Energy-Aware Opcode Design

Balaji V. Iyer [#1], Jason A. Poovey [*2], Thomas M. Conte [#3]

[1]bviyer@gatech.edu [2]japoovey@ncsu.edu [3]conte@cc.gatech.edu

[#]School of Computer Science, College of Computing
Georgia Institute of Technology, Atlanta, GA

[*]Department of Electrical & Computer Engineering
North Carolina State University, Raleigh, NC

***Abstract*— Embedded processors are required to achieve high performance while running on batteries. Thus, they must exploit all the possible means available to reduce energy consumption while not sacrificing performance. In this work, one technique to reduce energy is explored to intelligently design the instruction-opcodes of a processor based on a target-workload. The optimization is done using a heuristic that not-only minimizes switching between adjacent instructions, but also simplifies the decoding to reduce latches to save dynamic energy. On average, an optimized opcode is able to be decoded using 40-60% less latches in the decoder. In addition, it is shown that a decoder optimized for algorithms that had similar program structure, similar data-types or similar behavior exhibited consistent patterns of energy reduction. The techniques presented in this paper yield an average 10% reduction in the total dynamic energy. It is also shown that this heuristic can be used to achieve similar results on different issue-width processors.**

I. MOTIVATION

Embedded devices are required to perform several complex tasks that were once attempted by high-performance systems [18]. Moreover, the need for portability requires these devices to use batteries as primary source for energy [1] [4] [11] [18].

One solution is to take a general-purpose processor and customize it for an embedded system [1]. These embedded processors are simpler than their high-performance counterparts and require significant assistance from the compiler for scheduling, branch-handling etc. However, unlike high-performance systems, wide-availability of compilers, assemblers, and other utilities are limited [20].

The first logical step for designing (or choosing) such processors is to define the target application. This ONE target application represents the main workload of this processor. It is generally a good assumption that this target application is one of the most frequently executed applications in this system. If this one target application is able to be run at high performance while consuming less energy, then the overall system energy is reduced.

The main concentration of this work is to provide a heuristic for intelligent-design of the instruction opcodes for an embedded processor using one application as the target (or training application). The new-opcode configuration is created by analyzing the code-generator and reducing switching among the adjacent instructions occurring in the target application. The opcodes are designed such that frequently occurring instructions are decoded easily, which reduces the internal decoder power.

Unlike previous work, which requires the superset of all benchmarks to be run on the processor to gain any power/energy reduction ([9] [29]), we prove that one benchmark is enough to provide a significant amount of energy reduction. In addition, we show that an energy-efficient opcode-design can reduce energy in the decoder and other stages of the pipelined processor. Finally, we show the effects of processor issue-width scaling on the overall power reduction using this methodology.

For this work, the compiler is selected and designed before the processor. Using this design approach, the constraints imposed by the compiler (as shown in section 2) is known ahead of time, and the processor can be designed accordingly.

The paper is organized as follows. The related works are explained in section 2. Section 3 gives a brief introduction of a Retargetable code-generator. The experimental framework and the benchmark-set are explained in section 4. Section 5 explains the project methodology. The discussion of results is given in section 6 and the paper is concluded in section 7.

II. RELATED WORK

Several works have been proposed for power and energy reduction using intelligent opcode-design. To our knowledge, the only work that closely resembles ours is by Benini et al. [2]. The authors provide a methodology to reduce power of the decoder and the fetch units by examining the adjacent instructions in the trace and designing the ISA accordingly. The authors only consider instructions with fixed-size opcode and no sub-opcode fields. We show in later sections that this is an invalid assumption for several popular embedded-systems ISA. Second, the authors of [2] assume that switching directly corresponds to power consumption. We notice that blindly minimizing switching can lead to the addition of extra latches, which can make the power-savings contributed by the reduced switching. Finally, unlike [2] we present the power savings provided by an ISA trained using one application on a wide variety of applications.

Kim & Kim [10] and Woo, Yoon & Kim [29] describe methods for reducing hamming distance between adjacent instructions. Their works fail to mention the effects of power or energy on any particular units. They report their results just on switching activity and ignore other aspects inside a processor such as latches, wire-length etc.

Cheng and Tyson [3] provide frameworks for tuning instruction-sets. They tailor the instruction-set to the target application by compiling the program and then using a reconfigurable decoder to only decode instructions that are going to be used by the processor. Our method avoids this extra reconfiguration step, yet provides very comparable results.

978-1-4244-2658-4/08/$25.00 ©2008 IEEE

Varma et al. [28] studies the power reduction of switching in the register-bus and the bypass logic for the Intel X-scale processor. They indicate that switching in the register-port increases the instruction-energy by 10%. Similarly, Haga et al. [4] explore dynamically assigning function-units to reduce switching. They show a 26% power reduction in integer ALU.

Pechanek, Larin and Conte [17] present a technique for entropy-based encodings of the ISA. The primary concern for this work is on variable-size instruction which frequently occur in DSP architecture. Kalambur and Irwin [7] study ways to reduce data-fetch energy by adding an addressing mode for the ALU instructions to access operands from memory.

Tiwari, Malik and Wolfe in [26] and Tiwari et al. in [27] describe ways to reduce power and energy by modifying the amount of switching in software. They give detailed descriptions for instruction-level power reduction techniques for a specific set of applications. They claim that opcode-distribution typically only gives energy/power reduction in the decoder. In our work, we prove that up to 30% of the energy reduction is gained from other units of the processor, not just the decoder.

III. Code Generation using Retargetable Compilers

Retargetable compilers make generating codes for multiple architectures easier by splitting the compiler tasks into architecture-independent and architecture-dependent sections. **Figure 1** shows components of the most popular Retargetable compiler, the GNU Compiler Collection (GCC).

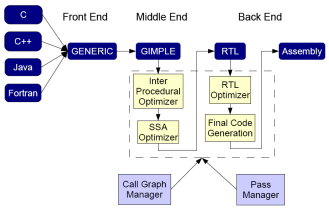

Figure 1: Components of the GNU Compiler [14]

GCC can be configured to accept several sources and provide executables for the wanted target architecture. The source code is parsed using an appropriate parser by the front-end, and converted to an intermediate language format, called GIMPLE, that is language-independent (GENERIC is a subset of GIMPLE). GIMPLE retains much of the structure of the parse-trees. GIMPLE is then translated to the three-operand, machine-independent format, Register-transfer-language (RTL).

Instruction scheduling is done on the RTL instructions. For this work, an aggressive scheduler using Treegion scheduling [6] is used to maximize the compiler's scheduling ability. Treegion-scheduling for GCC is implemented by Rosier and Conte on a gcc-4.0.2 branch available at GNU website [21][1].

Register allocation is done on the RTL instructions using the constraints of the target architecture provided by the compiler-architect. The RTL is then mapped to appropriate instructions available in the target architecture. This mapping is done using the machine-description code provided by the compiler-architect. If a one-to-one mapping is not found, the architect must provide an appropriate combination of instructions to handle such an RTL.

If the architect cannot represent a certain instruction in the machine-description, the instruction will not be emitted. In addition, two applications that contain the same RTL will contain the same instruction(s) in the executable.

[1] For this work, we took a patch from Rosier and Conte and applied it onto the GCC port made available by OpenRISC

IV. Experimental Framework

A. Processor Architecture

In order to gain a fine-grain understanding of energy dissipation in processors, a hardware-level model of the embedded processor is necessary. For this work, OpenRISC 1000 core is used [12] [31]. This is a five stage processor, with basic DSP capabilities. The instruction-set is similar to several popular embedded architectures such as ARM [23], MIPS [16], Atmel [30] etc.

The processor is synthesized at 13ns for *all* runs without any slack-violation using Artisan Physical IP 1 Volt, SAGE-X 90 nm RVT standard-cell library by ARM [33] using Synopsys Design Analyzer. This standard-cell library is equivalent to low-operating power libraries described by ITRS [34]. Such libraries are most-often used for embedded processors today [34] [19].

The synthesized processor is then placed-and routed using Cadence Encounter to obtain the parasitic information. Then, appropriate benchmarks are run through the synthesized cores using the Verilog VERA simulator to capture the switching activity in VCD format.

The switching information, the synthesized Verilog core, the parasitic information and the timing information is analyzed using Synopsys Primetime (formerly Primepower) to gain static and dynamic power values. These values are converted to energy values using the cycle-time information obtained from VERA simulator. Energy values are used because energy is more analogous to battery-life than power [27]. As per EE Times, after SPICE, Primetime currently provides the most accurate power values [5].

Measuring power in this format is very time consuming. This work was made possible using 6 SPARC and 6 Linux multiprocessor systems that were solely dedicated for this work for 40 days. This extra expense allows for accurate and authoritative energy values and helps reveal the fine-grain effects that are unable to be captured in high-level simulators.

B. Issue-Width Modification

To study the scalability of the new opcode-configuration, we combined two-issue OR32 data-paths and created a cluster. These clusters were combined together to create a 4-issue (2-Cluster) and 8-issue (4-Cluster) statically scheduled processor. To do inter-cluster copy of register-values, a specialized "l.copy" instruction was added into the ISA.

To compile benchmarks for this clustered-architecture, we added a UAS scheduler [15] on top of the Treegion-scheduler. GCC provides several hooks that intercept the instruction scheduler to manipulate and rearrange the ready-list. We used the "TARGET_SCHED_FINISH_GLOBAL" hook. There are several priority-assignments available in UAS, but the cycle-weighted-placement (CWP) was used since it gives the most optimal results.

C. *Benchmark Selection and Execution Methodology*

To accurately represent embedded system workloads, 6 benchmarks from Embedded Microprocessor Benchmarking Consortium (EEMBC) were chosen [32]. Table 1 explains the benchmarks in detail.

Table 1: EEMBC Benchmarks

Benchmarks	Description
aifir01	FIR Filter
conven00	Convolutional encoder
ospf	Open-shortest path first/Dijkstra's Algorithm
puwmod	Pulse Width Modulation Algorithm
routelookup	IP Datagram forwarding Algorithm
viterb00	Viterbi Decoder

EEMBC benchmarks are created by an independent consortium whose primary objective is providing representative workloads for embedded systems. These benchmarks must be run as per the specifications provided by EEMBC. **Figure 2** shows the structure of an EEMBC benchmark. All performance evaluation metrics must be measured only for the code between "th_signal_start" and "th_signal_finished." Also, EEMBC provides a minimal number of iterations that will guarantee the correct execution of all the benchmarks.

```
Benchmark_function(VOID)
    /* Initialize all the variables necessary  for benchmark */
    /* Read FILES and store in Arrays */
    Call th_signal_start()

    For (loop_cnt = 0; loop_cnt < ITERATION; loop_cnt++)
    {
        /*   THE ACTUAL TASK OF BENCHMARK    */
    }

    Call th_signal_finished()
    /* Write information in appropriate files */
Return from Benchmark_function
```

Figure 2: Structure of EEMBC Benchmarks

To run EEMBC on synthesized Verilog model, register and memory values available after the "th_signal_start" were inserted into the processor during the enabling of the reset signal. The program counter is then pointed to the start of the for-loop and the execution is begun. To check the accuracy, several debug-runs were performed by manually adding "$display" statements into the Verilog code (after synthesis) and the test-bench to monitor the control flow. The memory values were checked by comparing values obtained by the C simulator. Since the execution environment does not provide a way to interface an Operating System, system calls could not be handled correctly in hardware, thus only the benchmarks that had no system-calls inside the for-loop were selected.

D. *RISC Opcode Configuration*

Several major RISC instruction-sets have telescoping encoding [25]. In telescoping encoding, two similar instructions of same type (e.g. Arithmetic instructions) have the same primary opcode, and different secondary opcodes. There are two ways to decode instructions following such encoding: using either a parallel or serial approach. Figure 3 shows examples of parallel and serial approaches for decoding an "OR" instruction in OpenRISC.

In the parallel approach, the opcode and sub-opcodes are compared in a single step. Thus, as per Figure 3, to decode an OR instruction, the parallel approach takes 12 comparisons, regardless of a match. In the serial approach, the primary opcode is compared, then if there is a match, then the secondary opcode is compared. Such cascaded comparisons can introduce additional latches into the system

Suppose that there are 1 million OR instructions in a benchmark. If the decoder is written using a parallel approach, then to decode this instruction, there must be 12x1 million comparisons. In the serial approach, in addition to 12x1 million comparisons, 6x1 million additional latches are charged and discharged, which can consume significant amount of energy.

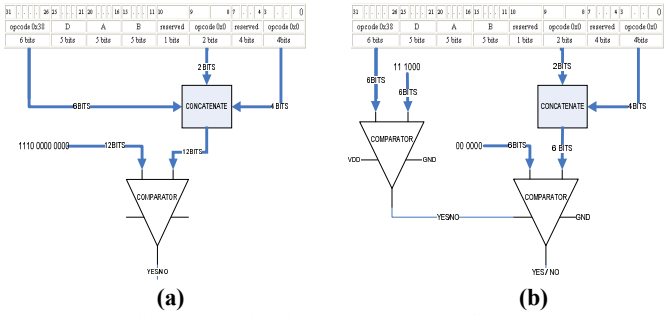

Figure 3: Decoding Approaches for Telescoping Encoding

Not all instructions have multiple fields. For example, the "return from exception (rfe)" instruction does not have a sub-opcode field. The unused bits in this instruction are left as "don't-cares." The alarming observation is that several instructions that occur commonly in several benchmarks have sub-opcode fields and instructions that rarely occur in a regular instruction execution have no sub-opcode field. This can have a significant impact on energy.

V. OPCODE-OPTIMIZATION METHODOLOGY

In this work one representative benchmark is sampled and the opcodes are re-designed such that commonly-occurring instructions in this benchmark are decoded easily. In addition, the opcodes are distributed such that the instructions that are adjacent to each-other have minimal switching. Solving this problem manually is exponentially difficult.

To extract the instruction trace of the training benchmarks, an OR1000 instruction-set simulator was written in C++. To find adjacent instructions, Markov chains were created from

the instruction trace. In the beginning, two, three and four-instruction chains were considered, but the three and four-instruction chains contributed minimally, thus we do not discuss them in detail in this paper.

To create the optimal opcode-distribution, the traces are analyzed using the algorithm described in **Figure 4**. The function accepts the instruction-trace of the training benchmark and a list of instructions the compiler is able to represent in its machine description.

This trace is then stepped through by another function that creates another list to hold all the instructions that the current trace is able to represent. This list holds all the instructions in descending value of the instruction-type's occurrence. This list is usually a subset of the GCC represented traces.

The instruction trace, along with the two lists, is fed into another function to prioritize the opcodes. For example, if "ADD" is the highest occurring instruction in the training list, then the priority encoder will try and make sure the ADD instruction gets a unique primary opcode and no sub-opcode.

When all the elements of the "Trace_Rep_Insns" list is visited, the optimizer visits all the instructions that GCC is able to represent, not found in "Trace_Rep_Insns." The rest of instructions in the ISA are given primary and secondary opcode fields. This function outputs the "Prio_Insn_Trace."

Next, the Prio_Insn_Trace is analyzed to make sure adjacent instructions have the lowest switching. This is dependent on the issue-width. For example, a processor with an issue-width 'n,' the adjacent instructions consists of instructions that are 'n' instructions apart.

```
Array Opcode_Optimization_Algorithm (Array Insn_Trace,
                                     INTEGER Trace_Size
                                     Array GCC_Represented_Insns)
{

  /* All the instruction types available in the trace */
  Trace_Rep_Insns = Find_Insn_Types (Insn_Trace);

  /* Give Priority in the following Order:
     1) Insns Represented In Trace
     2) Insns The compiler is able to represent that
     Omitted in 1
     3) Rest of the Instructions in ISA
  */
  Prio_Insn_Trace= Create_Priority_Encoding(Insn_Trace,
                                            Trace_Rep_Insns,
                                            GCC_Represented_Insns);

  /* Find adjecent Instructions in the Trace */
  Adj_Chains = Create_Adjency_Chains(Prio_Insn_Trace);

  /* Create a New Instruction Template with New Opcodes */
  Insn_Template = Minimize_Switching_Among_Adj_Chains(Prio_Insn_Trace,
                                                      Adj_Chains);
  /* Remap the opcodes to the new Configuration */
  New_Insn_Trace = Remap_Insn_Trace(Insn_Trace,Insn_Template);

  return New_Insn_Trace;
}
```

Figure 4: Opcode Optimization Algorithm

This adjacent-instruction chain along with the Prio_Insn_Trace is sent to a minimum-distance genetic algorithm [22] that minimizes switching among the adjacent instructions. This function gives an instruction template that contains information about the placement of various components of the instruction.

This template is used to remap the instructions from the original OpenRISC encoding to the new optimized encoding. Similarly, this template is used to remap all testing benchmark to the newly optimized encoding.

Figure 5 gives a flow-diagram for designing a new opcode and how a new benchmark is remapped using the template of the new opcode configuration.

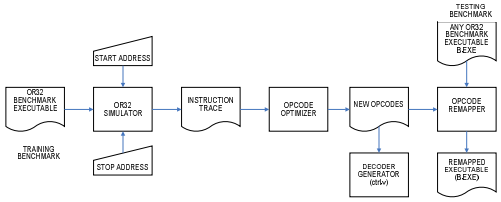

Figure 5: Flow-Diagram of our Methodology

VI. RESULTS

A. *Single Issue Results*

Each of the six benchmarks was used as a training-benchmark. Each of the benchmarks was then tested on all the "trained" processors. Figure 6 and Figure 7 displays the dynamic and static energy values for the base case. The results for static and dynamic energy savings are indicated in Table 2 and Table 3.

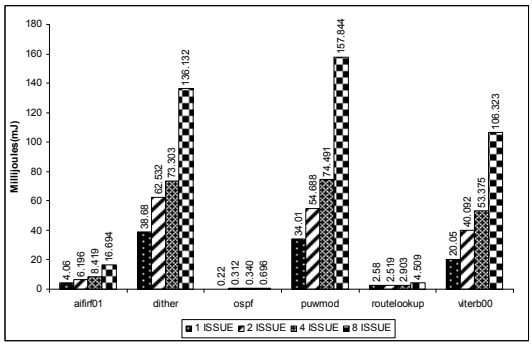

Figure 6: Dynamic Energy Distribution for Base Processor

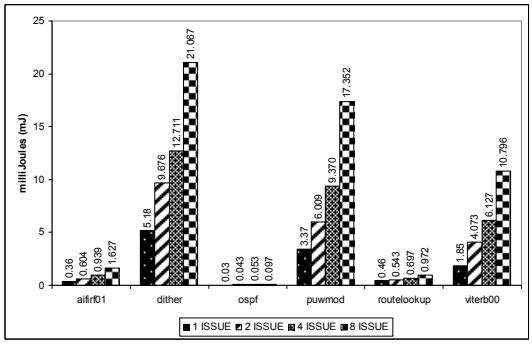

Figure 7: Static Energy Distribution for Base Processor

Table 2: Percentage Reduction in Dynamic Energy

TESTING BENCH	TRAINING BENCHMARK: aifir01	conven00	ospf	puwmod	routelookup	viterb00
aifirf01	**16%**	1%	14%	0%	1%	1%
conven00	2%	**16%**	3%	2%	2%	2%
ospf	16%	0%	**19%**	1%	0%	2%
puwmod	10%	-2%	0%	**17%**	9%	-6%
routelookup	2%	0%	3%	4%	**14%**	0%
viterb00	-2%	-1%	-4%	-2%	-4%	**16%**

Table 3: Percentage Reduction in Static Energy

TESTING BENCH	TRAINING BENCHMARK: aifir01	conven00	ospf	puwmod	routelookup	viterb00
aifirf01	**0%**	1%	-1%	-4%	0%	1%
conven00	7%	**8%**	8%	8%	8%	8%
ospf	-3%	-3%	**-3%**	-2%	-2%	-2%
puwmod	-1%	0%	-1%	**-1%**	-1%	-1%
routelookup	7%	9%	8%	8%	**8%**	8%
viterb00	1%	0%	2%	0%	0%	**-1%**

If the decoder contributes 15-20% of the total energy dissipated by the processor, then how can *Ospf* have an energy reduction as high as 19%? To answer this, the energy dissipation of all the processor components that were trained with *ospf* and tested on *ospf* was measured. Switching in all the major ports was captured using $display statements added into the processor. This was done on the non-synthesized processor since design analyzer cannot synthesize $display statements

The switching in the instruction port was reduced by 23%. The number of latches inside the decoder was reduced by 65%. Overall, a 60% reduction in the decoder energy (dynamic) dissipation was noticed. This translated to an approximate 11% reduction in energy in the processor. The freeze-unit and the exception unit each obtained approximately 5% reduction in energy, which contributed to 1% (of the 19% seen) of the total energy reduction. The fetch-unit had approximately 20% reduction, amounting to ~3% total energy reduction. The reduction in switching among the connecting busses contributed another 2% overall energy reduction.

As a result, even though the only component that was modified in the processor was the decoder, the reconfiguration of the opcodes created a domino effect for energy reduction in the processor. Previous high-level simulators have not been able to measure or achieve such fine-grained reductions.

To understand further about the characteristics of the benchmarks, we studied the high-level C code. Figure 8 show the number of function-calls done by each benchmark during their execution.

During each function-call all the registers used by the callee are saved on the stack and restored from the stack in the beginning and the termination of each call. In OpenRISC assembly, the only method to implement push and pop information into the stack is by using a load-word or store-word to access the stack. After all the loads or stores, the stack pointer (register r1) is incremented or decremented accordingly using an add-immediate.

When the dynamic trace of aifir01 was observed, there were a significant amount of memory operations. Similarly, the benchmark calls a function called "GetInputValues" which loads a significant amount of data from a global variable (inpVariableROM). To load from a global variable in OpenRISC, the 16 least significant bits of the variable is "ORed" with register zero that holds a constant zero into a target register. Then the 16 most significant bits are loaded into the register using a "movhi" instruction that loads its immediate value into the top 16 bits of a register. "Movhi" and "ori" instructions are analogous to the movw and movt instruction in the ARM architecture [23]. Pulse-width modulate also has a large number of function-calls, but that instruction-contribution is overshadowed by the excess number of computation done between the function-calls.

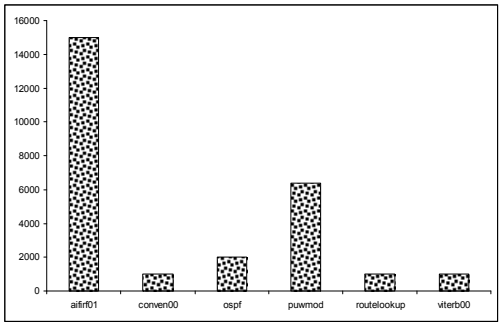

Figure 8: Function-calls in each Benchmark

It is popularly known that viterbi decoding is used to decode convolutional codes in communication systems. Both convolutional encoders (conven00) and viterbi decoders (viterb00) have significant amount of shifts, adds and memory operations. However, as per Table 2, an instruction-set that is optimized for one fails to give a significant energy reduction on the other.

After examining the instruction trace and the high-level code, it was found that conven00 did all computations on the 4-byte and 1 byte data widths, that is, they had variables and data that were either "int" or "char." Viterb00 did all computations on 2-byte data-widths (short). OpenRISC has different instructions for each data-size. Thus, most of the half-word instructions were given smaller opcodes and the word-level and byte-level ones were given longer opcodes. The opposite was done for conven00.

Next, the number of instruction-chains required to obtain 50% coverage was captured. Figure 9 shows this result. This number illustrates the diversity in the benchmark. For example, ospf requires only 4 chains to get 50% coverage; this implies that if another benchmark contains some of these chains, then a good energy reduction can be seen. *Aifir01* and *ospf* have 2 chains in common that falls in this range. Thus, a high dynamic energy reduction can be seen in *ospf* when the opcodes are optimized for *aifir01*.

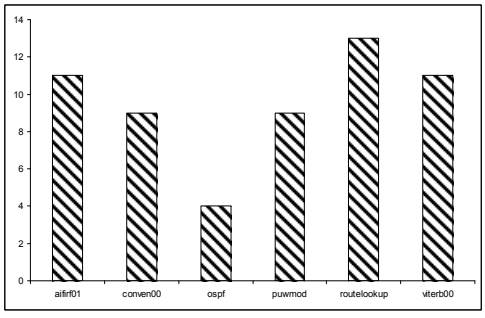

Figure 9: Number of Chains to gain 50% Coverage

Many of the explanations thus far in this section apply primarily to dynamic energy. Sub-threshold leakage is a dominating issue only in the memory hierarchy [8]. For this work, memory was not modeled since we were unable to find a synthesizable memory that can be interfaced with the Verilog core. **Figure 7** show that leakage energy, on average, contributes only 9% of the total energy consumption. As per

Table 3, leakage energy tends to increase slightly when there is some decrease in dynamic energy. This is because the primary way to reduce dynamic energy is to reduce switching in the processor interconnects. After further analysis of individual components of the processor, the increase in leakage energy was in components where switching is greatly reduced: the decoder and the instruction ports.

B. Energy-Saving Patterns with Issue-Width Scaling

Wider issue-widths are typically used to increase performance in processors. In statically scheduled machines, when the issue-width is changed, the whole benchmark, in general, must be re-compiled to gain the full-effectiveness of the compiler. In the previous section, the opcode-configuration (trained using a single-issue benchmark), was tested using single-issue benchmarks. In this section, the effect of scaling issue-widths on the trained processor is explored. The processor was trained with single-issue benchmarks and tested using 2-issue, 4-issue and 8-issue executables. The static and dynamic energy reduction is reported in Table 4-Table 9

Table 4: Percentage Reduction in Dynamic Energy (2 Issue)

TESTING BENCH	TRAINING BENCHMARK (Trained Using Single Cluster Trace)					
	aifirf01	conven00	ospf	puwmod	routelookup	viterb00
aifirf01	16%	1%	14%	0%	1%	1%
conven00	2%	16%	3%	2%	2%	2%
ospf	16%	0%	19%	1%	0%	2%
puwmod	10%	-2%	0%	17%	9%	-6%
routelookup	2%	0%	3%	4%	14%	0%
viterb00	-2%	-1%	-4%	-2%	-4%	16%

Table 5: Percentage Reduction in Dynamic Energy (4 Issue)

TESTING BENCH	TRAINING BENCHMARK (Trained Using Single Cluster Trace)					
	aifirf01	conven00	ospf	puwmod	routelookup	viterb00
aifirf01	14%	1%	14%	0%	14%	2%
conven00	2%	14%	3%	2%	3%	16%
ospf	15%	0%	19%	1%	15%	1%
puwmod	9%	-2%	0%	16%	16%	-10%
routelookup	2%	0%	3%	4%	11%	0%
viterb00	-2%	-1%	-4%	-2%	-6%	13%

Table 6: Percentage Reduction in Dynamic Energy (8 Issue)

TESTING BENCH	TRAINING BENCHMARK (Trained Using Single Cluster Trace)					
	aifirf01	conven00	ospf	puwmod	routelookup	viterb00
aifirf01	14%	1%	14%	0%	11%	2%
conven00	2%	14%	3%	5%	2%	16%
ospf	15%	0%	19%	4%	11%	1%
puwmod	10%	-2%	0%	16%	13%	-10%
routelookup	0%	0%	3%	5%	10%	0%
viterb00	-2%	-1%	-4%	3%	-9%	13%

When the processor is scaled, the major components that handle instructions (and thus opcodes) are scaled as well. For example, an 8-issue processor had 8 decoders, 8 instruction ports output from the fetch unit, and the freeze-unit and the except-unit took 8 instructions as inputs. Thus, the energy distribution also scaled accordingly in Figure 6 and Figure 7.

In all the cases, the dominating instructions in the trace remained the same (since the algorithm or the operations in the C-code is unchanged). The copy instruction, in all cases accounted for << 0.1% of the dynamic total instruction. With the same dominating instructions, the reduction in the number of latches charged and discharged remained the same. This contributed to the major energy reduction in all cases. All benchmarks except *routelookup* had very low IPC. The IPCs barely changed between 1-issue and 2-issue configurations. IPC distribution for all issue-widths is given in Figure 10.

Table 7: Percentage Reduction in Static Energy (2 Issue)

	TRAINING BENCHMARK (Trained Using Single Cluster Trace)					
	aifirf01	conven00	ospf	puwmod	routelookup	viterb00
aifirf01	1%	1%	-1%	-4%	0%	1%
conven00	4%	4%	6%	5%	4%	6%
ospf	-4%	-4%	-3%	-2%	-2%	-2%
puwmod	-2%	-2%	-1%	-1%	-1%	-1%
routelookup	5%	5%	4%	4%	5%	5%
viterb00	1%	1%	2%	0%	0%	-2%

Table 8: Percentage Reduction in Static Energy (4 Issue)

TESTING BENCH	TRAINING BENCHMARK (Trained Using Single Cluster Trace)					
	aifirf01	conven00	ospf	puwmod	routelookup	viterb00
aifirf01	1%	1%	-1%	-4%	0%	1%
conven00	4%	4%	6%	5%	4%	6%
ospf	-4%	-4%	-3%	-2%	-2%	-2%
puwmod	-2%	-2%	-1%	-1%	-1%	-1%
routelookup	5%	5%	4%	4%	5%	5%
viterb00	1%	1%	2%	0%	0%	-2%

Table 9: Percentage Reduction in Static Energy (8 Issue)

TESTING BENCH	TRAINING BENCHMARK (Trained Using Single Cluster Trace)					
	aifirf01	conven00	ospf	puwmod	routelookup	viterb00
aifirf01	1%	1%	1%	-4%	0%	1%
conven00	4%	4%	-6%	5%	3%	6%
ospf	-3%	-3%	3%	-2%	-1%	-2%
puwmod	-2%	-2%	1%	-1%	-1%	-1%
routelookup	3%	3%	-2%	2%	2%	3%
viterb00	1%	1%	-2%	0%	0%	-2%

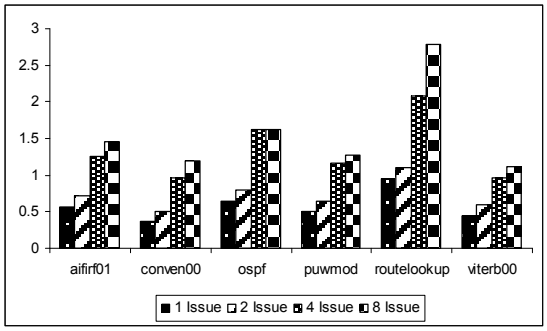

Figure 10: IPC for EEMBC Benchmarks

Several instructions that were adjacent in single-issue were adjacent in multi-issue executable, thus they had similar energy-reduction values. A more powerful compiler that has a larger view of the instructions can show a change in this result. *Routelookup* is the most parallel benchmark. When the issue-width changed, the adjacent-instruction chains changed that causes some minor difference in energy.

VII. CONCLUSION

Energy reduction is an important issue in an embedded system. Low-energy systems help increase the battery life of the system. There are several components in a processor that can be optimized to improve energy consumption. It is necessary to save energy in all ways possible while not sacrificing the performance.

In this work, a technique to optimize the instruction-set based on a sample of the workload for which the processor is designed is presented. The presented technique neither causes any additional cycle-count nor increase the clock period of the base processor. The only hardware modification that is necessary is the instruction decoder. The newly generated instruction-decoder can be swapped with the original without any further modifications to the processor. Finally, it was shown that when the issue-width was scaled in this VLIW processor, the energy-savings scale accordingly.

This paper shows that if the sample set is selected correctly, a 15-20% energy reduction is achieved by intelligently assigning opcodes and no performance is lost. In addition, it also provides a loose-rubric to design software for the particular processor. If the new software that is to be added to the system is designed in a similar structure and contains similar characteristics (e.g. memory intensive vs. computationally intensive or word-length vs. byte-length), there can be an energy reduction with no performance loss.

REFERENCES

[1] A. Bechini, T. M. Conte, C. A. Prete, "Opportunities and Challenges in Embedded Systems," IEEE Micro, 2004

[2] L. Benini, et al., "Reducing Power Consumption of Dedicated Processors Through Instruction-set Encoding," Proceedings of the Great Lakes Symposium on VLSI Design, pp. 8-12, February 1998

[3] A. C. Cheng, G. S. Tyson, "An Energy Efficient Instruction Set Synthesis framework for Low Power Embedded System Designs," IEEE Transactions on Computers, Vol. 54, No. 6, pp. 698-712, June 2005

[4] S. Haga, et al., "Dynamic Functional Unit Assignment for Low-Power," Proceedings of the Design, Automation and Test in Europe Conference, 2003

[5] R. Goering, "Synopsys Launches Power-Tool", EE-Times, 2000

[6] W. A. Havanki, S. Banerjia, T. M. Conte, "Treegion scheduling for wide-issue processors," HPCA, 1998

[7] A. Kalambur and M. J. Irwin, "An Extended Addressing Mode for Low-Power," Proceedings of the IEEE Symposium on Low Power Electronics, 1997

[8] N. S. Kim et al., "Leakage current: Moore's law meets static power," IEEE Computer, Vol. 26, Issue 12, 2003

[9] S. Kim, J. Kim, "Low-power data representation," Electronic Letters, Vol. 36, No. 11, 25th May 2000

[10] S. Kim, J. Kim, "Opcode encoding for low-power instruction fetch," Electronic Letters, Vol. 35, No. 13, 24th June 1999

[11] U. Ko, P. T. Balsara, W. Lee, "Low-Power Design techniques for High-Performance CMOS adders," IEEE Transactions on VLSI Systems, Vol. 3, No. 2, June 1995

[12] D. Lampret, "OpenRISC 1200 IP Core Specification," www.opencores.org, 2001

[13] M. Lorenz, R. Leupers, P. Marwedel, "Low-Energy DSP Code Generation Using a Genetic Algorithm," ICCD, 2001

[14] D. Novillo, "GCC- An Architectural Overview, Current Status and Future Directions," Proceedings of the Linux Symposium, Vol. 2, July 19-22nd, 2006

[15] E. Ozer, S. Banerjia, T. Conte, "Unified-Assign and Schedule: a new approach to scheduling for clustered register-files," Proceedings of International Symposium on Microarchitecture, 1998

[16] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design," Morgan Kauffman Publishers, 1998

[17] G .G. Pechanek, S. Larin, T. Conte, "Any-size instruction abbreviation technique for embedded DSPs," 15th Annual IEEE international ASIC/SOC conference, 2002

[18] M. Pedram and A. Abdollahi, "Low-Power RT-level synthesis techniques: a tutorial," IEE Proceedings-Computer and Digital Techniques, Vol. 152, No. 3, May 2005

[19] L. Pickup, S. Tyson, "Hot Chips...Not!" Chip Design Magazine, August/September, 2004

[20] N. Ramsey, J. W. Davidson, "Machine Description to build tools for embedded systems," Lecture Notes in Computer Science, Vol. 1474, 1998

[21] M. C. Rosier, T. M. Conte, "Treegion Instruction Scheduling in GCC," GCC Developers Summit, 2006

[22] Y. G. Saab, V. B. Rao, "Stochastic Evolution: A fast effective heuristic for some generic layout problems," Proceedings of 27th IEEE/ACM Design and Automation Conference 1990

[23] A. N. Sloss, D. Symes, C. Wright, "ARM Systems Developer's Guide," Elsevier Inc., 2004

[24] R. Stallman, "GCC Internals Manual for GCC 4.0.2," FSF Press, 2006 (available with the compiler source)

[25] A. Tannenbaum, "Structured Computer Organization," Pearson Education, 4th Edition, 1998

[26] V. Tiwari, S. Malik, A. Wolfe, "Compilation Techniques for Low Energy: An Overview." ISLPED, 1994

[27] V. Tiwari, et al., "Instruction Level Power Analysis and Optimization of Software," Journal of VLSI Signal processing, pp. 1-18, 1996

[28] A. Varma, E. Debes, I. Kozintsev, B. Jacob, "Instruction-level power dissipation in the Intel XScale Embedded Processor," Proceedings of SPIE's 17th Annual Symposium on Electronic Imaging Science & Technology, San Jose CA, January 2005

[29] S. Woo, J. Yoon, J. Kim, "Low-Power Instruction Encoding Techniques," Proceedings of SOC Design Conference, 2001

[30] "AVR32 Architecture Manual," http://www.atmel.com, 321 Pages, Revision A, Updated: 02/06

[31] "OpenRISC Architecture Manual," http://www.opencores.org, 2003

[32] "The Embedded Microprocessor Benchmark Consortium," http://www.eembc.org

[33] "CMOS 9SF (90 nm) RVT Process 1.0 Volt SAGE-X v3.0 Standard Cell library Databook," Revision 1.2, ARM Ltd, 2006.

[34] International Technology Roadmap for Semiconductors, http://www.itrs.net