# Improving Performance of Simple Cores by Exploiting Loop-Level Parallelism through Value Prediction and Reconfiguration

Tameesh Suri
ECE Department
State University of New York at Binghamton
Binghamton, NY 13902 USA
tameesh@binghamton.edu

Aneesh Aggarwal
ECE Department
State University of New York at Binghamton
Binghamton, NY 13902 USA
aneesh@binghamton.edu

## ABSTRACT

There is a growing trend towards designing simpler CPU cores that have considerable area, complexity, and power advantages. These cores are then leveraged in large-scale multicore processors or in SoCs for hand-held devices. The most significant limitation of such simple CPU cores is their lower performance. In this paper, we propose a technique to improve the performance of simple cores with minimal increase in complexity and area. In particular, we integrate a Reconfigurable Hardware Unit (RHU) that exploits loop-level parallelism to increase the core's overall performance. The RHU is reconfigured to execute instructions with highly predictable operand values from the future iterations of loops. Our experiments show that the proposed architecture improves the performance by an average of about 51% across a wide range of applications, while incurring a area overhead of only about 5.6%.

## Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiprocessors

## General Terms

Design

## Keywords

Dynamic Reconfiguration, Loop Level Parallelism, Data Value Prediction

## 1. INTRODUCTION

Recently, there has been a major shift in the microprocessor industry towards simpler cores that are integrated in multi-core processors or SoCs in hand-held devices. The shift in the design paradigm has been fueled by the higher power consumption and diminishing returns on investment for complex high performance cores. However, a simple core with fewer resources available to it degrades the performance of each thread of execution [20].

An attractive way to improve a core's performance is to exploit loop-level parallelism (LLP), *i.e.* executing multiple iterations of a loop in parallel [30]. A high-performance core exploits LLP through the capability of maintaining large number of in-flight instructions. This capability is not available to simple cores because they use much smaller instruction windows. Other mechanisms to exploit LLP, such as software pipelining[25], loop unfolding, and modulo scheduling[30] are either limited by the lack of run-time information or are ineffective in simple cores because of the limited resource availability. Approaches that schedule different iterations of loops on different cores in multi-core processors, e.g. [27], exploit LLP at the cost of thread-level parallelism (TLP).

In this paper we propose a mechanism that improves the performance by exploiting LLP without sacrificing TLP and with minimal increase in complexity and area. The performance is improved by integrating an off-the-critical path reconfigurable hardware unit (RHU)[35] in the core's datapath. The RHU structure has only a small area overhead, operates entirely asynchronously with the core's datapath, and is reconfigured to execute instructions from the future iterations of loops in parallel to the current iteration being executed on the core datapath. The traces consist of only those instructions whose operand values are highly predictable, so as to prevent frequent misspeculation.

Our experiments show that the average performance improves by about 51% across a wide variety of applications, while incurring an area overhead of only about 5.6%. It is highly unlikely that the transistors expended for the approach proposed in this paper can give such high return on investment with any other performance-improving mechanism. However, such a study is out of the scope of this paper.

## 2. CORE DESIGN

### 2.1 Basic Idea

Figure 1 shows the basic idea behind the proposed approach, where the RHU executes future iterations of loops. For instance, in Figure 1, the RHU runs two iterations ahead of the core. Only those instructions within a loop are executed on the RHU whose operands are highly predictable or


*CF'09,* May 18–20, 2009, Ischia, Italy.

are produced by other instructions executing on the RHU. A small stride-based data value predictor is used to predict instructions' operand values. Instructions executed on the RHU are not executed on the core's datapath. When the core reaches an iteration, a part of which has already been executed on the RHU, the operands of many of the remaining instructions in that iteration are already available. This further speeds up the execution of those instructions on the core. Meanwhile, the RHU moves on to the next iterations, as shown in the figure. As the RHU keeps executing future iterations, it keeps writing the values that will be consumed by the instructions executing on the core in a small buffer. The maximum number of instructions that can be executed on the RHU is limited to 70% of the total number of instructions in a loop. This is done to avoid executing so many instructions on the RHU that the core is almost idle. Our experiments showed that the best performance was achieved for different benchmarks, when the number of instructions executed on the RHU was between 60% and 70% of the loop instructions.

The instructions to be executed on the RHU are selected from within the inner-most loops in the applications, starting from the first instruction in the loop. The trace is truncated when a control or store instruction cannot be included in the trace, as explained later, or when an instruction that has been identified to be a part of the trace cannot be mapped onto the RHU, or when the maximum limit of RHU instructions has reached, or the RHU is fully occupied, or the loop-end has been reached. On an exception from within the trace or when a predicted input source operand value incurs misprediction, the execution is restarted from the start of the trace and is performed entirely on the core's original datapath.

When executing future loop iterations on the RHU, the outcomes of the branches in that iteration are unknown to the RHU. Hence, the traces are formed and mapped onto the RHU one basic block at a time. Once the reconfiguration bits for a basic block are generated, they are mapped onto the RHU. Trace formation is stopped when a control instruction cannot be included in the trace because when executing future iterations, the control flow within those iterations is not known. An alternative could be to provide an additional branch prediction unit to predict such branches. However, this would increase the area and complexity, and hence we stall traces on unincluded branch instructions. If the control path taken within the loop changes, the reconfiguration bits for the new basic blocks are generated and mapped onto the RHU.

## 2.2 Core Microarchitecture

We call the original instructions in 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.

**RHU Structure:** To keep the RHU structure simple, only integer ALU operations, including ld/st address generations, are performed on the RHU. The RHU structure is a two-dimensional array of interconnected ALUs. We use multiple serially connected simpler RHU structures (such as *2x2 RHU, 3x3 RHU*). A 2x2 RHU has 2 rows and 2 columns of Integer ALUs, as shown in Figure 2. A basic block is mapped onto a fresh RHU and can utilize multiple RHUs. However, on a branch instruction, the use of that RHU is stopped, and the target basic block starts on the serially next RHU. In the proposed RHU structure, instructions in the first row are provided with two live-ins per ALU and second row has one live-in per ALU. Instructions in both rows can produce live-outs. The live-outs from one RHU are multiplexed with the live-in ports into the inputs of the serially next RHU. 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.

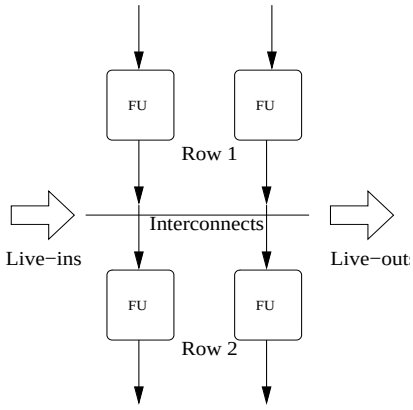



**Figure 2: Block diagram of a 2x2 RHU**

**Trace Detection, RHU Reconfiguration, and Trace Execution:** Figure 3 shows the integration of the RHU with the core data-path. When a looping-back branch commits, its target PC is compared with that of the previous loop-back branch. If there is a match, an inner-most loop is detected. Once, an inner-most loop is detected, the *Data-Value predictor*, discussed later, starts warming up by recording the operand values of the instructions for the next few iterations.

After the data value predictor is warmed up and detect instructions with highly predictable operands, the trace is formed and reconfiguration bits are generated for it, as discussed later. Reconfiguration bits are generated for one basic-block at a time. As the reconfiguration bits are generated, they are dispatched to reconfigure the RHU. In parallel, the reconfiguration bits generation for the next basic block, decided based on the current outcomes of the branch instruction ending the previous basic block, is started. Once, all the RHUs are occupied, or the maximum limit of RIs is reached, or a branch instruction cannot be mapped onto the RHU, or loop finishes, the reconfiguration bits generation is stopped. The RHU is reconfigured by reusing the live-in ports in the RHU that are used to provide the predicted operand values, as shown in Figure 3.

The data value predictor keeps predicting the values while the traces are being formed. Once, the trace formation is completed, and the entire trace is mapped onto the RHU, the traces are executed on the RHU using the predicted values. A ready bit is supplied along with each predicted value. This ready bit flows through the RHU, "waking" up the instructions mapped in the RHU. When the ready bit of a live-out is set, that live-out is ready to be consumed. When the live-outs that will be used by PIs become ready they are

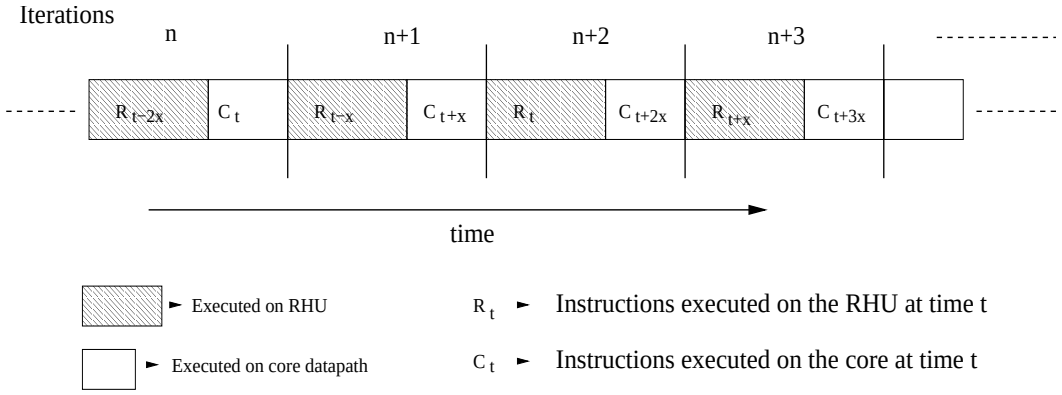



**Figure 1: Basic idea**

stored into a FIFO as shown in Figure 3. The live-outs are stored in the program order in the FIFO. In addition, a bit-vector (*instruction sequencer*) is provided that maintains the record of instructions executed on the RHU. When an instruction is included in the trace, its associated bit in the bit-vector is set. If an instruction is not included in the trace, its associated bit is reset. Note that the bit-vector needs to maintain this sequence for only those instructions that lie in the control flow followed within the loop. This bit-vector changes, only when the control flow within the loop changes and different basic blocks are included in the trace.

When the core reaches an iteration, it fetches all the instructions. However, in the decode stage, it discards the instructions whose bits are set, i.e. they have been executed on the RHU. But, before dropping the instruction, the core copies the value produced by that instruction from the FIFO into an additional integer architectural register file (AARF) shown in the Figure 3. The valid bit for that register in AARF is set to valid. If an instruction is going to be executed on the core, the valid bit of the register produced by that instruction is reset in AARF. When an instruction to be executed on the core encounters a valid register in AARF for its operand, it carries the value of the register to the scheduler. This only increases the width of the payload RAM portion of the scheduler, but does not impact the CAM portion of the scheduler. Furthermore, the core uses the outcome of the branches executed by the RHU, instead of the prediction done by the branch predictor.

On an exception from within the trace, or a load-miss, or a live-in value misprediction, the execution restarts from the start of the trace, and is performed entirely on the core's datapath. On a value misprediction, the trace is discarded and the entire process is repeated. Hence, only highly predictable instructions are included in the trace. When the control flow changes within a loop, that iteration is discarded, and again the process is repeated from the branch that changed its flow. The PI fetched immediately following the last instruction in the trace is marked so that when it commits, the branch predictor is updated for the branch instructions within the trace. As the trace instructions commit, the corresponding live-out registers from AARF are copied into the core's architectural register file. The RHU always maintains a fixed distance ahead of the core.

**Load/Store Instructions:** Only one load/store instruction is mapped in a row. When mapping instructions onto the RHU, the program order of loads and stores is maintained. However, the loads can be reordered between themselves. However, if a store cannot be placed in the RHU, the trace is truncated when the next load is encountered because memory disambiguation cannot be performed for this load. When stores execute on the RHU, they write their addresses and data into a small buffer. Multiple such small buffers are provided to maintain the iteration-distance between the core and the RHU. When a load executes, it not only accesses the cache, but also scans the current buffer used for store-to-load forwarding. As the store instructions of the trace commit, the values are written from the buffers into the cache. The accesses to the cache from the core and the RHU are multiplexed, however, the accesses from the RHU are given higher priority. If a load misses in the level 1 cache, then the trace execution is discarded and the core executes the entire loop when it reaches that iteration.

**Data-Value Predictor and Live-in verification** We use a small 30-entry stride based *Data-value Predictor*[38] in the core to predict the operand values. Stride-based predictor is primarily chosen as it has a small area footprint. To restrict mispredictions, only operands with very-high prediction accuracy are chosen as live-ins. To verify the predicted operand values, we also provide two additional (*live-in operand*) bit-vectors, one for each operand of the instructions. We assume a two-operand, one-destination RISC ISA. When a bit in the instruction sequencer bit-vector is set, and if that instruction uses a predicted live-in value, then the associated bit in the live-in operand bit-vector is also set. So, when the core drops instructions in the decode stage, but sees that predicted live-in values have been used by the dropped instructions, it inserts an instruction – check rhu prediction: *chkpred Rs, Rt.* When an chkpred instruction commits, it reads the registers Rs and Rt from the architectural register file and compares it with the predicted value from the data value predictor. To enable the operand prediction validation, the data value predictor is locked for a particular control flow within the loop once the traces using predicted values. The value predictor is not unlocked till all the predictions for that control flow have been validated.

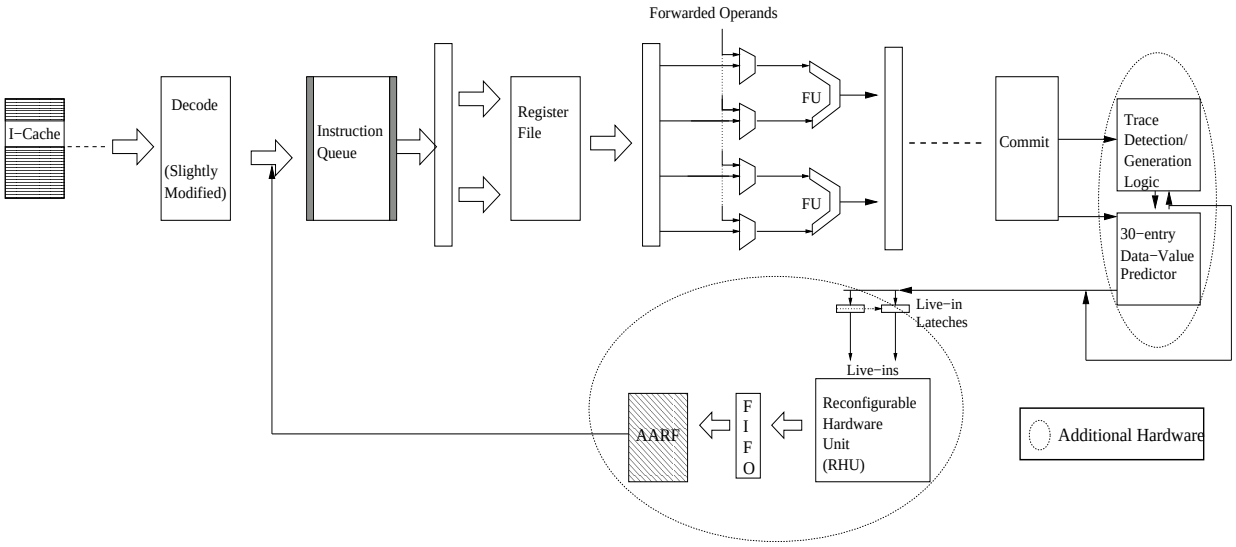


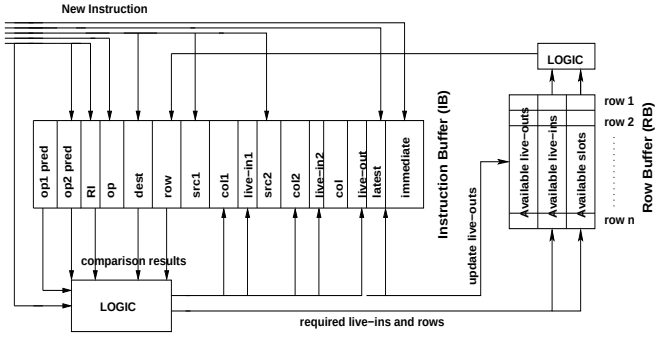

**Figure 3: Schematic diagram of the core datapath**

## 2.3 Reconfiguration Bits Generation

The reconfiguration bits generation hardware includes the *trace buffer*, which is incorporated within trace detection / generation logic as shown in Figure 3. The trace instructions are fetched, decoded, and forwarded to the trace buffer, along with sending them along the core's datapath. The details of the trace buffer hardware is shown in Figure 4. 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 several fields as depicted in Figure 4. We use a 10-entry instruction buffer, as we form traces one basic block at a time. 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 analyzed at a time.

The RHU reconfiguration bits are generated in two phases as shown in Figure 5. In phase 1 (Figure 5(a)), the decoded information of trace instructions are written into the trace buffer, one instruction at a time. While placing the instructions, the operation bits, source register identifiers, and immediate values are accordingly filled. The *RI bit* is set for RIs and the *latest bit* is set for RIs with valid destination. *op1 pred* and *op2 pred* bits are set for instructions with respective predictable source operands. 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 new entry. The *live-in* bits of the operands that do not match are set only if the respective operand is predictable. If the operand is not predictable, the instruction is marked as PI.

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, serially next RHU can be used. If no RHUs are available, the instruction entry is invalidated and phase 1 is halted. Phase 1 also halts when any of the other conditions discussed earlier for trace




op1 pred −− indicating whether the source operand 1 in predictable (1 bit)
op2 pred −− indicating whether the source operand 2 in predictable (1 bit)
RI −− indicating whether the instruction is a RI or a PI (1 bit)
op −− the operation control bits for the instructions (5 bits)
destination −− destination register identifier (5 bits)
src1 and src2 −− source register identifiers (5 bits each)
col1 and col2 −− RHU columns for RIs' two source registers' producers (4 bits each)
live−in1 and live−in2 −− indicating the live−in status of the two source registers (1 bit each)
live−out −− indicating whether the destination is a live−out (1 bit)
latest −− indicating the most recent RI writing to a destination (1 bit)
row and col −− the row and column assigned to the instruction (4 bits each)
immediate −− the immediate operand value (if any) (16 bits)

Total for each entry 54 bits

**Figure 4: Trace Buffer Hardware**

truncation are met. Phase 1 for each instruction requires 3 cycles per instruction.

Phase 2 starts after phase 1 and operates only on the instructions in the instruction buffer. In Phase 2, the live-outs are marked. In addition, the FIFO entries to be written with the live-out values are also marked.

Figure 6(a) shows an example of innermost loop taken from `bzip2` benchmark. B1, B2 and B3 are forward-looping branches within innermost loop. The highlighted path shows the branches and basic-blocks included. Figure 6(a) also shows the instructions included within the top most basic-block. The block requires one live-in, R8, and has two live-outs, R8 and R2. Figure 6(b) shows the chkpred instruction,

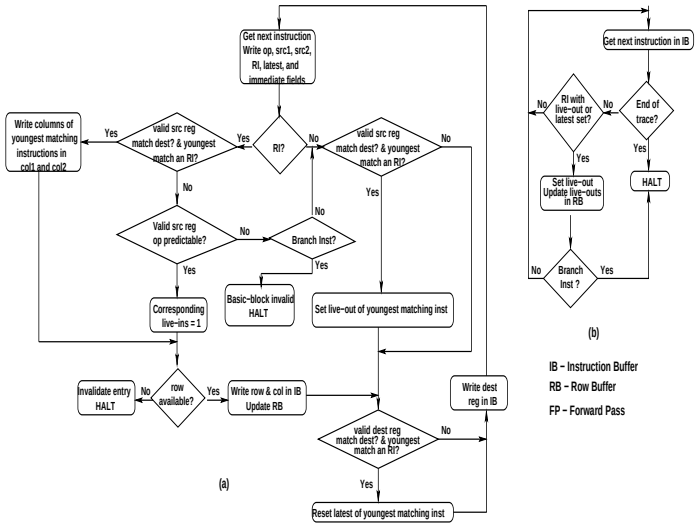



**Figure 5: (a) Phase 1; (b) Phase 2**

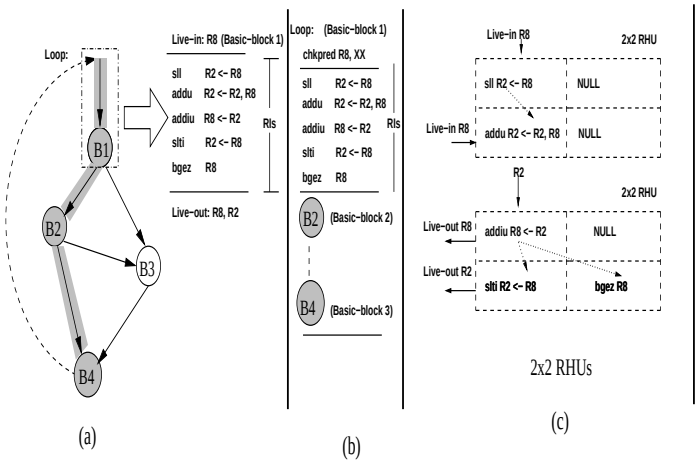



**Figure 6: An example illustrating a basic-block layout on two 2x2 RHUs**

which is fetched instead of RIs (marked in the example). Finally, figure 6(c) shows the layout of the basic block on two 2x2 RHUs.

# 3. EXPERIMENTAL RESULTS

## 3.1 Experimental Setup

We experiment with a simple RISC core using Portable Instruction Set Architecture (PISA), modeled on a modified Simplescalar simulator [6]. The hardware features and default parameters of each core are given in Table 1. The core consists of separate integer and floating point subsystems, with 1-cycle inter-subsystem communication latency. 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 core resources are constrained to make them similar to those used in the current multi-core implementations. 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 [19], making the reduced-complexity RHU feasible for processors with a wide range of operating frequencies.

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 [26] (`epic`, `g721`, `mpeg2`), and 9 MiBench [17] (`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 ten benchmarks (`applu`, `bzip2`, `gcc`, `mcf`, `vpr`, `sha`, `dijkstra`, `adpcm` `CRC32`, and `FFT`).

In this paper, we experiment with two prediction accuracy schemes; *100pred* – where instruction operands with prediction accuracy of 100 percent are candidates for inclusion in the trace, and *80pred* – where instructions may be included with operand accuracy of 80 percent or more.

## 3.2 Classification of Loop Instructions for RHU Design

To determine the size and number of the RHUs to be used, we studied the amount of control and memory dependencies within the inner-loop. Figure 7(a) shows the average number of branches, loads and stores that exist within the innermost loop along with average instructions included using 100pred and 80pred schemes. Figure 7(a) shows that the average number of branches, loads, and stores in an innermost loop are 4.8, 4.5 and 2.8 respectively. The extreme exceptions were `applu` and `mgrid` (not shown in the figure) that do not have any branches in the innermost loops simulated. Hence, the loop bar is zero for `applu`. On the other hand, `adpcm`, `vpr` and `mesa` had more branches than the average. However, the average number of branches, loads, and stores included in the traces are about 1, 2, and 1.5, respectively. Number of instructions included on an average marginally increases with 80pred over 100pred scheme.

Figure 7(b) shows the average size of basic blocks within an innermost loop. The average is about 5 instructions with `applu` and `mgrid` as exceptions, which have an average of 46 and 52 instructions, respectively. Note that these benchmarks do not have any branches within the loops. Based on the results of Figure 7(b), we experiment with three different RHU structures, *2x2 RHU*, *3x3 RHU*, and *AsymmRHU*, which has 4 ALUs in top row, 3 ALUs in second row and 2 ALUs in third row. We keep equal number of total ALUs in *3x3 RHU* and *AsymmRHU*. Based on the results of Figure 7(a), we experiment with six, eight, and ten sets of above mentioned RHUs connected in a serial fashion.

Figure 8(a) presents overall RHU utilization averaged across the benchmarks. RHU utilization is described as the number of ALUs occupied divided by the total number of ALUs. It is evident from the figure 8(a) that 2x2 RHUs have the highest utilization, which is as expected. Furthermore, six RHUs have the highest utilization. Figure 8(b) presents average number of RHUs used per innermost loop and basic-block with varying degree of RHU structure. Providing infinite RHU resources, we use on an average 9 RHUs per loop. However, by restricting the number of RHUs to 6, the av-

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| *Fetch/Commit Width* | 4 instructions | *Instr. Window Size* | 16 Int/16 Mem/16 FP instructions |
| *ROB Size* | 96 instructions | *Issue Width* | 2 Int/2 Mem/2 FP |
| *Speculative Register File* | 48 Int/48 FP, | *Int. Functional units* | 2 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**

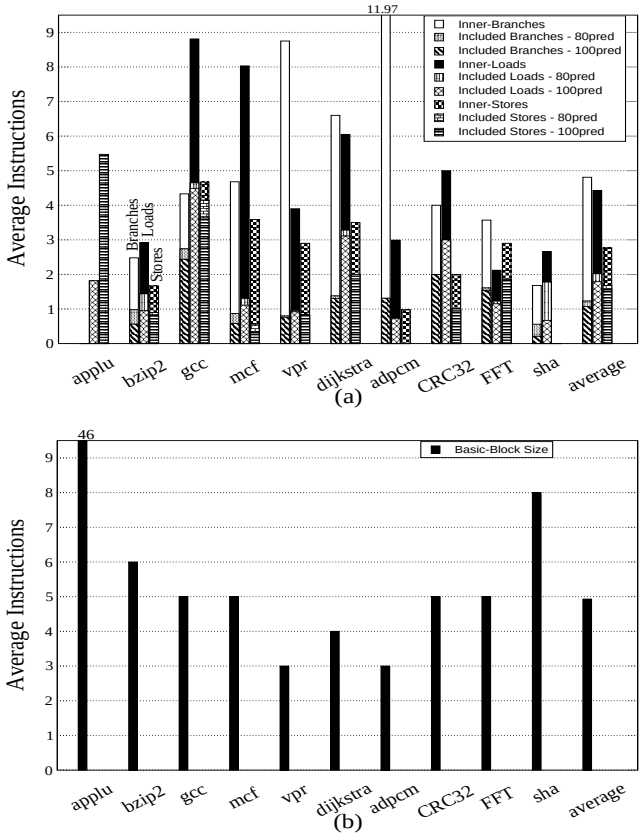



**Figure 7: (a) Average Number of existing and potential Branches, Loads and Stores within innermost loop (b) Average instructions in basic-blocks**

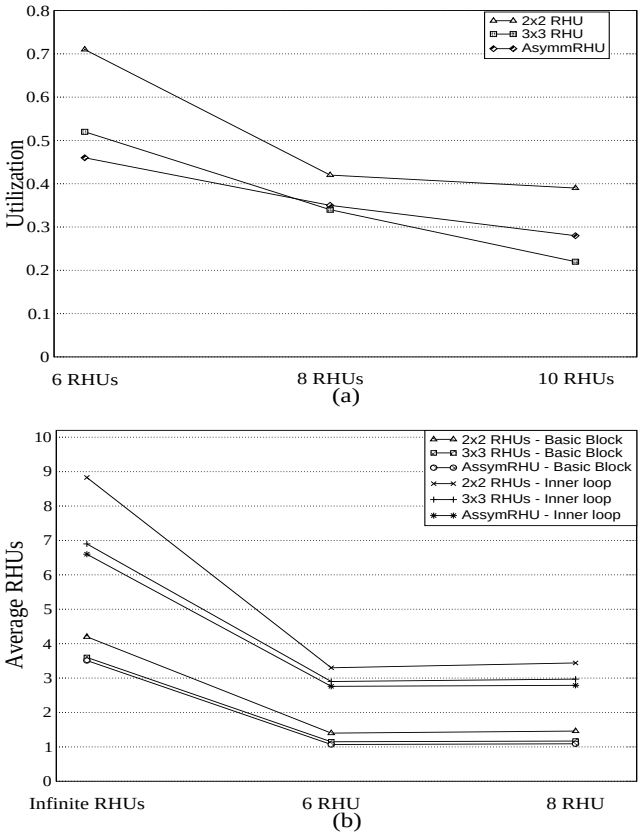



**Figure 8: (a) RHU row utilization for different RHU structures (b) Average Number of RHUs used for inner-loops and basic-blocks**

erage number of 2x2 RHUs used per loop is around 4.3 and basic block is around 1.4. The average number of RHUs increases marginally when eight 2x2 RHUs are provided.

The choice is really between having more 2x2 RHUs or fewer 3x3 or AsymmRHUs. However, the 2x2 RHUs are much simpler to design than 3x3 and AssymRHUs, and have less area and interconnect overhead. Hence, for the rest of the paper, we present results with six 2x2 RHUs connected in a serial fashion. Across all the benchmarks, we found that over 80% of loop instructions can be accomodated using six 2x2 RHUs.

## 3.3 Area Results

We integrated six 2x2 RHUs with one SUN T1 Open-Source core [28] of an eight-core processor, to get the area requirement. The design for the RHU and reconfiguration bits generation hardware was synthesized using Synopsys Design Compiler using a TSMC 90nm Standard cell library [37], and was placed and routed using Cadence SoC Encounter. After integrating the additional hardware, the core area increased by about 5.6%. Figure 9 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. Figure 9 shows that the RHU is placed close to

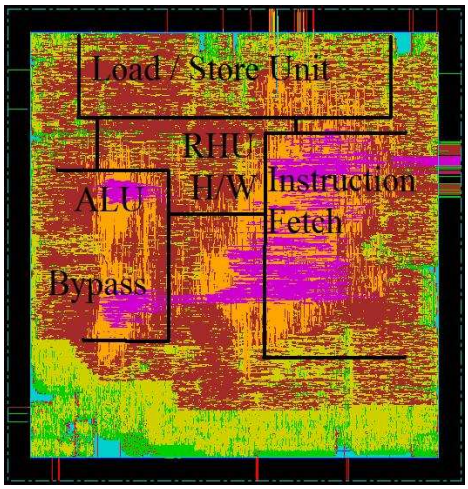



**Figure 9: Die image of a Core**

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

## 3.4 Complexity Analysis

The only modification to the core pipeline include a slightly modified decode, which will incur an additional gate-delay to decide if an instruction will execute on RHU or the superscalar. An extensive previous study [29] shows that issue-logic and data bypass are decisive in the overall processor clock speed. It is important to note that our approach does not impact the scheduler portion of the CAM. Synthesis and place-and-route confirmed that the overall cycle-time of the processor remains unchanged.

## 3.5 Operand Prediction Results

Figure 10 shows the prediction accuracy for predicted operands using 100pred is 98.6% and using 80pred is 97.4%. For most of the benchmarks, the prediction accuracy drops by about 2% from 100pred to 80pred, however for `sha`, `epic` and `vortex` the accuracy drops by about 4%, 6.4% and 5.8% respectively. Lowering the prediction counter we are able to target more instructions (discussed later). The performance improvement, however, is subjected to the percentage increase in RI instructions and the effective mispredictions.

**RHU Coverage:** We observed that the RIs formed about 17% fraction of the overall instructions executed for the six 2x2 RHU using 100pred, and about 21% using 80pred. 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.

**RHU-Consumer Coverage:** It is important to note that the consumers of RHU-instructions do not wait in the IQ for their operands to be ready. This also significantly increases the overall performance. We observed that the direct consumers of the RHU-instructions formed about 13% fraction of the overall instructions executed for the six 2x2 RHU using 100pred, and about 17% using 80pred. This decreases the overall time the entire consumer-chain spends within the IQ.

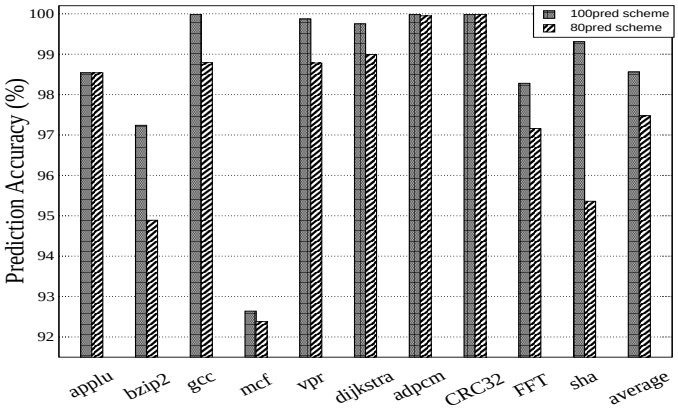



**Figure 10: Prediction accuracy for 100pred and 80pred**

## 3.6 Instruction and Operand distribution of RIs

Figure 11(a) presents distribution of dependent and predictable operands and instructions which form RIs. Note that only those instructions are included as RIs that either have predictable operands or have their operands produced within the trace. On an average, about 58% of operands for RIs are predictable and the remaining 42% are produced within the trace. Instruction-wise distribution suggests that roughly 36% of instructions are completely dependent on instructions included within the RHU. Another 46% of instructions included are completely predictable, and the remaining 18% instructions have one operand dependent upon another RI instruction and the other is predictable.

Figure 11(b) shows that one-operand instructions dominate RIs, averaging at 64% of total RIs. Two-operand instructions form another 29% of RIs and the remaining 7% are zero-operand instructions such as `lui`. One-operand instructions use predicted values 65% of the time, whereas two-operand instructions are completely predicted 20% of the time. Two-operand instructions require a dependent and predicted value 62% of the time.

## 3.7 Performance Results

Next, we present the performance (IPC) improvement with six 2x2 RHUs, over the base processor. The IPC speedup is shown in Figure 12. Figure 12 shows that average performance improves by about 39% by using 100pred scheme, and by about 51% using 80pred scheme. Our approach performs almost the same using 100pred and 80pred scheme for `applu`, `gcc`, `adpcm`, `CRC32` and `FFT`. The prediction accuracy using both scheme for `applu`, `adpcm` and `CRC32` remains the same, and so did the RHU coverage, explaining the same speedup. However, for `gcc` and `FFT` the prediction accuracy drops by about 1% and the coverage increases by about 0.6%, explaining a very marginal increase in performance. A few benchmarks, such as `sha`, `susan` and `epic` show negligible performance improvement with 100pred scheme, but achieve a speedup of 82%, 190% and 41% with 80pred scheme, respectively. 100pred scheme was too restrictive to form traces for these benchmarks, and the coverage increased by about 28% on an average across these benchmarks with 80pred scheme. A slight performance drop across a few benchmarks, such as

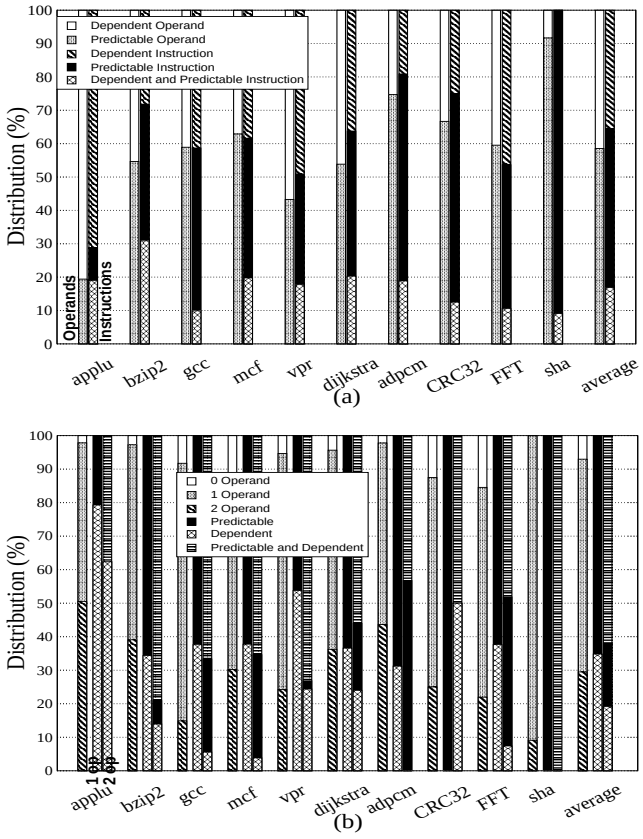



**Figure 11: (a) Distribution of dependent and predictable operands and instructions (b) Distribution of operand-wise dependent and predictable instructions**

`dijkstra` and `mcf` was also recorded using 80pred scheme. This is due to the fact that re-execution was more due to prediction accuracy drop in these benchmarks, whereas in other benchmarks, re-execution due to lower prediction accuracy was hidden due to much higher RHU coverage achieved by 80pred scheme.

We also compared the performance of our approach with that of double-sized integer and memory schedulers. Double-sized scheduler configuration doubles the issue queue size, the issue width, and also increases the number of functional units for the integer and memory sub-systems. As double-sized scheduler achieved a performance improvement of about 30% over the base machine, compared to a 51% improvement achieved with our approach. 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, by not executing instructions on the core datapath. Furthermore, our approach pre-executes instructions using predicted values, thus reducing the data-dependent related stalls. It is also important to note that doubling the schedulers for better performance will have much higher impact on the on complexity, area, and power consumption because of increases in scheduler size and additional functional units, forwarding paths and register file ports.

Figure 13 presents IPC speedup achieved with our approach over various machine configurations. Functional units

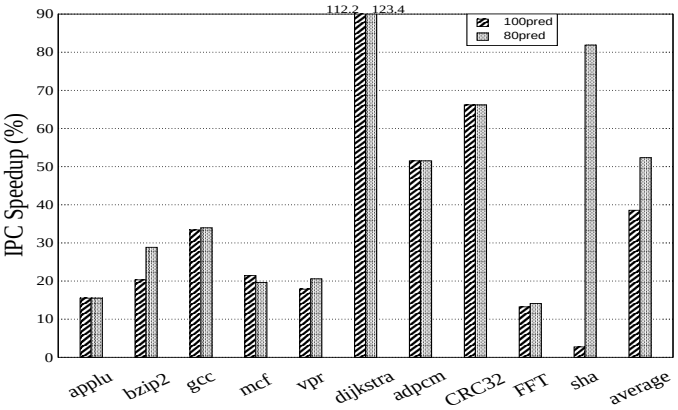



**Figure 12: IPC Speedup over base machine**

are adjusted accordingly in the base machine configuration. The configuration can be read using the following format – Rob.*aa*_Width.*b.c.d*_Queue.*e.f.g* – where *aa* is the Rob and Register File (addition of integer and floating-point) size, *b, c, d* are issue width for Integer, Floating-point and memory subsystems, and *e, f, g* are issue queue size for the three subsystems. Figure 13 shows that our approach gives significant performance improvement for wide range of configurations.

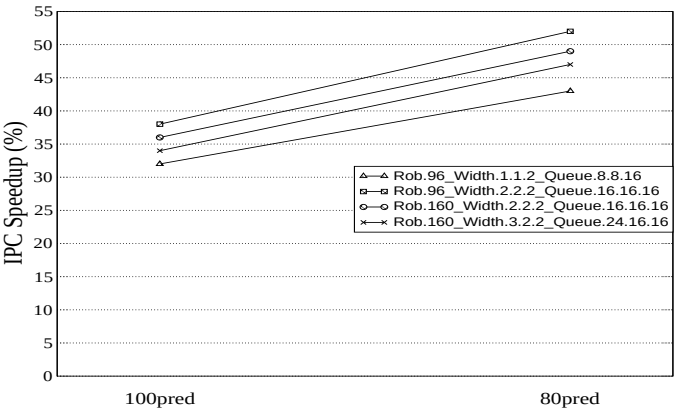



**Figure 13: IPC Speedup trend for different machine configurations**

**Reduction in power consumption:** Our approach is also expected to reduce dynamic power consumption by reducing the switching activity in the core's datapath. We measured the activity in the register files and the dynamic scheduler, which represent a substantial portion of energy dissipation in present day out-of-order superscalar microprocessors. For instance, register file in Motorola M.CORE architecture consumes 16% of the total processor power[15]. Figure 14 shows the overall reduction in read and write accesses to register file and in the register identifier broadcasts in the issue queue to wakeup dependent instructions. The results are shown for both, 100pred and 80pred scheme. On an average, our scheme reduces the register file accesses by 29-35% and instruction wakeup broadcasts by about 31-34%. Note that the reduction in the read/write accesses to the scheduler is approximately equal to the RHU coverage, discussed

earlier, because these instructions are not executed on the core's datapath.

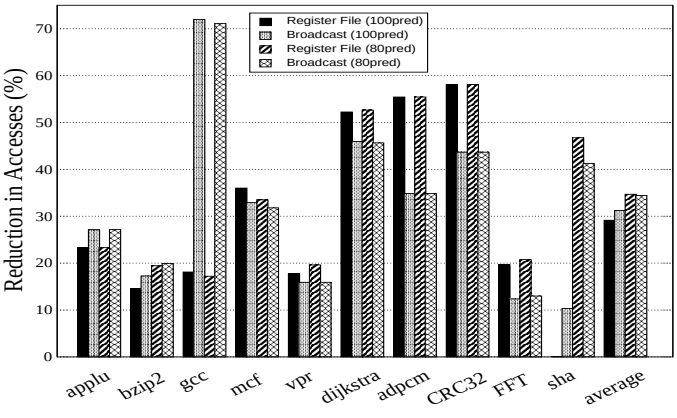



**Figure 14: Reduction in Register File and instruction wakeup accesses**

# 4. RELATED WORK

**Loop-level Parallelism:** Other approaches, proposed in the past, to exploit LLP are software pipelining [25], loop unfolding, and modulo scheduling[31]. These approaches require compiler involvement and large on-chip resources to be beneficial; hence, they are not suitable for small cores. For instance, software pipelining and modulo scheduling place high pressure on register files, making them inefficient for cores with fewer registers.

Thread-level speculation (e.g.[21]) can also be used for optimistic execution of code regions before all values are known. TLS is somewhat similar to our approach if each thread executes a different iteration of a loop. However, high hardware complexity and area overhead is incurred in TLS to track dependence violations and architectural state rollbacks. Furthermore, TLS consumes many processing elements in a processor [33], effectively having significant impact on the processors capability to exploit thread level parallelism. TLS is thus infeasible for small cores used in SOCs and multi-cores where both TLP and LLP may be simultaneously exploited.

Clark et. al [12] propose a virtual framework for dynamically mapping modulo-scheduled loops across loop accelerators. The loop accelerator contains various functional units and has a much higher area overhead compared to our proposed design. It is very evident that such a loop accelerator may not be beneficial for exploiting LLP and TLP, as it will reduce the number of cores.

Our approach alleviates the limitations, which are primarily hardware complexity and area overhead, incurred by the previously proposed approaches. Our approach also does not impact the cycle-time of the critical path.

**Reconfigurable units:** A few efforts have attempted to integrate FPGA modules with an all-purpose processor. PRISM [2], Spyder [24], 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. Chimaera [18], PRISC [32], and OneChip [39] integrate the FPGA as a functional unit (RFU) in the processor datapath with direct access to the processor register file. Tightly integrating an FPGA with a core will have significant area, power, and reconfiguration time overhead because of their highly general reconfigurability, and cannot be used for dynamic reconfiguration for exploiting LLP. Previous studies [11, 9, 35] propose a restrictive reconfigurable hardware that is either truncated at branch or memory instructions or can be used only for in-order processors, severely limiting its applicability. Authors in [36] execute memory instructions on reconfigurable hardware for out-of-order cores. However, all mentioned proposals attempts to exploit instruction level parallelism (ILP) rather than LLP.

Non-reconfigurable custom functional units have been extensively used in application specific processors by identifying portions of an applications data flow graph that can be efficiently implemented in hardware [1, 5, 10, 16, 22, 34, 23]. The authors in [3, 4, 13] also propose fusion of instructions in a dependence chain to form a mini-graph and replacing the mini-graph with a special instruction, which is then executed on a custom (not reconfigurable) function unit. This mechanism is used to improve the instruction level parallelism, and not loop level parallelism, in constrained cores.

Commit time trace formation has also been proposed to improve the fetch bandwidth and perform dynamic optimizations in superscalar processors [14]. However, the reconfiguration instruction generation in our approach is significantly different from the trace formations for superscalar processors.

# 5. CONCLUSION

Simpler cores have become more attractive because of their area, complexity, and power consumption advantages. Furthermore, these simple cores can be leveraged to build large multi-core processors as well as to build SoCs for handhelp PDAs and smart phones. However, the major limitation of simple cores is their lower performance. In this paper, we propose a mechanism to alleviate the lower performance limitation of these simple cores by integrating small reconfigurable units that exploit loop-level parallelism. The RHU is reconfigured to execute future iterations of inner-most loops using predicted operands. Only those instructions are executed on the RHU whose operands are highly predictable or are dependent on other instructions being executed on the RHU. We also 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 51% with about 5.6% area overhead.

# 6. REFERENCES


[1] K. Atasu et al., "Automatic application-specific and instruction-set extensions under microarchitectural constraints" *DAC*, 2003

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

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

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

[5] P. Brisk et al., "Instruction generation and regularity extraction for reconfigurable processors" *CASES*, 2002

[6] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Arch. News*, June 1997.

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

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

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

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

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

[12] N. Clark et al. "VEAL: Virtualized Execution Accelerator for Loops"*Proc. ISCA*, 2008

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

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

[15] D. R. Gonzales, "Micro-RISC architecture for the wireless market" *IEEE Micro*, 19(4):30-37, 1999

[16] D. Goodwin and D. Petkov, "Automatic generation of application specific processors" *CASES*, 2003

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

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

[19] S. K. Hsu et. al., "An 8.3GHz Dual Supply/Threshold optimized 32b integer ALU-register file loop in 90nm CMOS" *ISLPED*, 2005

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

[21] L. Hammond et. al., "Data speculation support for a chip multiprocessor," *ACM TACO*, 2(3):247-279, 2005.

[22] I. Huang and A. M. Despain, "Synthesis of application specific instruction sets" *IEEE TCAD*, 1995

[23] Z. Huang et. al., "Design of dynamically reconfigurable datapath processors" *ACM Trans. on Embedded Computing Systems*,Vol 3, No. 2 2004

[24] C. Iseli and E. Sanchez, "Spyder: a sure (superscalar and reconfigurable) processor," *Journal of Supercomputing, 9(3):231-252*, 1995.

[25] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines.," *Proc. PLDI*, 1988.

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

[27] S. Lieberman et al., "Extracting Statistical Loop-Level Parallelism using Hardware-Assisted Recovery", *University of Michigan CSE Technical Report, CSE-TR-528-07*, 2007

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

[29] S. Palacharla et al., "Complexity-Effective Superscalar Processors," *Proc. ISCA*, 1997.

[30] B. Rau et. al., "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proc. 14th Annual Microprogramming Workshop*, 1981.

[31] B. Rau., "Iterative modulo scheduling: An algorithm for software pipelined loops," *Proc. Micro*, 1994.

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

[33] G. Sohi et al., "Multiscalar processors" *Proc. ISCA*, 1995

[34] F. Sun et al., "Synthesis of custom processors based on extensible platforms" *ICCAD*, 2002

[35] T. Suri et al., "Scalable Multi-cores with Improved Per-core Performance using Off-the-critical Path Reconfigurable Hardware" *Proc. IEEE International Conference on High Performance Computing*, 2008

[36] T. Suri et al., "Improving Scalability and Per-core Performance in Multi-cores through Resource Sharing and Reconfiguration" *Proc. IEEE International Conference on VLSI Design*, 2009

[37] "TSMC 90nm Core Library - TCBN90GHP", *Application Note - Revision 1.2*, 2006

[38] K. Wang et al. "Highly accurate data value prediction using hybrid predictors," *Proc. Micro*, 1997.

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