Int J Parallel Prog (2010) 38:203–224
DOI 10.1007/s10766-010-0128-3

Improving Adaptability and Per-Core Performance of Many-Core Processors Through Reconfiguration

Tameesh Suri · Aneesh Aggarwal

Received: 30 June 2009 / Accepted: 4 January 2010 / Published online: 23 January 2010

Abstract Increasing the number of cores in a multi-core processor can only be achieved by reducing the resources available in each core, and hence sacrificing the per-core performance. Furthermore, having a large number of homogeneous cores may not be effective for all the applications. For instance, threads with high instruction level parallelism will under-perform considerably in the resource-constrained cores. In this paper, we propose a core architecture that can be adapted to improve a single thread's performance or to execute multiple threads. In particular, we integrate Reconfigurable Hardware Unit (RHU) in the resource-constrained cores of a many-core processor. The RHU can be reconfigured to execute the frequently encountered instructions from a thread in order to increase the core's overall execution bandwidth, thus improving its performance. On the other hand, if the core's resources are sufficient for a thread, then the RHU can be configured to executed instructions from a different thread to increase the thread level parallelism. The RHU has low area overhead, and hence has minimal impact on scalability of the number of cores. To further limit the area overhead of this mechanism, generation of the reconfiguration bits for the RHUs of multiple cores is delegated to a single core. In this paper, we present the results for using the RHU to improve a single thread's performance. Our experiments show that the proposed architecture improves the per-core performance by an average of about 23% across a wide range of applications.

T. Suri (✉) · A. Aggarwal
Department of Electrical and Computer Engineering, State University of New York, Binghamton, NY 13902, USA
e-mail: tameesh@binghamton.edu; tsuri1@binghamton.edu

A. Aggarwal
e-mail: aneesh@binghamton.edu

Keywords Dynamic reconfiguration · Instruction level parallelism · Heterogeneous multi-core design · Per-core performance · Multi-core scalability

1 Introduction

Increasing the number of cores in multi-core processors requires sacrificing the resources in each core. For instance, in going from a 6-way issue single core superscalar processor to a quad-core processor, the issue width in each core has to be reduced to two to keep the same die area [31]. Fewer per-core resources degrades the performance of each thread of execution [16]. With increasing number of cores, the reduction in resources is also exacerbated by the increasing die area requirement for peripheral hardware such as interconnects, snoopy logic, etc [28].

Such a resource-constrained core may not be efficient for all kinds of applications. Threads with limited instruction-level parallelism may perform well on such cores. However, these cores will severely limit the performance of threads with much higher ILP. To address this problem in many-core processors, we propose a core architecture that can be adapted to suit a thread's requirement. In particular, each core is provided with an off-the-critical path reconfigurable hardware unit (RHU) in its datapath. Such cores are termed as RHU-cores. The RHU is simply reconfigured to execute instructions. The instructions can be selected from a different thread, effectively executing multiple threads in parallel, if the core resources are sufficient for the thread being executed on it. On the other hand, if the core resources fall short of a thread's requirement, the RHU is reconfigured to execute instructions for that thread to increase the overall execution bandwidth. Simultaneous Multi-Threaded (SMT) approach to achieve the same further reduces the thread's performance when executing in the SMT mode. In this paper, we focus only on reconfiguring the RHU for improving a thread's performance. *When using RHU for another thread, the overall core architecture and RHU reconfiguration process remains exactly the same; the only difference is that another context is required.*

The RHU has a small area overhead, operates entirely asynchronously with the core's datapath, and is reconfigured to execute the frequently executed traces of instructions. These trace instructions do not consume the core's resources, such as the front-end and execution bandwidth, registers, ROB-entries, etc, which are then available to other instructions, effectively increasing the per-core resources, and hence the per-core performance.

The reconfiguration bits for the RHU are generated at run-time to avoid recompilation of legacy codes. To further reduce the area overhead of the proposed mechanism, and hence maintain scalability, we propose a hardware/software co-design for reconfiguration bits generation, delegating the bits generation for multiple cores to a single core, organizing the reconfiguration bits as reconfiguration instructions with distinct opcodes, and utilizing the existing datapath for bit generation and RHU reconfiguration. We term the hardware used for reconfiguration instruction generation as RIG-hardware and the cores with RIG-hardware as RIG-cores.

Separating the RHU- and RIG-cores limits the per-core area overhead, maintains the scalability of the number of cores, and reduces the opportunity cost of integrating other

resources. The proposed architecture also better utilizes the RIG-hardware because if the RIG-hardware is included in each core, it will be idle for the majority of cycles; traces are formed once and executed many times. Furthermore, there will be no performance impact of trace generation if the number of threads is smaller than the number of cores because the RHU-cores do not incur any overhead for generating the reconfiguration instructions. Providing the RIG-hardware in each core may forfeit this benefit. If the number of threads concurrently executing is more than the number of cores, then only the thread scheduled on the RIG-core may have some performance impact. With run-time reconfiguration instructions generation, the proposed architecture does not require recompilation of legacy codes to improve their performance. Moreover, by making the reconfiguration instructions a part of the ISA, the proposed architecture can also support statically generated reconfiguration instructions.

Our experiments show that the RHU-core performance improves by about 23% across a wide variety of applications. The performance of the RIG-core lowers by an average of only about 0.4% to achieve the performance gains in the RHU-cores. Our studies show that our approach incurs a small per-core area overhead.

2 Proposed Multi-Core Architecture

2.1 Basic Idea

The cores in this multi-core architecture are divided into RIG-cores and RHU-cores. Once a RHU is reconfigured for a trace of instructions, those instructions are only executed on the RHU and not on the core's original datapath. On an exception or a branch misprediction from within the trace, the execution is restarted from the start of the trace and is performed entirely on the core's original datapath. The RHU is reconfigured through dynamically generated reconfiguration bits organized into chunks of 32-bits, called *reconfiguration instructions*. We assume 32-bit long instructions in our architecture. Each reconfiguration instruction consists of a 6-bit specialized operation code (opcode), which specifies the type of RHU reconfiguration performed by that instruction, and 26 reconfiguration bits.

A RIG-core generates the reconfiguration instructions for RHUs of multiple RHU-cores. When a RHU-core detects a frequently executed trace of instructions, it requests a RIG-core for reconfiguration instructions. The RIG-core generates the reconfiguration instructions, and communicates them back to the RHU-core. We do not provide the RIG-cores with a RHU, to somewhat equalize the per-core transistor budget. The cores use the existing interconnects for reconfiguration-related communications. The number of RHU-cores served by a single RIG-core is a design choice. Figure 1 shows one possible design for a 16-core processor, where each four-core cluster includes one RIG-core serving the remaining three RHU-cores.

When executing different threads on the core and the RHU, the RHU is reconfigured for traces from a different thread. Once, the RHU is reconfigured, the core and the RHU execute the two threads in parallel.

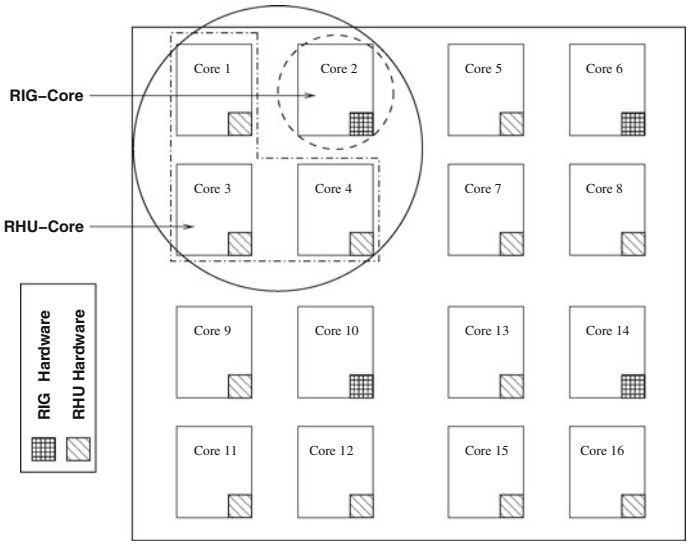

Fig. 1 Schematic diagram of a 16-core processor with designated RIG-cores and RHU-cores

2.2 RHU-Core

We call the original application instructions that are part of a trace executed on the RHU as RHU-instructions (*RIs*) and those executed on the core's original datapath as Proc-instructions (*PIs*). The results of RIs consumed by PIs are defined as live-outs, and those of PIs consumed by RIs are defined as live-ins. RIs are selected from within the innermost loops in the applications. To keep the architecture simple, a trace starts from the first instruction in the loop and consists of contiguous instructions. Figure 2 shows the schematic diagram of the RHU-core datapath. The *trace detection* logic detects traces of frequently executing instructions. When the reconfiguration instructions are received back, they are stored in the RHU-core thread's address space; at the end of its code section. When the start of a trace is detected in the fetch stage, its reconfiguration instructions are fetched, decoded using a slightly modified decode stage, and forwarded to the RHU. The RHU is reconfigured using the reconfiguration bits from the instructions. Once all the reconfiguration instructions are fetched, the fetch continues from the original instruction after the last instruction in the trace. Next, we discuss the RHU-core datapath in detail.

RHU Structure: To keep the RHU structure simple, only integer ALU operations, including address generation for memory instructions, are performed on the RHU. The most intuitive RHU structure is a two-dimensional array of interconnected ALUs. Figure 3a shows a RHU with three rows and four columns (identified as 3×4 RHU in the paper). This RHU structure exploits ILP and is also able to execute strings of dependent instructions.

We use a non-clocked RHU, i.e. the RHU is just a combinational logic. A non-clocked RHU reduces the RHU complexity, the RHU power consumption, and the

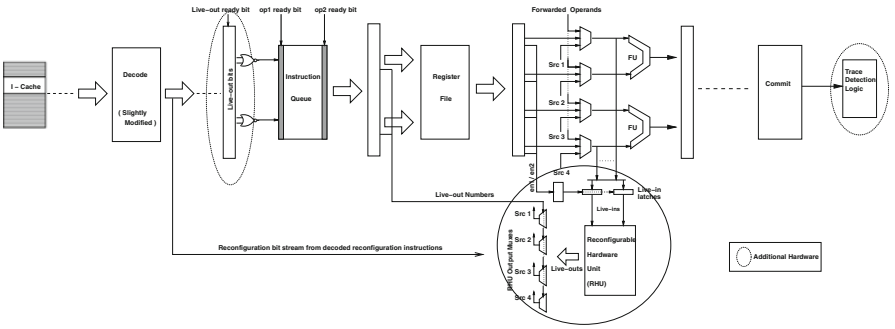

Fig. 2 Schematic diagram of the RHU-core datapath

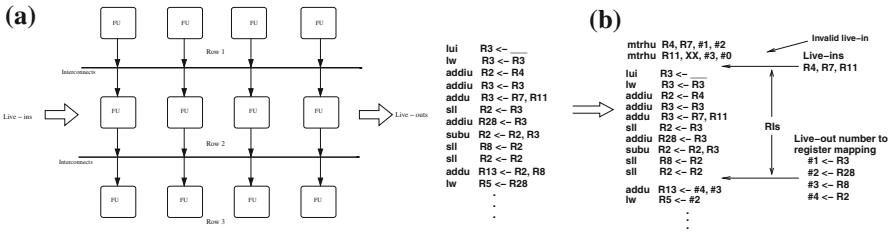

Fig. 3 **a** RHU structure with three rows and four columns (identified as 3×4 RHU) and **b** An example illustrating a trace of RIs along with the associated mappings, mtrhu instructions, and dependent PIs

overall live-in to live-out latency of the RHU. It is highly unlikely that the result of a RI is required in all the ALUs in the next row. Hence, to further reduce the RHU-complexity, an ALU's output is forwarded only to the left operand of four ALUs in the next row. This also ensures a FO4 delay for each ALU output and restricts the length of the interconnects. The number of live-ins and live-outs per row are also limited to further reduce the RHU-complexity. However, since the instructions in the first row only require live-ins and the instructions in the last row only produce live-outs, more live-ins are provided to the first row and more live-outs are extracted from the last row.

Since the time of live-out availability is not known, to wake-up PIs dependent on live-outs, a ready bit is propagated from all live-in values along all the datapaths in the RHU. An AND gate is used per ALU to generate the ready bits. Each ready operand sets its ready bit; when the ready bits of both the operands of an ALU are set, the result's ready bit is also set. Eventually, when the ready bit of a live-out is set, the dependent PIs are woken up, as discussed later. RHU also includes internal bit-cells to reconfigure the interconnect and the ALUs, and to store the immediate operands.

Communication between RHU and core datapath: Our approach uses a pseudo instruction—move to rhu: *mtrhu* R_s, R_t, *en1, en2*—to forward the live-in values to the RHU. En1 and en2 are small immediate values packed together in the immediate field of the instruction. This instruction forms a part of the reconfiguration instructions for a trace, and is executed on the core's original datapath. When an mtrhu instruction issues, it forwards the two source registers (R_s and R_t), and en1 and en2 to the RHU,

as shown in Fig. 2. The R_s and R_t values are latched in the live-in latches specified by en1 and en2. Once a live-in value is latched, it is not overwritten in the execution of that instance of the trace on the RHU.

The live-outs from the RHU are also numbered. No latches are provided for the live-out values, because the RHU is a combinational logic and once the live-in values are latched, the live-out values do not change. PIs dependent on RIs are renamed to the appropriate live-out numbers. The live-out values are directly forwarded into the functional units using the live-out muxes, as shown in Fig. 2. A PI dependent on a live-out activates the appropriate live-out mux on issue. Figure 3b shows an example of a trace of RIs along with the live-out registers to live-out numbers mapping (this mapping is also a part of the reconfiguration instructions), the mtrhu instructions, and the renaming of the dependent PIs. This example is taken from the `applu` Spec2K benchmark. The trace has three live-ins, requiring two mtrhu instructions to forward them to the RHU.

To rename dependent PIs, an additional bit is used for each rename map table entry. Every time a trace is executed, the live-out register to number mapping is copied into the rename map table and the associated bits are set. The bits are reset when the registers are overwritten by PIs. To wakeup PIs dependent on live-outs, each entry in the issue queue CAM portion is provided with live-out bits, equal to the maximum number of live-outs, as is shown in Fig. 2. At dispatch, the live-out bits for live-outs required by a PI are set if those live-outs have not yet been produced. Each live-out ready bit from the RHU (discussed earlier) resets the corresponding live-out bit in the CAM entries. The live-out bits of an entry are NORed together and the output of the NOR gate is participates in determining the availability of the left operand [33]. If a PI has both operands dependent on live-outs, its right operand ready bit is set, i.e. the PI waits for the left operand ready bit to be set. This approach, does not impact the delays in the issue queue logic.

Trace Detection: The *trace detection logic* consists of one *target* register, one *source* register, a fixed number of branch *outcome* bits, and a small saturating *threshold* counter. When a looping-back branch commits, its PC is compared with the source register, its target address with the target register, and the outcomes of the following branches with successive branch outcome bits. The source and target registers and the outcome bits are also updated with the new values. If all the comparisons match, the threshold counter is incremented, else the counter is reset. When the counter saturates, it is reset and the following actions are taken for the trace: (i) if it is already mapped onto the RHU or generation of its reconfiguration instructions is pending, nothing is done, (ii) if its reconfiguration instructions are already generated, they are mapped onto the RHU, and (iii) if it is an entirely new trace, a reconfiguration instructions generation request is sent to the RIG-core. To send the request, the RHU core communicates the core id, the target instruction's physical address, and the branch outcome bits to a RIG-core.

RHU Reconfiguration and Trace Execution: To generate longer traces, trace formation continues beyond instructions that cannot be executed on the RHU, such as complex integer and floating-point instructions, and halts when a RI candidate cannot

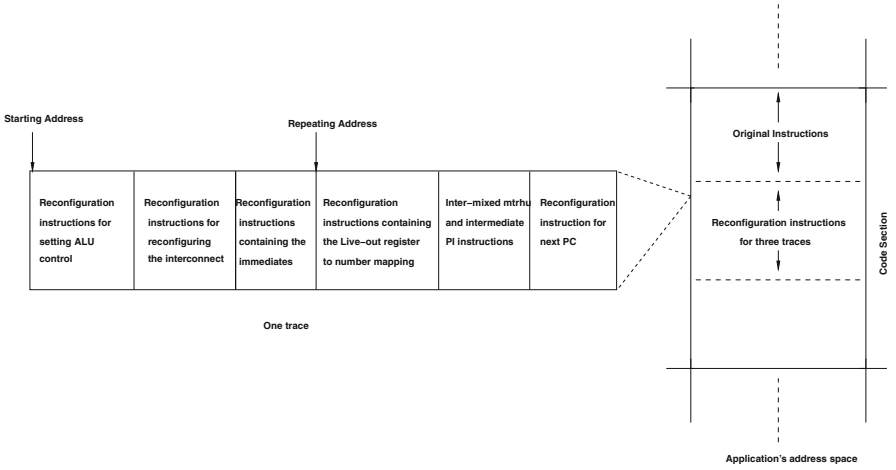

Fig. 4 Reconfiguration instruction organization

be mapped on the RHU. The intermediate PIs are included as part of the reconfiguration instructions. Figure 4 shows a trace consisting of the reconfiguration instructions and the intermediate PIs. A separate opcode is used for each type of reconfiguration instruction. The next PC is the address of the PI immediately following the end of the trace. The reconfiguration instructions for the ALU, the interconnect, and the immediate values are required only the first time the RHU is reconfigured for a trace. The rest of the reconfiguration instructions are required every time the trace is executed. Hence, two memory addresses—starting and repeating—are stored for each trace as shown in Fig. 4.

In our implementation, we store the three most recently encountered traces; a new trace replaces the least recently used trace. We found that storing more traces did not give any noticeable improvement. The traces are stored at the end of the code section for the thread running on the RHU-core, as shown in Fig. 4. An additional 3-entry *trace address* buffer is also provided in the RHU-core. Each entry in this buffer includes, the start PC, the branch PCs and their outcome bits, the starting and repeating addresses, the age for replacement, and two status bits for a trace. When the threshold counter in the trace detection logic saturates, the target PC and the outcome bits are compared with those in the trace address buffer. On a match, if the status bits of the matching entry are ‘01’ (indicating that the trace is already mapped onto the RHU) or ‘10’ (indicating that trace is being mapped onto the RHU) or ‘11’ (indicating that the reconfiguration instructions generation is pending), then nothing is done. Else the status bits are made ‘10’, but the reconfiguration of the RHU is not started.

The trace address buffer is also accessed in the fetch stage on any predicted looping-back branch. If the PC of the target matches an entry in the trace address buffer, then the fetch is diverted to the starting address of the trace if the status bits are ‘10’, or to the repeating address of the trace if the status bits are ‘01’. If the status bits are ‘10’, they are changed to ‘01’. When a trace is executed for the first time, the RHU is reconfigured. Finally the next PC is used to continue fetching from the original code.

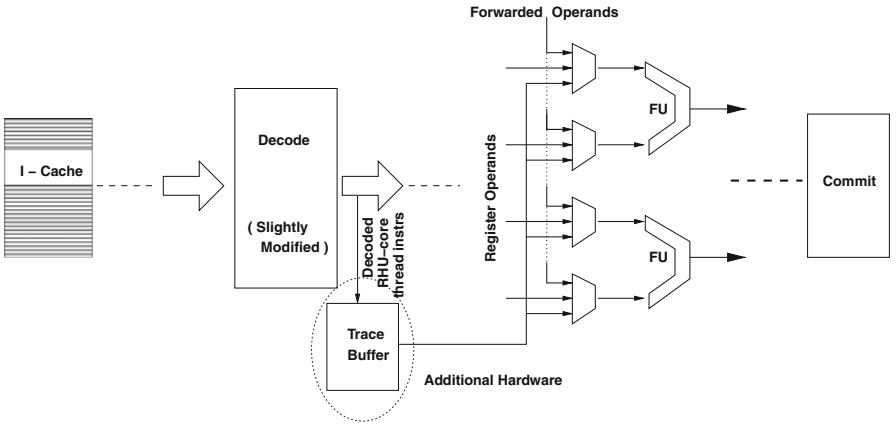

Fig. 5 Schematic diagram of the reconfiguration instructions generation core datapath

On an exception or branch misprediction from within the trace, the execution restarts from the start of the trace, using original instructions, and is performed entirely on the core's datapath. The status bits for the trace buffer are made '00'. When the trace commits, the branch predictor is updated for the branch instructions within the trace using the PCs and the outcomes from the trace address buffer, and the live-out registers are copied into the architectural register file. The RHU cannot be reused before the current instance of the trace mapped on it commits.

2.3 RIG-Core

Exclusive hardware mechanisms to generate reconfiguration instructions may have a large area overhead. In this paper, we propose a hardware/software co-design for reconfiguration instructions generation, where the reconfiguration bits are generated in hardware and then converted into reconfiguration instructions using an embedded software.

Figure 5 shows the schematic diagram of the RIG-core datapath. The RIG-hardware includes the *trace buffer*, shown in Fig. 5, and a small request queue, not shown in the figure, to queue the requests if the core is busy with another trace. Since the requests are expected to arrive infrequently, only a small request queue may be required. If the request queue is full, the request is dropped and the RHU-core is intimated. The RHU-core accordingly updates status bits in the trace address buffer.

The RIG-core fetches instructions for the RHU-core thread using the existing fetch datapath. While fetching the RHU-core thread instructions, RIG-core stalls the fetch of the current thread running on it. The RIG-core thread is not context-switched out of the core, and its instructions already in the pipeline continue to execute. The RHU-core thread instructions are decoded and forwarded to the trace buffer, as shown in Fig. 5.

The trace buffer hardware is shown in Fig. 6. It has two buffers, instruction buffer to store the instruction information and row buffer to store the availability of live-ins, live-outs, and slots in the RHU rows. Each instruction buffer entry consists of

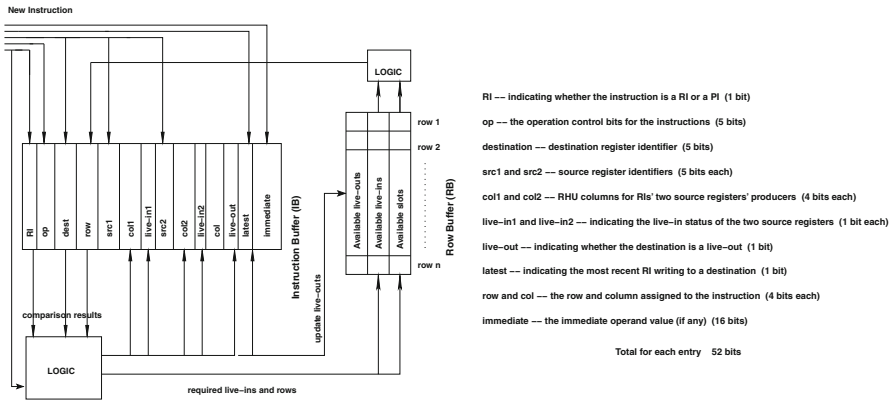

Fig. 6 Trace buffer hardware

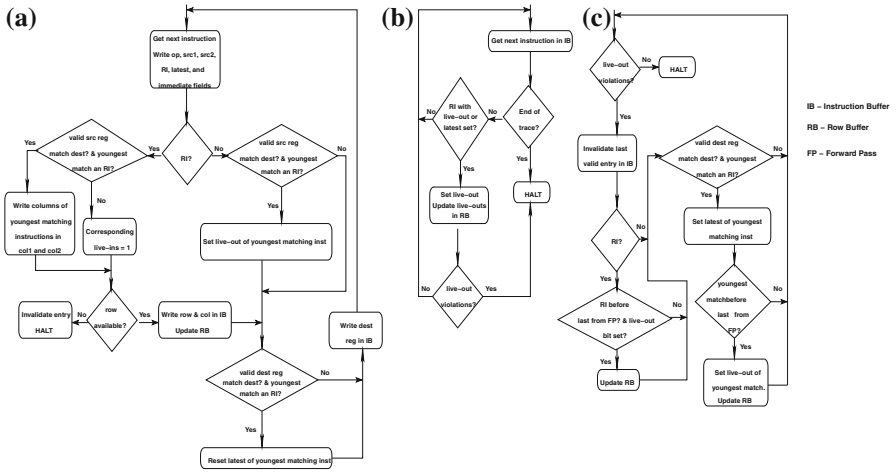

Fig. 7 **a** Phase 1; **b** Forward pass of Phase 2; **c** Reverse pass of Phase 2; of reconfiguration bits generation

several fields as depicted in Fig. 6. We observed that a 30-entry instruction buffer, requiring about 195 bytes, is sufficient to form the traces for a RHU with 36 instruction slots. The row buffer requires about 6-12 bytes depending on the number of live-ins and live-outs per row. A combinational logic is used to perform the trace buffer operations, as discussed later in this section.

To optimize the hardware for reconfiguration bits generation, only one RHU-core thread instruction is fetched and analyzed at a time.

Reconfiguration Bits Generation: The RHU reconfiguration bits are generated in two phases as shown in Fig. 7. In phase 1 (Fig. 7a), the operation bits, source register identifiers, and immediate values of the RHU-core thread instructions are written into the trace buffer, one instruction at a time. The *RI bit* is set for RIs and the *latest bit* is set for RIs with valid destination. In parallel, each instruction compares its operand

register identifiers with destination register identifiers of the RIs in the instruction buffer. If the current instruction is a PI, the youngest matching entries are marked as live-outs. If the current instruction is a RI, the columns of the youngest matching RIs are written into the *col1* and *col2* fields of the current instruction. The *live-in* bits of the operands that do not match are set.

The topmost row where all the operands of an RI are available is allocated to each RI. The columns in a row are used from left to right. The availability of rows and columns is obtained from the row buffer. If an instruction cannot be assigned a row, its entry is invalidated and phase 1 is halted. Phase 1 for each instruction requires 3 cycles. Phase 2 starts after phase 1 and operates only on the instructions in the instruction buffer. Hence, the RIG-core thread instructions can be fetched and executed in parallel to phase 2. Once phase 2 completes, the RIG-core thread fetch is stalled and embedded software instructions are fetched and executed to form the reconfiguration instructions.

Phase 2 has a forward pass (Fig. 7b) and a reverse pass (Fig. 7c) through the instruction buffer. In the forward pass, each RI with a set live-out bit decrements the appropriate live-out availability in the row buffer. Additionally, each RI with a set latest bit also sets its live-out bit and decrements the appropriate live-out availability in the row buffer. The forward pass of phase 2 requires one cycle per instruction. If the row buffer updates lead to live-out violations, then the forward pass is halted.

The reverse pass keeps invalidation the last valid instruction till the live-out violations are removed. The reverse pass may even continue beyond the instruction where the forward pass halted, in which case the live-out availability in the row buffer is incremented for all invalidated RIs with set live-out bit. The destination of each invalidated RI is also compared with the destinations of previous RIs; the latest bit of the youngest matching RI is set. If the RI, whose latest bit is set, is before the instruction where the forward pass halted, then its live-out bit is also set. If that live-out bit was not already set, then the live-out availability in the row buffer is appropriately decremented. The reverse pass requires two cycles per instruction.

Reconfiguration Instructions Generation: Once the hardware fills the instruction buffer fields, the embedded software reads each instruction buffer entry and generates the reconfiguration instructions. On system startup, the start address of the software is provided by the operating system to the RIG-core. Since the embedded software is initiated by the hardware, switching to the supervisor mode may not be required.

We use a specialized load instruction—*ldib* R_s *immediate*—in the embedded software to directly access the data in the instruction buffer. The immediate value specifies the instruction buffer entry to be read. The ldib instruction places the instruction buffer entry bits in register R_s for a 64-bit machine and in registers R_s and $R_s + 1$ for a 32-bit machine. The RIG-core datapath is implemented such that the ldib instructions use the existing register file ports. The embedded software instructions are fetched when the all in-flight RIG-core thread instructions have committed, and executed in-order. These instructions use only the speculative register file; they do not update the architectural register files. These instructions do not access the memory as well. Hence, the context of the RIG-core thread is intact in the core. We briefly discuss the operations performed by the embedded software.

The embedded software loads the instruction buffer entries into the core's registers. Shift and compare operations are then performed on these registers to extract the reconfiguration bits for each row. For instance, the ALU control reconfiguration instructions require the col and op field values and the interconnect reconfiguration instructions require the col1, col2, live-in1, live-in2, and col field values. The extracted reconfiguration bits are shifted into one of the registers to form reconfiguration instructions. The reconfiguration instructions are forwarded to the RHU-core as they are formed.

Once all the rows are completed, the reconfiguration instructions specifying the live-out registers to numbers mapping are formed. This is followed by forming the reconfiguration instructions for the inter-mixed mtrhu and PI instructions. For this, the instruction buffer is traversed in-order collecting the live-in registers (using the live-in1, live-in2, src1, src2, and row field values), and simultaneously forming the mtrhu instructions with two live-ins per instruction. When a PI is encountered, any pending live-in is placed in a mtrhu instruction. The collection of live-in registers is started after the PI, and the process continues.

Trace Formation Techniques: Trace formation for RHU is dependent on the placement of instructions in the RHU. In the simplest trace formation technique—technique 1—phase 1 is followed by the forward pass of phase 2 and then by the reverse pass of phase 2 till all the live-out violations are resolved. A more complex technique—technique 2—inserts forwarding instructions (FIs) in the RHU to forward values across rows. For instance, if value produced in row one is required in row three, then a FI is inserted in row two to forward the value. The FI can be made visible in the external ISA to facilitate static reconfiguration instruction generation. A FI is treated as any regular RI.

FIs are inserted in phase 1 if a row cannot be allocated using the existing criteria. For live-out violations in phase 2, FIs are also inserted to forward and extract the live-out values from other rows. To avoid shifting the instructions in the instruction buffer, the FIs in phase 2 are added after the last valid entry in the instruction buffer. For these FIs, the immediate field holds the index of the instruction whose live-out is forwarded by them. In the reverse pass of phase 2, if an instruction is invalidated, the FIs associated with that instruction are also invalidated.

After each RI is invalidated in the reverse pass, the forward pass violating instruction is re-checked for violations. If it can be included in the trace, then reverse pass is halted and the forward pass is continued. If another violating instruction is encountered, then forward pass is halted and the reverse pass is re-initiated, and the process continues. In this technique, two additional cycles per instruction are added in each phase when a FI is involved.

Our experiments show that technique 2 forms the largest traces, at the expense of higher trace formation latency. We also experimented with other techniques that had complexity and performance between the two techniques. To save space, we do not discuss the intermediate techniques. Figure 8 shows the traces formed for a 4×4 RHU by the two techniques for the example of Fig. 3b. In this example, only one live-in and one live-out are provided for each intermediate row. The `ld` instructions are not included in the trace in this example, as discussed in Sect. 2.4. The FIs enable the

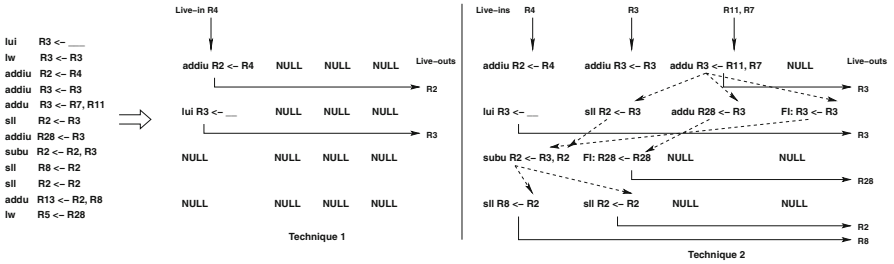

Fig. 8 An example illustrating the traces formed with the two techniques

inclusion of a total of nine instructions in the trace, as compared to two without them. Hence, we only present the results with technique 2.

2.4 Integration Models

We experiment with two RHU integration models. In one model—RHU Excluding Memory Instructions (REMI)—load and store instructions are only executed as PIs. In the other model—RHU Including Memory Instructions (RIMI)—load and store instructions are also executed as RIs. Trace formation in REMI skips memory instructions.

In our RIMI implementation, a maximum of one memory instruction can be placed in a specific column in each row. An address bus is extracted from that column in each row to output the memory address. For a store instruction in the RHU, the value to be stored is also extracted as a live-out from the same row in which the store instruction is present. In the RIMI model, the memory addresses output from the RHU are multiplexed into the existing load/store buffer (LSB). For this, LSB entries, equal to the number of RHU rows, are allocated for the trace memory instructions. The values loaded by load instructions within the RHU are directly forwarded to the live-in ports used by them.

The relative program order of stores and that of loads and stores is maintained, i.e. a store preceding other stores and loads is placed in a row prior to them, to maintain the correct order in the LSB. However, the relative program order between loads may not be maintained.

3 Experimental Results

3.1 Experimental Setup

We experiment with a quad-core processor with three RHU-cores and one RIG-core. The cores are interconnected using a 64-bit bus interconnect. In this paper, we experiment with non-data-sharing threads scheduled on the cores. The processor is realistically modeled by modifying a Simplescalar simulator to work in a multi-core mode.

Table 1 Experimental parameters for each core

Parameter	Value	Parameter	Value
Fetch/commit width	4 Instructions	Instr. window size	8 Int/8 mem/16 FP instructions
ROB size	96 Instructions	Issue width	1 Int/1 mem/2 FP
Speculative register file	48 Int/48 FP	Int. functional units	1 ALU, 1 Mul/div, 1 AGU
Load/store buffer	40 Entries	FP Functional units	2 ALU, 1 Mul/div
Branch predictor	Gshare 4 K entries	L2—cache (shared by 4-cores)	Unified 2M, 8-way assoc., 20 cycles
L1—I-cache	16 K, Direct-mapped, 1 cycle latency	L1—D-cache	16 K, 4-way assoc. 64 bytes block, 1 cycle latency

We assume a data transfer rate in the bus of three cycles for the first 64 bits and one cycle for each additional 64-bit transfer.

The hardware features and default parameters of each core are given in Table 1. Each core consists of separate integer and floating point subsystems. The issue queue and the issue width in the integer subsystem are further partitioned among the integer and memory instructions. Separate architectural register files are used to maintain the architectural state of the registers. The per-core resources are constrained to depict a core in a multi-core processor with large number of cores, and are similar to those in the current multi-core implementations. For instance, AMD has a 3-issue core and Intel 64 has a 4-issue core. Note that with increasing number of cores, the per-core resources may further reduce, in which case our approach is expected to give better performance improvement. The presence of the embedded software instructions in the L1 and L2 caches is realistically modeled by tagging those instructions and fetching and evicting them using the existing replacement policies.

For benchmarks, we use a collection of 15 Spec2K (`art`, `ammp`, `applu`, `apsi`, `bzip2`, `equake`, `gcc`, `mcf`, `mesa`, `mgrid`, `parser`, `swim`, `vortex`, `vpr`, `wupwise`), 3 MediaBench [29] (`epic`, `g721`, `mpeg2`), and 9 MiBench [15] (`sha`, `basicmath`, `bitcount`, `qsort`, `dijkstra`, `susan`, `adpcm`, `CRC32`, `FFT`) benchmarks. The statistics are collected for 200M instructions after skipping 1B instructions for Spec2K benchmarks and 50M instructions for the rest. For better legibility, we present the individual results of a representative set of nine benchmarks (`art`, `equake`, `mesa`, `mgrid`, `vpr`, `sha`, `susan`, `CRC32`, and `FFT`). This set represents the average performance improvement of a benchmark on an RHU-core. For each benchmark running on the RIG-core, we experiment with three different sets of benchmarks running on the RHU-cores. For instance, with `art` on the RIG-core, we experiment with 3 sets on the RHU-cores—`equake`, `mesa`, and `mgrid`; `mgrid`, `vpr`, and `sha`; `susan`, `CRC32`, and `FFT`. We evaluate the performance of each benchmark on an RHU-core after averaging its performance in all its runs on an RHU-core. Similar approach is used to evaluate the performance of a benchmark on an RIG-core.

In this paper, we start with a 6×6 RHU and progressively improve the RHU structure to increase the overall trace size. A maximum of six branches can be included in

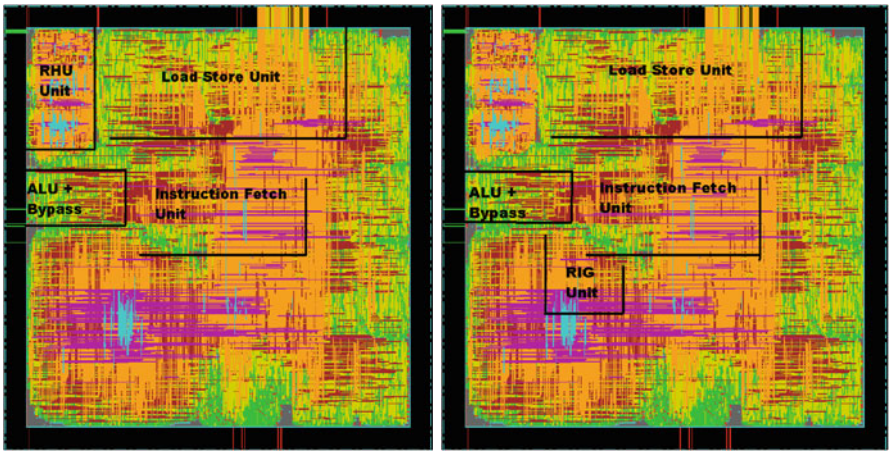

Fig. 9 Die image of RHU core and RIG core

a trace. Each ALU is connected to one on the left, one immediately below, and two on the right. The ALUs at the end are connected to one immediately below and three on the side. *We assume the delay in each RHU row to be equal to one CPU clock cycle*. Studies have shown that ALUs can be operated at high speeds [18], making the reduced-complexity RHU feasible for processors with a wide range of operating frequencies.

3.2 Area Results

We integrated a 36-ALU RHU (in particular a 4×9 RHU) with one SUN T1 Open-Source core [32] of an eight-core processor, to estimate the area overhead of RHU- and RIG- hardware. The design for the RHU and RIG-hardware was synthesized using Synopsys Design Compiler using a TSMC 90nm Standard cell library [40], and was placed and routed using Cadence SoC Encounter. After integrating the RHU, the core area increased by about 2.5%. The RIG-hardware adds about 3% to the core area. Figure 9 shows the die image of the RHU and the RIG cores. If the RHU and the RIG-hardware are included in every core, the per core area overhead would have increased by about 5.5%, instead of 3 and 2.5%. Furthermore, our experiments show that integrating the RHU and RIG-hardware in each core does not give any noticeable performance benefits over our approach.

The per core resources of the SUN T1 Opensource core may not exactly match the per core parameters in Table 1. However, integration of the additional hardware into the SUN T1 core gives a reasonably accurate measure of the per-core area overhead of our approach in an eight-core processor. Figure 9 shows that the RHU is placed close to the functional and the load/store units as the RHU interacts with them, whereas the RIG-hardware is placed close to the fetch/decode and the functional units.

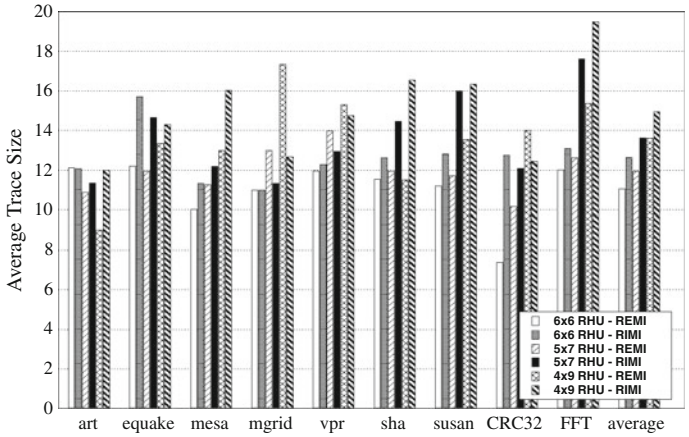

Fig. 10 Average trace sizes of original instructions for different RHU structures

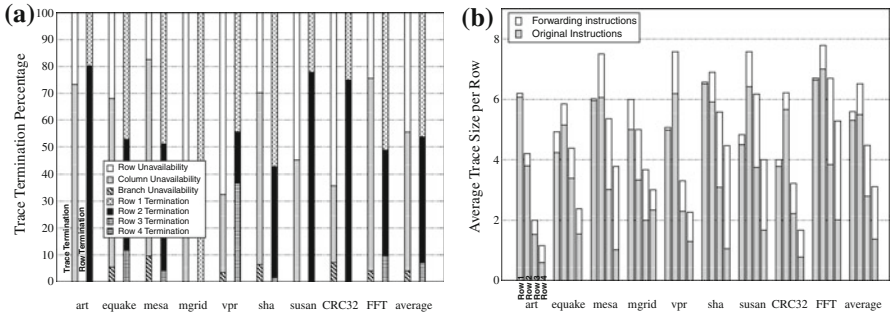

Fig. 11 **a** Percentage distribution of the different reasons for trace termination, and **b** Average number of original and forwarding instructions per row, on 4 × 9 RHU for RIMI

3.3 Trace Results

In our experiments, we observed that two live-ins and two live-outs per intermediate row are enough to form large traces. Row one is provided with nine live-ins and one live-out is extracted from each ALU in the last row. Our experiments showed that the average trace size (not including the FIs) was considerably smaller than the maximum possible 36 instructions for the 6 × 6 RHU. The traces were small primarily because they were terminated due to *column unavailability*, i.e. an RI is not able to get a slot in the required row. Hence, we also explored 5 × 7 and 4 × 9 RHUs, which provide more columns than rows, while requiring almost the same number of ALUs as 6 × 6 RHU.

Figure 10 compares the trace sizes for 6 × 6, 5 × 7, and 4 × 9 RHUs. The 4 × 9 RHU performs the best with an overall average trace size of about 15 original instructions for RIMI. Figure 10 shows that the average trace size generally increases from REMI to RIMI. However, for some benchmarks such as `mgrid` and `vpr`, including the memory instructions reduces the average trace size. We observed that this was primarily because of an increase in the number of FIs due to store instructions.

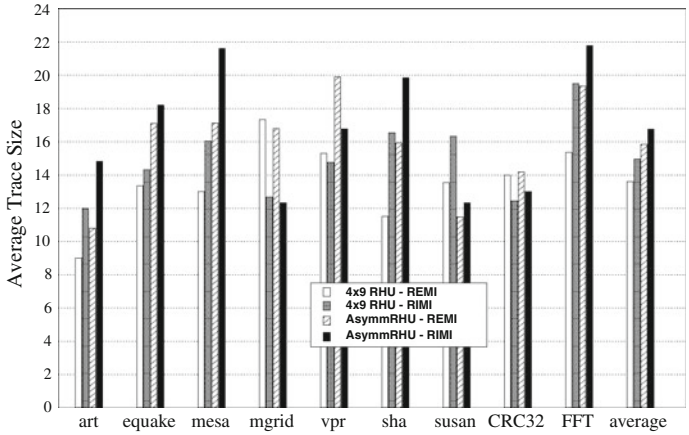

Fig. 12 Average trace sizes in RIMI and REMI for 4 × 9 RHU and AsymmRHU

The first bar in Fig. 11a shows the percentage distribution of the reasons for trace termination on the best performing 4 × 9 RHU for RIMI. A trace is considered terminated because of *branch unavailability* if branches beyond the maximum allowed are encountered, and a trace is considered terminated because of *row unavailability* if a dependent needs to be placed beyond the last row. Figure 11a shows that trace terminations due to column and row unavailability are almost balanced for the 4 × 9 RHU. The second bar in the figure shows the percentage distribution of trace terminations due to column unavailability in the different rows. Most of the column unavailability related trace terminations occur in the top two rows.

Figure 11b further shows the average number of instructions in the rows of a 4 × 9 RHU. It can be seen that most of the original instructions are concentrated in the top two rows, whereas most of the FIs are concentrated in the bottom two rows. This suggests that many FIs are inserted in the bottom two rows to forward the values produced by the instructions in the top two rows, primarily to extract them as live-outs. Figure 11b also shows that more instructions are placed in rows one and two. Overall, Fig. 11 suggests that more columns and live-out ports are required in the top two rows.

To further increase the trace sizes, we exploit the observations in Fig. 11 and investigate an asymmetrical RHU structure—*AsymmRHU*—for the 36 ALUs. AsymmRHU is provided 11 columns in the first and second rows, six columns in the third row, five columns in the fourth row, and three columns in a fifth row. A fifth row is added to reduce the trace terminations due to row unavailability observed in Fig. 11a. However, the live-ins and live-outs per intermediate row are kept at two. All the ALUs in rows four and five are provided with live-outs. In AsymmRHU, each ALU output is still forwarded to four ALU-inputs in the next row. Figure 12 compares the trace sizes, excluding the FIs, of AsymmRHU with the 4 × 9 RHU. Figure 12 shows that the trace sizes increase with AsymmRHU, with the overall average reaching almost 17 instructions in RIMI.

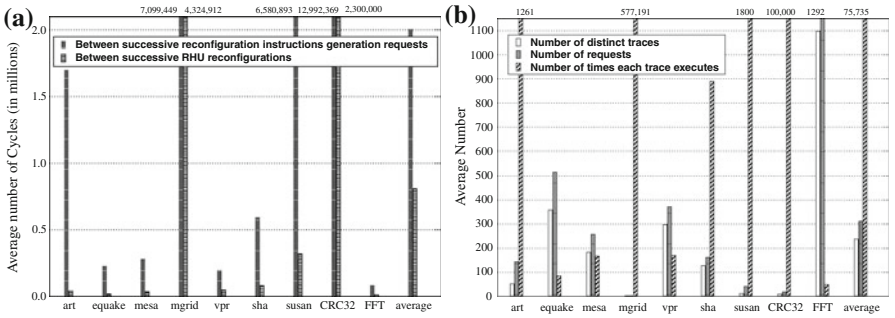

Fig. 13 **a** Average number of trace generation requests and RHU reconfigurations and **b** Average number of new trace requests and times each trace executes

RHU Coverage: We observed that the RIs formed a considerable fraction of the overall instructions executed in the applications, about 25% for REMI and about 33% for RIMI for 4 × 9 RHU. The corresponding numbers for AsymmRHU are 29 and 37%, respectively. The percentage of instructions executed as RIs depends on the size of the inner-most loops and the percentage of the total instructions in the application that lay within the inner-most loops.

3.4 RIG-Core Performance Impact

In our experiments, we observed that the hardware takes an average of about 180 cycles, across the benchmarks, for generating the RHU reconfiguration bits. The embedded software takes an average of about 428 cycles for converting the reconfiguration bits into reconfiguration instructions. Our experiments showed that the average performance overhead in the RIG-core is less than 0.1%.

The low performance impact in the RIG-core is because of infrequent trace generations. Figure 13a presents the average number of cycles between successive trace generation requests and between successive RHU reconfigurations. Figure 13b shows the average number of distinct traces formed, the average number of new trace generation requests, and the average number of times each trace is executed. Figure 13b shows that a trace is executed a large number of times once it is formed. The number of requests is larger than the number of distinct traces because of evictions. On an average, only about 300 traces are generated, in the process of committing 500 million instructions, per benchmark. Hence, the average number of cycles between successive requests is high, about 2 million cycles as shown in Fig. 13b. The RIG-core, thus, receives only a small number of requests and spends minimal time in generating the reconfiguration instructions for the RHU-cores.

Interconnect Pressure: The low frequency of requests also results in negligible impact on the interconnect pressure from the trace-generation-related communication between RHU- and RIG-cores for reconfiguration instructions generation. We

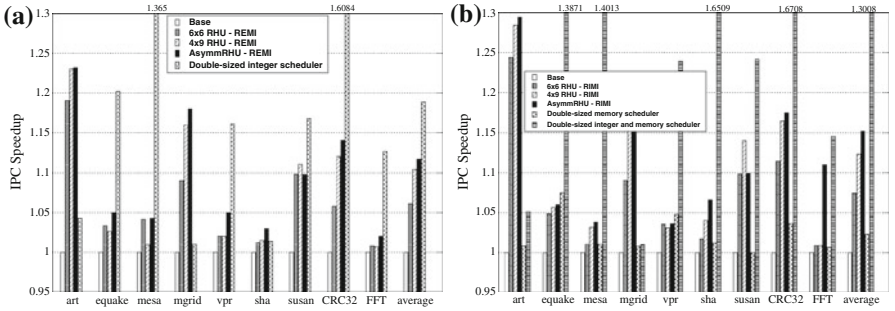

Fig. 14 IPC speedup of RHU-cores with **a** REMI, and **b** RIMI compared to the base core configuration and that of cores with double-sized schedulers

observed that an average total of only about 49 words are communicated per trace between the RIG-core and the RHU-cores.

3.5 RHU-Core Performance Results

Figure 13a shows that an average of about 800,000 cycles elapse between successive RHU reconfigurations. Hence, the overhead of executing the reconfiguration instructions to reconfigure the RHU for the first time is also negligible.

Next, we present the performance (IPC) improvement of RHU-cores, with 6×6 RHU, 4×9 RHU, and AsymmRHU, over the base core, for REMI in Fig. 14a, and for RIMI in Fig. 14b. Figure 14 also shows the IPC speedup of cores with double-sized integer scheduler, with double-sized memory scheduler, and with double-sized integer and memory scheduler. A double-sized scheduler doubles the issue queue size and issue width of the base case shown in Table 1. The number of functional units are accordingly increased. Figure 14 shows that the RIMI model performs better than the REMI model. The maximum average IPC speedup of about 15% is obtained with AsymmRHU in RIMI. The 4×9 RHU achieves about 12% IPC speedup in RIMI. The corresponding numbers for REMI are 11.7% for AsymmRHU and 10.4% for 4×9 RHU.

Interestingly, our approach performs significantly better than all the double-sized scheduler configurations for `art` and `mgrid`. This is because the double-sized scheduler is still limited by other resources such as the fetch width, registers, etc., the pressure on which is somewhat relieved by the RHU. Additionally, when instructions are executing on the RHU, the effective issue queue size and issue width may more than double during that time. Our approach almost always performs better than the double-sized memory scheduler configuration. However, the average performance of AsymmRHU in RIMI is about 2% lower than the double-sized integer scheduler and about 14% lower than the double-sized integer and memory schedulers. This is because the RHU only speeds up a part of the loop. The rest of the application runs with the narrow width. *It is important to note that doubling the schedulers for better performance may have much higher impact on the scalability than our approach because of the increased scheduler size and additional functional units, forwarding paths and register file ports.* Our experiments showed that the average number of instructions issued

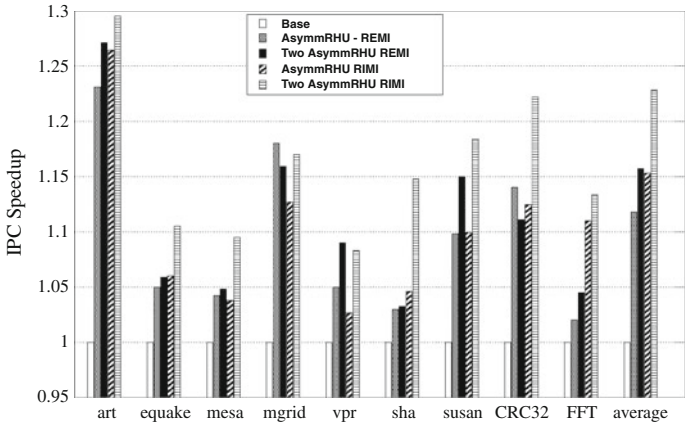

Fig. 15 IPC speedup of one AsymmRHU per core and two AsymmRHUs per core

per cycle in the integer and memory schedulers increased from about 0.48 in the base configuration to about 0.65 in AsymmRHU in RIMI, but fell short of 0.76 observed in the double-sized integer and memory scheduler configuration.

3.6 Two RHUs Per-Core

To further improve the RHU-core performance, we experimented with two RHUs in each RHU-core. Two traces are formed from each innermost loop. The RIG-core datapath is not modified. We observed that the RIG-core performance impact increases to about 0.4% due to forming more traces. The first trace starts from the first instruction of the innermost loops. The second trace starts from an instruction that is approximately at the middle of the loop, provided that that instruction is not included in the first trace. This approach maximizes the distance between the traces, thus exploiting local ILP within each RHU and distant ILP in the two RHUs.

To find the middle of the loop, an *instruction counter* is included in the trace detection logic. The instruction counter counts the number of instructions in an innermost loop. Another set of target register, branch outcome bits, threshold counter, and trace address buffer is used for the second traces. The remaining operations for trace detection, RHU reconfiguration, and trace execution are similar for the two traces in a loop. The only difference is that for the second trace, the PCs of all fetched instructions between the end of the first trace and the end of the loop are checked to detect the start of the second trace Second trace generation is initiated if no second trace has executed in a loop for a predetermined time period.

Figure 15 compares the speedup of two AsymmRHUs per core with that of a single AsymmRHU per core. Figure 15 shows that speedup increases for two AsymmRHU from about 11.7% to about 16.4% in REMI and from about 15% to about 23% in RIMI. We do not observe double performance improvement with two RHUs because two traces could not be formed in some loops, and there was not enough distant ILP to be exploited in some other cases. The area overhead of two RHUs is increases to about 5% in Fig. 9.

4 Related Work

To the best of our knowledge, no other efforts have been made to improve the scalability as well as per-core performance in multi-core processors. Other approaches have been proposed to improve the execution bandwidth of a single core superscalar processor. A few efforts have attempted to integrate FPGA modules with an all-purpose processor. PRISM [2], Spyder [23], Piperench [8] used a loosely coupled off-chip FPGA as a co-processor. Garp [7] integrates a FPGA as a co-processor on the processor chip. The FPGA-based co-processor approach cannot be used to improve both scalability and per-core performance of a multi-core processor. Other co-processor based proposals [25,30,36,38,41,43] were developed for application specific architectures.

Chimaera [17], PRISC [34], and OneChip [42] integrate the FPGA as a functional unit (RFU) in the processor datapath with direct access to the processor register file. The compiler statically generates the RFU instructions and FPGA reconfiguration bit-streams, which are used to dynamically reconfigure the FPGA. Authors in [26,44] also propose similar architectures. Tightly integrating an FPGA with a core will significantly limit the scalability because of the area overhead of a highly generalized reconfigurable FPGA [11]. FPGAs also tend to be considerably slower and have higher energy consumption as compared to the ICs. Furthermore, FPGAs prevent dynamic trace formation because of the extensive overhead in generating and communicating the huge bit-streams required for reconfiguring the FPGA.

A few other approaches execute aggregated instructions on custom functional units for single core superscalar processors. Intel's Pentium M [24] fuses certain micro-op pairs to reduce the number of x86 instructions that decode into micro-op sequences. The authors in [19,20] propose extensions to the x86 ISA to support the fusion of micro-op pairs. Macro-op scheduling [27] also fuses dependent ALU instruction pairs and executes them on closed-loop ALUs. These approaches target pairs of ALU instructions. Dynamic strands [37] extend macro-op scheduling beyond pair-wise aggregation still using closed-loop ALUs. The authors in [3,4,12] propose fusion of instructions in a dependence chain to form a mini-graph and replacing the mini-graph with a special instruction. These architectures provide non-reconfigurable custom functional units for the mini-graphs. Other approaches identify portions of an application's data flow graph that can be efficiently implemented in hardware [1,5,10,14,21,22,39] to design custom hardware for them in application-specific processors. This process is highly computationally intensive and is performed statically [11].

Clark et al. [11] propose a restrictive reconfigurable custom compute accelerator (CCA) that has a maximum of four inputs and two outputs. This CCA is reconfigured to execute subgraphs that consist of a small number of instructions terminating at branch and memory instructions. The subgraph formation and execution methodology is proposed for a single core processor, whereas our approach is targeted towards multi-core processors. However, the authors provide very good insights into reconfigurable computing for dependence graphs. The authors acknowledge the performance limitations of terminating at branch and memory instructions in [9], a restriction not present in our approach. Hence, in [9], they also propose execution of more arbitrary acyclic sub graphs that cross branch boundaries and include memory instructions.

This approach requires store-load collapsing within the sub-graph, and is targeted for single-issue in-order embedded processors.

Commit time trace formation has also been proposed to improve the fetch bandwidth and perform dynamic optimizations in superscalar processors [13]. However, the reconfiguration instruction generation in our approach is significantly different from the trace formations for superscalar processors.

5 Conclusion

In a multi-core processor, scalability of the number of cores and per-core performance conflict one another. The design choice is between having more cores with poor per-core performance and having good per-core performance but with fewer cores. In this paper, we explore a multi-core architecture that improves per-core performance, while maintaining the ability to scale the number of cores. In the architecture, the cores are divided into two categories—RHU-cores and RIG-cores. RHU-cores have integrated reconfigurable hardware unit (RHU) to improve their performance. The reconfiguration instructions for multiple RHU-cores are generated by a single RIG-core, thus reducing RIG-hardware overhead and improving its utilization. We propose innovative mechanisms to integrate the RHU in the core's datapath, to generate reconfiguration instructions using a hardware/software co-design, and to reconfigure the RHU. These mechanisms keep the area of the additional hardware requirement to a minimum, and have a small impact on the scalability of the number of cores. The proposed architecture improves the average per-core performance of RHU-cores by about 23%. The approach has a 0.4% impact on the RIG-core performance to achieve the performance gains in the RHU-cores.

References

1. Atasu, K. et al.: Automatic application-specific and instruction-set extensions under microarchitectural constraints. DAC (2003)
2. Athanas, P. et al.: Processor reconfiguration through instruction-set metamorphosis. IEEE Computer, **26**(3), (1995)
3. Bracy, A. et al.: Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth. Proc. MICRO (2004)
4. Bracy, A. et al.: Serialization-Aware Mini-Graphs: Performance with Fewer Resources. Proc. MICRO (2006)
5. Brisk, P. et al.: Instruction generation and regularity extraction for reconfigurable processors. CASES (2002)
6. Burger, D., Austin, T.M.: The SimpleScalar Tool Set, Version 2.0. Computer Arch. News, June (1997)
7. Callahan, T. et al.: The garp architecture and c compiler. IEEE Comput. **33**(4), 62–69 (2000)
8. Chou, Y. et al.: Piperench implementation of the instruction path coprocessor. Proc. MICRO (2000)
9. Clark, N. et al.: An architecture framework for transparent instruction set customization in embedded processors. Proc. ISCA (2005)
10. Clark, N. et al.: Processor acceleration through automated instruction-set customization. Proc. MICRO (2003)
11. Clark, N. et al.: Application specific processing on a general purpose core via transparent instruction set customization. Proc. MICRO (2004)
12. Corliss, M.L. et al.: DISE: A programmable macro engine for customizing applications. Proc. ISCA (2003)

13. Fahs, B. et al.: Performance characterization of a hardware mechanism for dynamic optimization. Proc. MICRO (2001)
14. Goodwin, D., Petkov, D.: Automatic generation of application specific processors. CASES (2003)
15. Guthaus, M.R. et al.: MiBench: a free, commercially representative embedded benchmark suite. Work. Workload Characterization (2001)
16. Hammond, L. et al.: A single-chip multiprocessor. IEEE Comput. **30**(9), 79–85 (1997)
17. Hauck, S. et al.: The chimaera reconfigurable functional unit. Proc. FCCM (1997)
18. Hsu, S.K. et al.: An 8.3 GHz dual supply/threshold optimized 32b integer ALU-register file loop in 90 nm CMOS. ISLPED (2005)
19. Hu, S. et al.: An approach for implementing efficient superscalar CISC processors. Proc. HPCA (2006)
20. Hu, S., Smith, J.: Using dynamic binary translation to fuse dependent instructions. Int. Symp. on CGO (2004)
21. Huang, I., Despain, A.M.: Synthesis of application specific instruction sets. IEEE TCAD (1995)
22. Huang, Z. et al.: Design of dynamically reconfigurable datapath processors. ACM Trans. on Embedded Computing Systems. **3**(2) (2004)
23. Iseli, C., Sanchez, E.: Spyder: a sure (superscalar and reconfigurable) processor. J. Supercomput. **9**(3), 231–252 (1995)
24. Intel corporation, mobile intel pentium 4 m-processor datasheet, Jun. 2003. http://www.intel.com/design/mobile/datashts/250686.htm
25. Jacob, J.A., Chow, P.: Memory interfacing an instruction specification for reconfigurable processors. Symp. FPGAs (1999)
26. Kastrup, B. et al.: Concise: a compiler-driven cpld-based instruction set accelerator. Proc. FCCM (1999)
27. Kim, I., Lipasti, M.: Macro-op scheduling: relaxing scheduling loop constraints. Proc. MICRO (2003)
28. Kumar, R. et al.: Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling. Proc. ISCA (2005)
29. Lee, C., et al.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. Proc. MICRO (1997)
30. Miyamori, T., Olukotun, K.: Remarc: reconfigurable multimedia array co-processor. IEICE Trans. Inf. Syst. **E82-D**(2), 389–397 (1999)
31. Olukotun, K. et al.: The case for a single-chip multiprocessor. ASPLOS-VII (1996)
32. Sun Microsystems, Inc.: OpenSPARC T1 micro architecture specification. Sun Microsystems, Inc (2006)
33. Palacharla, S. et al.: Complexity-effective superscalar processors. Proc. ISCA (1997)
34. Razdan, R., Smith, M.: A high-performance microarchitecture with hardware-programmable functional units. Proc. MICRO (1994)
35. Renau, J. et al.: (2005) SESC Simulator. http://sesc.sourceforge.net
36. Rupp, C.R. et al.: The napa adaptive processing architecture. Proc. FPGAs for computing machines (1998)
37. Sassone, P., Wills, D.: Dynamic strands: collapsing speculative dependence chains for reducing pipeline communication. Proc. MICRO (2004)
38. Singh, H. et al.: Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans. Comput. **49**(5), 465–481 (2000)
39. Sun, F. et al.: Synthesis of custom processors based on extensible platforms. ICCAD (2002)
40. TSMC 90 nm Core Library—TCBN90GHP, Application note—Revision 1.2 (2006)
41. Vassiliadis, S. et al.: The molen polymorphic processor. IEEE Trans. Comput. **53**(11) (2004)
42. Wittig, R., Chow, P.: Onechip: an fpga processor with reconfigurable logic. Proc. FCCM (1996)
43. Wong, S. et al.: Coarse reconfigurable multimedia unit extension. Proc. 9th Euromicro workshop on Parallel and Distributed Processing (1996)
44. Ye, Z. et al.: A c compiler for a processor with a reconfigurable functional unit. Proc. Symp. on FPGAs (2000)