Scalable Multi-cores with Improved Per-core Performance Using Off-the-critical Path Reconfigurable Hardware

Tameesh Suri and Aneesh Aggarwal

Department of Electrical and Computer Engineering
State University of New York at Binghamton
Binghamton, NY 13902, USA
{tameesh,aneesh}@binghamton.edu

Abstract. Scaling the number of cores in a multi-core processor constraints the resources available in each core, resulting in reduced per-core performance. Alternatively, the number of cores have to be reduced in order to improve per-core performance. In this paper, we propose a technique to improve the per-core performance in a many-core processor without reducing the number of cores. In particular, we integrate a Reconfigurable Hardware Unit (RHU) in each core. The RHU executes the frequently encountered instructions to increase the core's overall execution bandwidth, thus improving its performance. We also propose a novel integrated hardware/software methodology for efficient RHU reconfiguration. The RHU has low area overhead, and hence has minimal impact on the scalability of the multi-core. Our experiments show that the proposed architecture improves the per-core performance by an average of about 12% across a wide range of applications, while incurring a per-core area overhead of only about 5%.

1 Introduction

Recently, there has been a major shift in the microprocessor industry towards multi-core processors. Multi-core processors have considerably fewer resources available 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 [24]. The per-core resources are reduced to be able to integrate more number of cores on a single chip. A 56-entry issue queue on a four-way issue PA-8000 processor takes up about 20% of the die area [22]. It is obvious that a similar sized scheduler cannot be integrated in each core of a multi-core processor with large number of cores.

Fewer per-core resources degrades the performance of each thread of execution [12]. Scaling the number of cores on a chip may further reduce the per-core resources. Increasing the number of cores also exacerbates the reduction in resources by increasing die area required for peripheral hardware such as interconnects [20]. Hence, it may be argued that in a typical multi-core processor, the design choice is between increasing the number of cores with poor per-core

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 365–377, 2008.

performance and having a good per-core performance but with fewer cores on the chip.

In this paper, we propose a multi-core architecture that improves the performance of the resource-constrained cores of multi-cores with large number of cores, while maintaining the scalability of the number of cores. The per-core performance is improved by integrating an off-the-critical path reconfigurable hardware unit (RHU) in its datapath. The RHU 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, which are then available to other instructions, effectively increasing the per-core resources, and hence the per-core performance. The proposed RHU structure also has only a small area overhead, and hence only a small impact on the scalability of the number of cores.

We also propose a novel methodology for run-time RHU reconfiguration, in which RHU reconfiguration bits are generated using a hardware/software co-design and are divided into reconfiguration instructions. The reconfiguration bits are generated at run-time to avoid recompilation of legacy code. Our experiments show that per-core performance improves by about 12% across a wide variety of applications. The area overhead to achieve the improvement is about 5% per core.

2 Multi-core Design

2.1 Basic Idea

Each core in a multi-core has an integrated RHU that is reconfigured to execute frequently executed instructions from the thread executing on the core. 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, thus effectively increasing the resources for other instructions. 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. The proposed architecture also supports static (compile-time) reconfiguration instruction generation by making these instructions visible in the external ISA. Each reconfiguration instruction consists of a specialized operation code (opcode) followed by reconfiguration bits.

2.2 Core Microarchitecture

Different instructions are simultaneously executed on the RHU and the core's original datapath. We call the original instructions (not the reconfiguration 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. To maximize the benefit of the

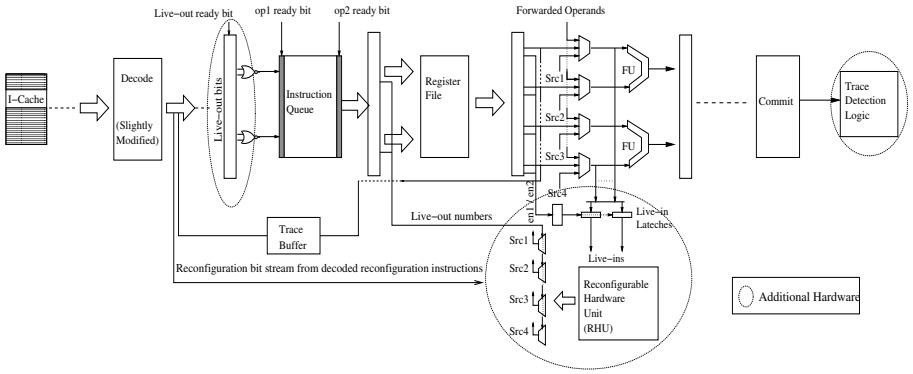

Fig. 1. Schematic diagram of the core datapath

RHU and to keep the architecture simple, RIs are selected from within the innermost loops in the applications; starting from the first instruction in the loop and forming a trace of contiguous instructions. Figure 1 shows the schematic diagram of the core datapath.

The *trace detection* logic detects traces of frequently executing instructions. Once a trace is detected, reconfiguration instructions are generated for it, as discussed later. To limit area overhead, we store the reconfiguration instructions in the 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 instead of the original instructions. The reconfiguration instructions are decoded and the RHU is reconfigured using the reconfiguration bits. After the trace reconfiguration instructions are over, the fetch continues from the original instruction following the last instruction in the trace.

RHU Structure: To keep the RHU structure simple, only integer ALU operations are performed on the RHU. The most intuitive RHU structure is a two-dimensional array of interconnected ALUs. 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 overall live-in to live-out latency of the RHU. To drastically reduce the RHU-interconnect 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. The number of live-ins and live-outs per row are also limited to further reduce the RHU-complexity.

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. This instruction 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 Figure 1. The R_s and R_t values are latched in the live-in latches specified by en1 and en2. Once a live-in value

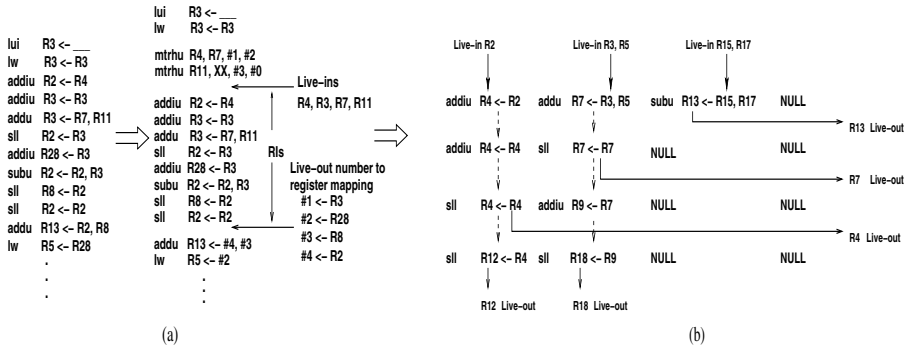

Fig. 2. An example illustrating: (a) Trace of RIs along with the associated mappings, mtrhu instructions, and dependent PIs (b) Trace layout on a 4x4 RHU

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. 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 Figure 1. Figure 2(a) 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. Figure 2(b) shows an example of a trace mapped on a 4x4 RHU, using a simple trace formation technique discussed later.

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 Figure 1. The live-out bits for live-outs required by a PI are set at dispatch, if those live-outs are not available. A ready bit is propagated in the RHU from all live-in values along all the datapaths in the RHU. Each live-out ready bit from the RHU resets the corresponding live-out bit in the CAM entries; the operand becomes ready when all live-out bits are reset.

Trace Detection: The *trace detection logic* records the most recently encountered backward branching instruction, and the outcome bits of a few following branches to detect innermost loops. When a frequently executed trace is detected from within an innermost loop, 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, its reconfiguration instructions are generated.

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 be mapped on the RHU. The intermediate PIs are included

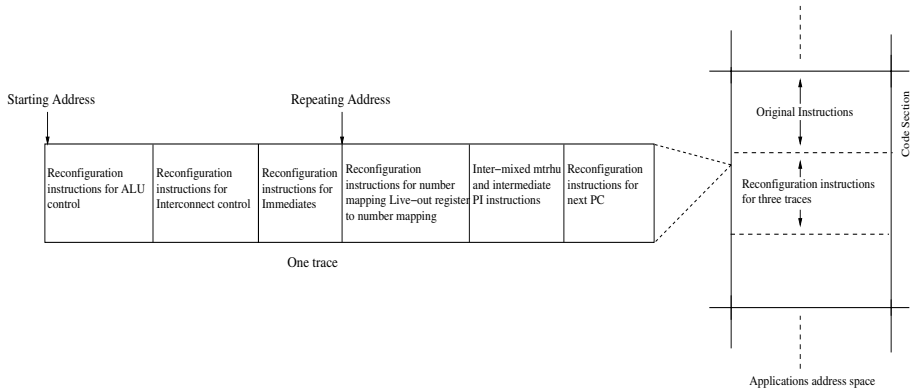

Fig. 3. Reconfiguration instruction organization

as part of the reconfiguration instructions. Load and store instructions are also executed as PIs to simplify the memory disambiguation. Figure 3 shows a trace consisting of the reconfiguration instructions and the intermediate PIs. The next PC is the address of the PI immediately following the end of the trace. The instructions for reconfiguring the RHU are executed 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 Figure 3.

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, as shown in Figure 3. An additional 3-entry *trace address* buffer is also provided to maintain the status of the traces. Every backward branching instruction's target is compared with the start PC of the traces in the trace buffer, and the execution is accordingly directed. 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. It is important to note that the RHU cannot be reused before the current instance of the trace mapped on it commits.

2.3 Reconfiguration Instructions Generation

Performance overhead of exclusive software mechanisms to generate reconfiguration instructions is high, while area overhead of exclusive hardware mechanisms is high. In this paper, we explore an integrated hardware/software mechanism for reconfiguration instructions generation, where the reconfiguration bits are generated in hardware and then converted into reconfiguration instructions using an embedded software.

The reconfiguration instruction generation hardware includes the *trace buffer*. The trace instructions are fetched, decoded, and forwarded only to the trace buffer. The thread is not context-switched out of the core, and its instructions

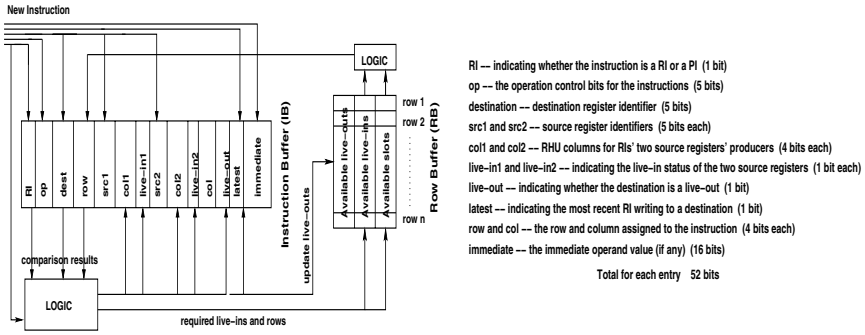

Fig. 4. Trace Buffer Hardware

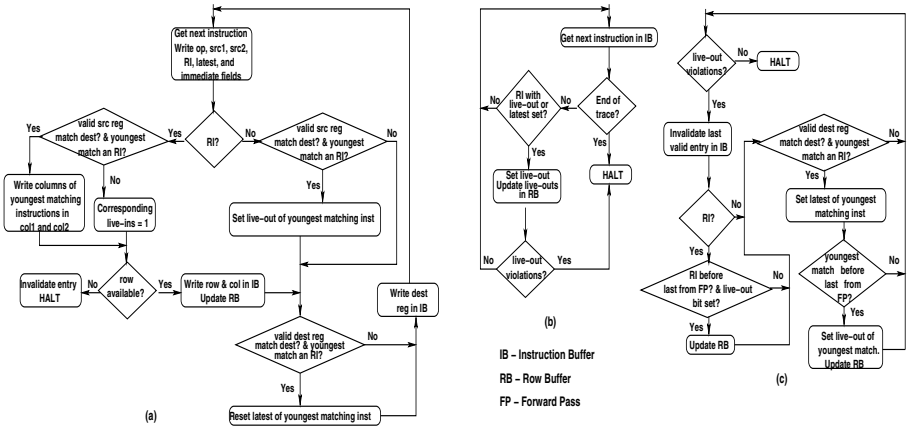

Fig. 5. (a) Phase 1; (b) Forward pass of Phase 2; (c) Reverse pass of Phase 2; of reconfiguration bits generation

already in the pipeline continue to execute. The trace buffer hardware (shown in Figure 4) 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. 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. Additional logic is also needed to control the trace buffer operations, as discussed later in this section. To optimize the hardware for reconfiguration bits generation, only one instruction is fetched and analyzed at a time.

Reconfiguration Bits Generation: The RHU reconfiguration bits are generated in two phases as shown in Figure 5. In phase 1, the trace buffer entries are filled for each instruction. In parallel, the dependencies are resolved by comparing the operands and destinations of instructions. In this phase, the rows and columns are also allocated to instructions depending on the availability of

operands. 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. Phase 2 has a forward pass(Figure 5(b)) and a reverse pass (Figure 5(c)) through the instruction buffer. In the forward pass, the live-outs are determined. If a live-out port is not available, then the forward pass is halted. The forward pass of phase 2 requires three cycles per instruction.

The reverse pass is used to remove any live-out violations in the forward pass. It invalidates the last valid entry in the instruction buffer, and attempts to restart the forward pass after every invalidation. If the forward pass is restarted, then reverse pass is halted and the forward pass is continued till another violation is encountered, and the process goes on. The reverse pass requires four cycles per instruction.

Reconfiguration Instructions Generation: Once phase 2 completes, embedded software (part of the operating system) instructions are fetched and executed to form the reconfiguration instructions. The embedded software reads each instruction buffer entry and generates the reconfiguration instructions. 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 embedded software instructions are executed in-order when the all in-flight thread instructions have committed. These instructions use only the speculative register file; they do not update the architectural register files, as shown in Figure 1. These instructions do not access the memory as well. Hence, the context of the thread is intact in the core. 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. The extracted reconfiguration bits are shifted into one of the registers to form reconfiguration instructions.

3 Experimental Results

3.1 Experimental Setup

We experiment with a quad-core processor. In this paper, we experiment with non-data-sharing threads scheduled on the cores. The hardware features and default parameters of each core are given in Table 1. 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 benchmarks, we use a collection of Spec2K and MiBench [11] 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`). We evaluate the

Table 1. Experimental parameters for each core

Parameter	Value	Parameter	Value
Commit Width	4 instr.	*Instr. Window Size*	8 Int/8 Mem/16 FP
ROB Size	96 instr.	*Issue Width*	1 Int/1 Mem/2 FP
Spec. Register File	48 Int/48 FP,	*Int. FUs*	1 ALU, 1 Mul/Div, 1 AGU
Load/store buffer	40 entries	*FP FUs*	2 ALU, 1 Mul/Div
Branch Predictor	gshare 4K entries	*L2 - cache* (shared by 4-cores)	unified 2M, 8-way assoc 20 cycles
L1 - I-cache	16K direct-mapped 1 cycle	*L1 - D-cache*	16K 4-way assoc. 64 bytes block, 1 cycle

performance of each benchmark after averaging its performance with different combinations of benchmarks running on the four cores. We assume the delay in each RHU row to be equal to one CPU clock cycle.

3.2 Area Results

We integrated a 4x9 RHU with one SUN T1 OpenSource core [25] of an eight-core processor, to get the area requirement. The design was synthesized using Synopsys Design Compiler using a TSMC 90nm Standard cell library [31], and was placed and routed using Cadence SoC Encounter. After integrating the additional hardware, the core area increased by about 5%. Figure 6(a) shows the die image of the modified core.

The per core resources of the SUN T1 Opensource core may not exactly match the per core parameters given 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. The RHU

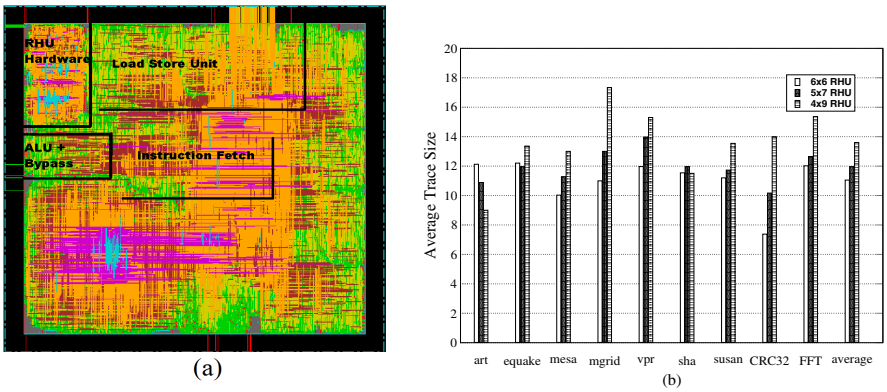

Fig. 6. (a) Die image of a Core (b) Average trace sizes of original instructions for different RHU structures

is placed close to the functional and the load/store units as the RHU interacts with them, whereas the reconfiguration bits generation hardware is placed close to the fetch/decode and the functional units.

3.3 Trace Results

We observed that two live-ins and two live-outs per intermediate row are enough to form maximum-sized 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 sizes were considerably smaller than the maximum possible 36 instructions for the 6x6 RHU. The traces were small primarily because they were terminated due to *column unavailability*, *i.e.* an instruction had to be placed in a row, but no free slots were available in that row. Hence, we also explored 5x7 and 4x9 RHUs that provide more columns than rows, while requiring almost the same number of ALUs as 6x6 RHU.

Figure 6(b) compares the trace sizes for 6x6, 5x7, and 4x9 RHUs. The 5x7 and 4x9 RHUs have significantly larger traces than the 6x6 RHU. The 4x9 RHU performs the best with an overall average trace size of about 15 instructions.

We also studied the reasons for trace termination on the best performing 4x9 RHU; the detailed results are not shown to conserve space. We observed that the traces were mostly terminated because of two reasons: *column unavailability* in the top two rows, and *row unavailability*, where a dependent needs to be placed beyond the last row. This suggests that more columns are required in the top two rows. To further increase the trace sizes, we 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. 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 atmost four ALU-inputs in the next row. Figure 7(a) compares the trace sizes of AsymmRHU with the 4x9 RHU. The trace sizes increase with AsymmRHU, with the overall average reaching almost 16 instructions.

RHU Coverage: We observed that the RIs formed about 17% fraction of the overall instructions executed for the 4x9 RHU, and about 21% for AsymmRHU. 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 Performance Results

Next, we present the performance (IPC) improvement of RHU-cores, with 6x6 RHU, 4x9 RHU, and AsymmRHU, over the base core. The IPC speedup is shown in Figure 7(b). Figure 7(b) also shows the IPC speedup of cores with double-sized integer scheduler. A double-sized scheduler doubles the issue queue size and

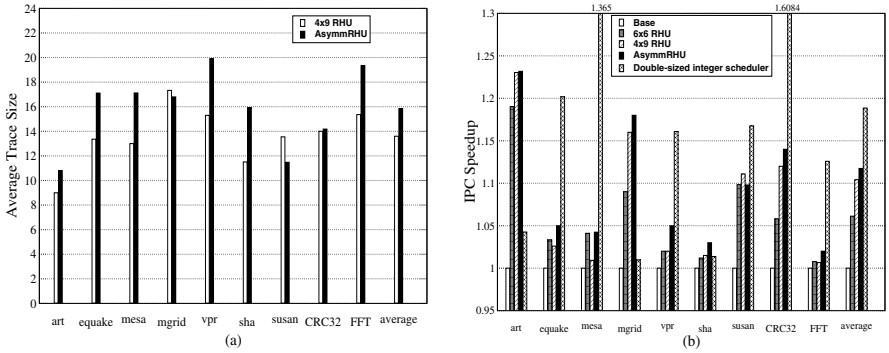

Fig. 7. (a) Average trace sizes for 4x9 RHU and AsymmRHU (b) IPC speedup of RHU-cores compared to the base core configuration and that of cores with double-sized schedulers

issue width of the base case shown in Table 1. The number of functional units are accordingly increased. 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. The maximum average IPC speedup of about 12% is obtained with AsymmRHU. The 4x9 RHU achieves about 10% IPC speedup.

Interestingly, our approach performs significantly better than the double-sized scheduler configuration for `art`, `mgrid`, and `sha`. 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. Still, the average performance of AsymmRHU is about 5% lower than the double-sized scheduler.

Our experiments showed an average of about 2 million cycles between successive reconfiguration instruction generation and an average of about 800,000 cycles between successive RHU reconfigurations. Hence, the overhead of our approach is minimal. The performance results in Figure 7(b) include the overhead of generating and executing the reconfiguration instructions.

4 Related Work

Previous studies integrate FPGA modules with a processor to improve performance. PRISM [1], Spyder [16], Piperench [5], and Garp [4] use a loosely coupled FPGA as a co-processor. Similar co-processor based proposals [18], [23], [32], [28], [30], [34] target application specific architectures.

Chimaera [13], PRISC [26], and OneChip [33] 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. FPGAs have high area overhead, are considerably slower, and have higher energy consumption as compared to the ICs. Furthermore, FPGAs incur extensive overhead in generating and communicating the huge bit-streams required for reconfiguring them.

Other approaches execute aggregated instructions on custom functional units, for instance, [14], [15], [17], [19] fuse x86 micro-op pairs. These approaches target pairs of ALU instructions. Dynamic strands [29] extend beyond pair-wise aggregation still targeting Integer ALU instructions. The authors in [2], [3], [9] fuse a dependence chain to form a special instruction, which is then executed on non-reconfigurable custom functional unit.

Clark et al. [8] propose a restrictive reconfigurable custom compute accelerator (CCA) that has a maximum of four inputs and two outputs, executing subgraphs of a small number of instructions terminating at branch and memory instructions. The authors acknowledge the performance limitations of terminating at branch and memory instructions in [6], a restriction not present in our approach. Hence, in [6], 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 [10]. 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. In this paper, we explore an architecture that improves per-core performance, with minimal impact on area. In the architecture, the cores have integrated reconfigurable hardware unit (RHU) to improve their performance. 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. The proposed architecture improves the average per-core performance by about 12% with about 5% area overhead.

References

1. Athanas, P., et al.: Processor reconfiguration through instruction-set metamorphosis. IEEE Computer 26(3) (1995)
2. Bracy, A., et al.: Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth. In: Proc. MICRO (2004)
3. Bracy, A., et al.: Serialization-Aware Mini-Graphs: Performance with Fewer Resources. In: Proc. MICRO (2006)
4. Callahan, T., et al.: The garp architecture and c compiler. IEEE Computer 33(4), 62–69 (2000)

5. Chou, Y., et al.: Piperench implementation of the instruction path coprocessor. In: Proc. MICRO (2000)
6. Clark, N., et al.: An architecture framework for transparent instruction set customization in embedded processors. In: Proc. ISCA (2005)
7. Clark, N., et al.: Processor acceleration through automated instruction-set customization. In: Proc. MICRO (2003)
8. Clark, N., et al.: Application Specific Processing on a General Purpose Core via Transparent Instruction Set Customization. In: Proc. MICRO (2004)
9. Corliss, M.L., et al.: DISE: A Programmable Macro Engine for Customizing Applications. In: Proc. ISCA (2003)
10. Fahs, B., et al.: Performance characterization of a hardware mechanism for dynamic optimization. In: Proc. MICRO (2001)
11. Guthaus, M.R., et al.: MiBench: A free, commercially representative embedded benchmark suite. Work. Workload Characterization (2001)
12. Hammond, L., et al.: A Single-Chip Multiprocessor. IEEE Computer 30(9) (September 1997)
13. Hauck, S., et al.: The chimaera reconfigurable functional unit. In: Proc. FCCM (1997)
14. Hu, S., et al.: An Approach for Implementing Efficient Superscalar CISC Processors. In: Proc. HPCA (2006)
15. Hu, S., Smith, J.: Using Dynamic Binary Translation to Fuse Dependent Instructions. In: Int. Symp. on CGO (2004)
16. Iseli, C., Sanchez, E.: Spyder: a sure (superscalar and reconfigurable) processor. Journal of Supercomputing 9(3), 231–252 (1995)
17. Intel Corporation, Mobile Intel Pentium 4 M-Processor Datasheet (June 2003), `http://www.intel.com/design/mobile/datashts/250686.htm`
18. Jacob, J.A., Chow, P.: Memory interfacing an instruction specification for reconfigurable processors. In: Symp. FPGAs (1999)
19. Kim, I., Lipasti, M.: Macro-op Scheduling: Relaxing Scheduling Loop Constraints. In: Proc. MICRO (2003)
20. Kumar, R., et al.: Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In: Proc. ISCA (2005)
21. Lee, C., et al.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In: Proc. MICRO (1997)
22. Lotz, J., et al.: A Quad-Issue Out-of-Order RISC CPU. In: Proc. Int'l. Solid-State Circuits Conf. (1996)
23. Miyamori, T., Olukotun, K.: Remarc: Reconfigurable multimedia array coprocessor. IEICE Trans. on information and systems E82-D(2), 389–397 (1999)
24. Olukotun, K., et al.: The Case for a Single-Chip Multiprocessor. In: ASPLOS (1996)
25. Sun Microsystems, Inc. OpenSPARC T1 Micro Architecture Specification, Sun Microsystems, Inc. (2006)
26. Razdan, R., Smith, M.: A high-performance microarchitecture with hardware-programmable functional units. In: Proc. MICRO (1994)
27. Rotenberg, E.: Trace cache: a low latency approach to high bandwidth instruction fetching. In: Proc. MICRO (1996)
28. Rupp, C.R., et al.: The napa adaptive processing architecture. In: Proc. FCCM (1998)
29. Sassone, P., Wills, D.: Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication. In: Proc. MICRO (2004)

30. Singh, H., et al.: Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans. on Computers 49(5), 465–481 (2000)
31. TSMC 90nm Core Library - TCBN90GHP, App. Note - Revision 1.2 (2006)
32. Vassiliadis, S., et al.: The molen polymorphic processor. IEEE Trans. on Computers 53(11) (2004)
33. Wittig, R., Chow, P.: Onechip: An fpga processor with reconfigurable logic. In: Proc. FCCM (1996)
34. Wong, S., et al.: Coarse reconfigurable multimedia unit extension. In: Proc. 9th Euromicro workshop on Parallel and Distributed Processing (1996)