tubella and A. Gonzales. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, pages 14–23, 1998.