2009 22nd International Conference on VLSI Design

Improving Scalability and Per-core Performance in Multi-cores through Resource Sharing and Reconfiguration

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

Abstract

Increasing the number of cores in a multi-core processor reduces per-core performance. On the other hand, providing more resources to each core limits the number of cores on a chip. In this paper, we propose a mechanism to improve the per-core performance while maintaining the scalability. In particular, we integrate a Reconfigurable Hardware Unit (RHU) in the resource-constrained cores to improve their performance. The RHU executes the frequently encountered instructions to increase the core's overall execution bandwidth, thus improving its performance. 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 performance improving mechanism, generation of the reconfiguration bits for the RHUs of multiple cores is delegated to a single core. Our experiments show that the proposed architecture improves the per-core performance by an average of about 23% across a wide range of applications, while incurring a small per-core area overhead.

1. Introduction

In any technology generation, increasing the number of cores in multi-core processors requires reducing resources in each core, which is further exacerbated by the increasing die area requirement for peripheral hardware such as interconnects, snoopy logic, etc [20]. Fewer per-core resources degrades the performance of each thread of execution [12]. In this paper, we propose a mechanism that improves the per-core performance while maintaining the scalability of the number of cores. We build on prior work [30], where a core's performance is improved by integrating an off-the-critical path reconfigurable hardware unit (RHU) in its datapath. Speed-up is obtained by executing frequently executed instructions on the RHU. These instructions do not consume the core's resources, effectively increasing the per-core performance.

In this paper, we use the approach for a multi-core processor. We also extend the approach by including memory instructions, as it was a limiting factor in [30]. The reconfiguration bits for the RHU are generated at run-time and each core consists of a hardware/software co-design methodology to generate reconfiguration bits along with the RHU. Furthermore, we propose a innovative methodology of delegating the reconfiguration bits generation for multiple cores to a single core. The reconfiguration bits are arranged as reconfiguration instructions. 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, 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. With this approach, 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.

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.

1063-9667/09 $25.00 © 2009 IEEE
DOI 10.1109/VLSI.Design.2009.58

2 Proposed Multi-core Architecture

There are two main themes of the proposed architecture – reconfiguration to improve performance and division of cores into RIG-cores and RHU-cores to maintain scalability. The RHU is reconfigured through dynamically generated reconfiguration instructions, consisting of a 6-bit specialized operation code (opcode).

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. We do not provide the RIG-cores with a RHU, to somewhat equalize the per-core transistor budget. The number of RHU-cores served by a single RIG-core is a design choice. For instance, in a 16-core processor, each four-core cluster may include one RIG-core serving the remaining three RHU-cores.

2.1 RHU-Core and trace execution

The RHU-core organization is mostly the same as that in [30]; we do not discuss it in detail here for want of space. Furthermore, the execution of a trace is also the same as explained in [30]. The differences are the following: Trace buffer – which is used to generate reconfiguration instructions in [30] is now a part of the RIG-core datapath, as shown in figure 1. Also, once a trace is detected by the RHU-core, reconfiguration instruction generation request is sent to a RIG-core. When the reconfiguration instructions are received back from the RIG-core, they are stored at the end of the RHU-core thread's code section.

In our 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. 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 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.

2.2 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 1 shows the schematic diagram of the RIG-core datapath. The RIG-core fetches instructions for the

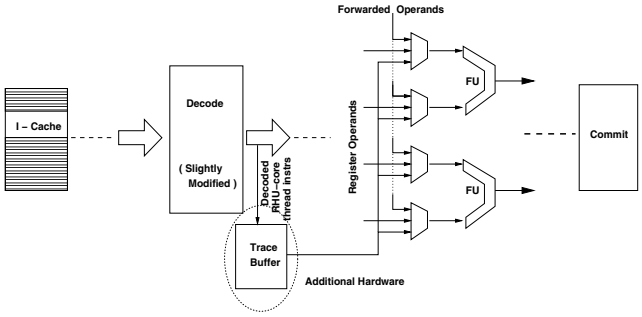

Figure 1. Schematic diagram of the Reconfiguration Instructions Generation core datapath

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 [30], as shown in Figure 1.

Trace buffer hardware and operation is identical as used in [30] However, to optimize the hardware for reconfiguration bits generation, only one RHU-core thread is fetched and analyzed at a time.

In trace formation, we use an innovative method of forwarding instructions (FIs) to forward values across rows, in order to obtain larger traces for the RHUs. Trace sizes were limited in [30] due to unavailability of column or row in the RHU. FIs provide a cost-effective method to obtain larger traces than having more live-ins, live-outs or ALUs. As more live-ins are available in the top row and more live-outs are provided in the bottom-most row, FIs can be included to forward the values, and allow more flexibility in placing an instruction. FIs are also used 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. A FI is treated as any regular RI.

The RHU reconfiguration bits are generated in two phases, as detailed in [30]. However, phase 1 and 2 are modified to incorporate FIs. We briefly go over the operation of reconfiguration bit generation. In phase 1, 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. FIs are inserted in phase 1 if operands of an instruction are available in different rows. 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. Phase 2 has a forward pass and a reverse pass through the instruction buffer. In the forward pass, the live-

outs are determined. If a live-out port is not available, then FIs are inserted, and if even that fails, 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 and remains the same as in [30]. The reverse pass requires four cycles per instruction.

We also experimented with simpler trace formation techniques, but the experiments showed that this technique forms the largest traces, at the expense of higher trace formation latency.

Once phase 2 completes, the RIG-core thread fetch is stalled and embedded software [30](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. 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. The operation of the embedded software is detailed in [30], and remains the same. The reconfiguration instructions are forwarded to the RHU-core as they are formed.

3 Experimental Results

3.1 Experimental Setup

We experiment with a quad-core processor with three RHU-cores and one RIG-core. 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 instance, Hydra [24] has four 2-issue cores.

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 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.

3.2 Area Results

We integrated a 36-ALU RHU (in particular a 4x9 RHU) with one SUN T1 OpenSource core [25] 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 [31], 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 2 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.

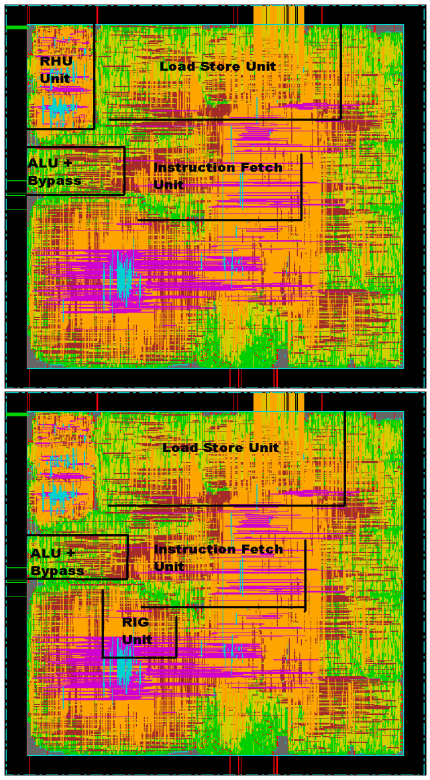

Figure 2. Die image of RHU Core and RIG Core

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. Previous studies [22, 24] suggest that a slight increase in the width of a core will easily increase the core area much more than the RHU. Hence, issue-width of a core is constrained while scaling the number of cores in a CMP. Figure 2 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.

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 4K entries	*L2 - cache* (shared by 4-cores)	unified 2M, 8-way assoc., 20 cycles
L1 - I-cache	16K, direct-mapped, 1 cycle latency	*L1 - D-cache*	16K, 4-way assoc. 64 bytes block, 1 cycle latency

Table 1. Experimental parameters for each core

3.3 Trace Results

We experiment with 36-ALUs, investigating 6x6, 5x7 and 4x9 RHUs. The 4x9 RHU performed the best with an overall average trace size of about 15 instructions. We observed that trace terminations due to column and row unavailability are almost balanced for the 4x9 RHU. We further observed 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. Overall, our experiments suggested that more columns and live-out ports are required in the top two rows.

To further increase the trace sizes, we also 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. 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 3 compares the trace sizes, excluding the FIs, of AsymmRHU with the 4x9 RHU. Figure 3 shows that the trace sizes increase with AsymmRHU, with the overall average reaching almost 17 instructions. Our experiments also showed that the RIs formed a considerable fraction of the overall instructions executed in the applications, about 33% for 4x9 RHU and 37% for AsymmRHU.

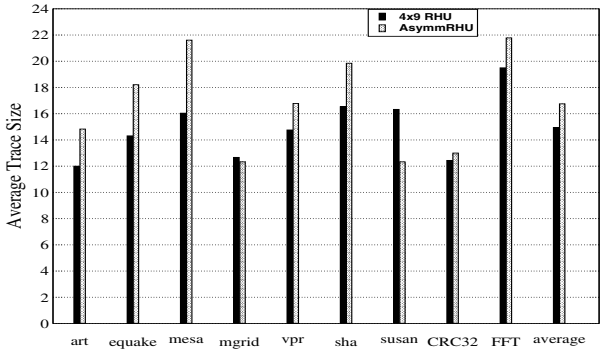

Figure 3. Average trace sizes for 4x9 RHU and AsymmRHU

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 4 presents the average number of cycles between successive trace generation requests and between successive RHU reconfigurations. 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 Figure 4. The RIG-core, thus, receives only a small number of requests and spends minimal time in generating the reconfiguration instructions for the RHU-cores.

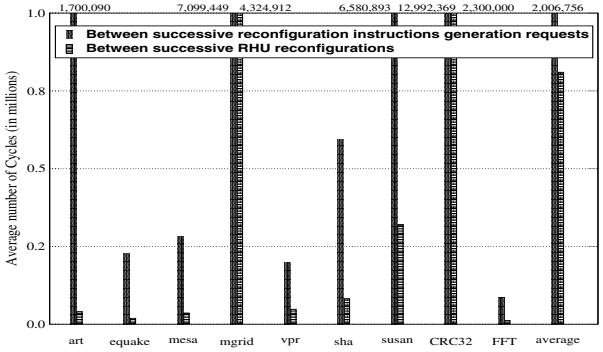

Figure 4. Average number of Trace generation requests and RHU reconfigurations

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 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 4 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 6x6 RHU, 4x9 RHU, and AsymmRHU, over the base core, in Figure 5. Figure 5 also shows the IPC speedup of cores 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. The maximum average IPC speedup of about 15% is obtained with AsymmRHU. The 4x9 RHU achieves about 12% IPC speedup.

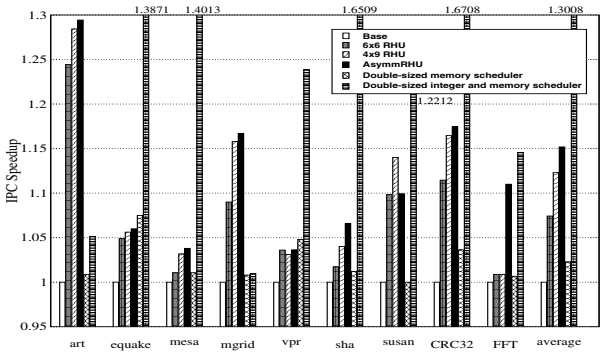

Figure 5. IPC speedup of RHU-cores compared to the base core configuration and that of cores with double-sized schedulers

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 is 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 per cycle in the integer and memory schedulers increased from about 0.48 in the base configuration to about 0.65 in AsymmRHU, 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.

Figure 6 compares the speedup of two AsymmRHUs per core with that of a single AsymmRHU per core. Figure 6 shows that speedup increases for two AsymmRHU from about 15% to about 23%. 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 Figure 2.

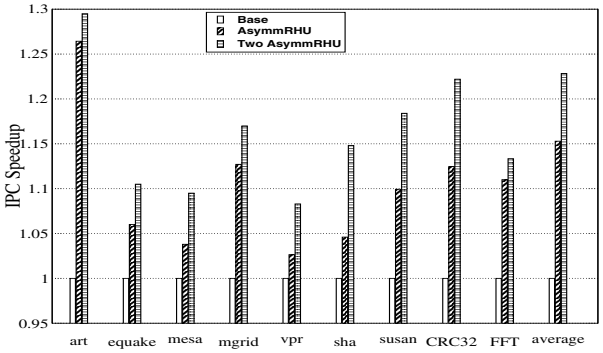

Figure 6. IPC speedup of one AsymmRHU per core and two AsymmRHUs per core

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] [27] [29] 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 [28] 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.

Authors in [30] execute memory instructions as PIs to simply memory disambiguation. Trace generation hardware and RHU are included in each core, resulting in a large per-core area overhead.

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. 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] P. Athanas et al., "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer,26(3)*, 1995.

[2] A. Bracy et al., "Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth," *Proc. MICRO*, 2004.

[3] A. Bracy et al., "Serialization-Aware Mini-Graphs: Performance with Fewer Resources," *Proc. MICRO*, 2006.

[4] T. Callahan et al., "The garp architecture and c compiler," *IEEE Computer,33(4):62-69*, April 2000.

[5] Y. Chou et al., "Piperench implementation of the instruction path coprocessor" *Proc. MICRO*, 2000

[6] N. Clark et al. "An architecture framework for transparent instruction set customization in embedded processors," *Proc. ISCA*, 2005.

[7] N. Clark et al., "Processor acceleration through automated instruction-set customization" *Proc. MICRO*, 2003

[8] N. Clark et al. "Application Specific Processing on a General Purpose Core via Transparent Instruction Set Customization"*Proc. MICRO*, 2004

[9] M. L. Corliss et al., "DISE: A Programmable Macro Engine for Customizing Applications", *Proc. ISCA*, 2003

[10] B. Fahs et al., "Performance characterization of a hardware mechanism for dynamic optimization" *Proc. MICRO*, 2001

[11] M. R. Guthaus et al."MiBench: A free, commercially representative embedded benchmark suite", *Work. Workload Characterization*, 2001

[12] L. Hammond et al., "A Single-Chip Multiprocessor," *IEEE Computer*, Volume 30, No. 9. Sept. 1997.

[13] S. Hauck et al., "The chimaera reconfigurable functional unit," *Proc. FCCM*, 1997.

[14] S. Hu et al. "An Approach for Implementing Efficient Superscalar CISC Processors," *Proc. HPCA*, 2006.

[15] S. Hu and J. Smith, "Using Dynamic Binary Translation to Fuse Dependent Instructions," *Int. Symp. on CGO*, 2004.

[16] C. Iseli and E. Sanchez, "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," Jun. 2003. http://www.intel.com/design/mobile/datashts/250686.htm.

[18] J. A. Jacob and P. Chow, "Memory interfacing an instruction specification for reconfigurable processors," *Symp. FPGAs*, 1999.

[19] I. Kim and M. Lipasti, "Macro-op Scheduling: Relaxing Scheduling Loop Constraints," *Proc. MICRO*, 2003.

[20] R. Kumar et al., "Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling," *Proc. ISCA*, 2005.

[21] C. Lee et al., "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems", *Proc. MICRO*, 1997

[22] J. Lotz et al., "A Quad-Issue Out-of-Order RISC CPU," *Proc. Int'l Solid-State Circuits Conf.*, 1996.

[23] T. Miyamori and K. Olukotun, "Remarc: Reconfigurable multimedia array co-processor," *IEICE Trans. on information and systems, E82-D(2):389-397*, 1999.

[24] K. Olukotun et al., "The Case for a Single-Chip Multiprocessor," *ASPLOS*, 1996.

[25] Sun Microsystems, Inc. "OpenSPARC T1 Micro Architecture Specification," *Sun Microsystems, Inc.*, 2006.

[26] R. Razdan and M. Smith, "A high-performance microarchitecture with hardware-programmable functional units," *Proc. MICRO*, 1994.

[27] C.R. Rupp et al., "The napa adaptive processing architecture," *Proc. FPGAs for computing machines*, 1998.

[28] P. Sassone and D. Wills, "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication," *Proc. MICRO*, 2004.

[29] H. Singh et al.,"Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. on Computers, 49(5): 465-481*, 2000.

[30] T. Suri and A. Aggarwal,"Scalable Multi-cores with Improved Per-core Performance using Off-the-critical Path Reconfigurable Hardware," *Proc. HiPC*, 2008.

[31] "TSMC 90nm Core Library - TCBN90GHP", *App. Note - Revision 1.2*, 2006

[32] S. Vassiliadis et al. "The molen polymorphic processor," *IEEE Trans. on Computers*, Vol. 53, Issue 11, 2004.

[33] R. Wittig and P. Chow, "Onechip: An fpga processor with reconfigurable logic," *Proc. FCCM*, 1996.