Length Adaptive Processors: A Solution for the Energy/Performance Dilemma in Embedded Systems

Balaji V. Iyer Jesse G. Beu Thomas M. Conte
{biyer3, jbeu3, conte}@cc.gatech.edu
School of Computer Science,
College of Computing,
Georgia Institute of Technology, Atlanta, GA

Abstract

Embedded-handheld devices are the predominant computing platform today. These devices are required to perform complex tasks yet run on batteries. Some architects use ASICs to combat this energy-performance dilemma. Even though they are efficient in solving this problem, ASICs are very inflexible. Thus, it is necessary for a general purpose solution. In addition, no single processor configuration provides the best energy-performance solution over a diverse set of applications or even throughout the life of a single application. Thus the processor needs to be adaptable to the specific workload behavior. Code-generation and code-compatibility are the biggest challenges in such adaptable processors.

In this work, we provide an embedded processor that has the flexibility of a general purpose processor with the specialization of an ASIC. It is able to dynamically modify its issue width with one VLIW instruction overhead. This processor is designed in Verilog, synthesized, DRC-checked, and placed and routed. Its energy and performance values are reported using industrial-strength transistor-level analysis tools to dispel several myths that were thought to be dominating factors in embedded systems. In addition, we provide the software tools that help achieve optimized code for such dynamic architectures and discuss some of the code-generation procedures and challenges.

1. Motivation

Embedded-handheld devices are the predominant computing platform today [3]. These devices are required to perform tasks that were once only attempted by high-performance systems [38]. For example, a modern mobile phone, in addition to sending and receiving audio-signals, can capture images and video, maintains a daily planner, enables web browsing and sends and receives digital information. Another constraint imposed upon these systems is that they must still use batteries as the primary power source [3]. Thus, it is important for embedded-handheld devices to give comparable performance with a high-performance system while consuming significantly less energy.

Some designers have tried to combat this problem by using application-specific integrated circuits (ASIC) as the primary embedded processor. An ASIC processor, however, is inflexible and has to be re-designed whenever a new application is introduced into the system. To gain a flexible solution, architects have used a tailored general purpose processor (GPP) for embedded systems. These processors are simple and require significant help from the compiler or operating system for scheduling, branch-prediction, etc. [3]. The advantage that these tailored GPP have over ASIC is that when new applications are introduced, the processor can execute the application without having to redesign the system.

Even though these tailored processors provide a flexible solution, diverse characteristics among embedded applications and diversity within an application make it impossible to select one processor-configuration that provides the optimal energy-performance balance. In the past, there have been three major works that tried to study and solve this problem: the Lx [15], Tensilica Xtensa 7[46] and the OptimoDE [10] processors. These processors provide a static-scalable solution that allows customization of the processor for target application(s).

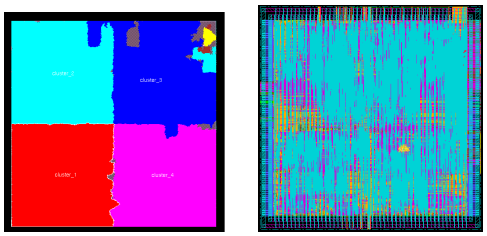

Figure 1: 8-Issue CLAW Architecture (After Placement and After Routing)

In this work, we extend the ideas of Lx and OptimoDE to provide dynamic-scalability. We propose a Clustered Length-Adaptive Word Processor (CLAW) that is able to provide the flexibility of a general-purpose processor while having an adaptable nature. CLAW allows dynamic modification of the issue width with one-instruction's worth of overhead. In such architectures, generating code to make them adaptable for several configurations becomes the biggest challenge [15][10]. We provide a toolchain that uses aggressive scheduling techniques to find the optimal processor configuration and communicates its findings to the processor. A prototype for the CLAW processor is designed in Verilog, synthesized, DRC checked and placed and routed. An eight-issue (4-Cluster) implementation of CLAW is shown in Figure 1. The processor also has the ability to execute up to eight threads in hardware where the number of hardware threads is controlled by either the user or the operating system. For brevity, in this study we only

discuss the dynamic scalability of CLAW for single-threaded applications.

The main focus of this work is to generate efficient code for flexible issue-width architectures through interaction between the hardware and the compiler. The paper is organized as follows. Section 2 describes related-work in this area. In this section we also describe how CLAW differs from previous work. Section 3 provides an architecture level description for CLAW. In section 4, we describe the challenges of compiling for CLAW. Section 5 describes the experimental-framework, and the benchmarks. We present our results in section 6 and we provide a conclusion in section 7.

2 Related Work

The idea of customizing a general-purpose processor for an application was first proposed by [16]. To our best knowledge, the only processors that provide flexibility and adaptability like CLAW are the Lx [15], Tensilica Xtensa LX2 [46] and the OptimoDE processors [10]. Figure 2 shows the design process of Lx, OptimoDE, Tensilica and CLAW (assuming we are designing the processor to target programs A and B). The only major difference between Optimode and Tensilica is that Optimode allows the user to fully customize the instructions, while Tensilica uses a standardized ISA. Lx architects provide a framework that analyzes a benchmark (or a set of benchmarks) and design a processor with appropriate issue-width, function-units, etc. to maximize the processor performance using the appropriate energy budget. OptimoDE framework tries to analyze the source-code and provide hints to the user regarding the optimal issue-width, function-units, data-path sizes, etc. Standard function units are inserted by the tools, but custom-units must be hand-generated.

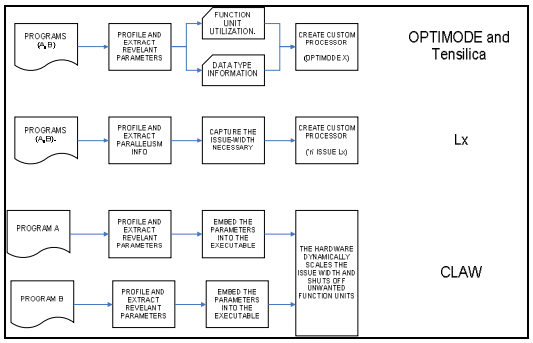

Figure 2: OptimoDE, Tensilica, Lx and CLAW Design-flow

The biggest drawback for Lx and OptimoDE is they are static approaches. Let's assume we are trying to add a new application ('C') into the processors designed in Figure 2 As shown in Figure 3, if a new application is introduced into these systems, the processors must be redesigned for optimal functioning, which can be expensive and time-consuming. This problem is overcome in CLAW by providing mechanisms to dynamically adapt issue-widths and function-unit sizes during compile-time.

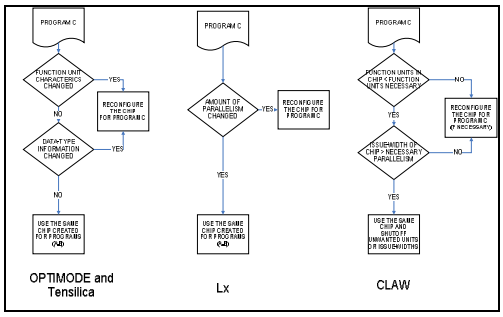

Figure 3: Steps for adding new Application into OptimoDE, Tensilica, Lx and CLAW

To do the dynamic modification of issue width, the most successful method employed by several high-performance processors is gating the clock for the unused units [29][32][21][30][36]. The granularity of the unit can be a specific gate [4], a function-unit [21][1][36], processor-stage [22][30][27][29], or an entire cluster [32][5]. Each of the methodologies described can be beneficial, depending on the application. The key question is at what part of the program must the gating occur so that optimal energy is consumed with virtually no performance degradation? We provide the answer to this using our CLAW software-framework.

Several super-scalar designers have studied this problem. In out-of-order dynamic-scheduling processors, however, this problem is trivial because the processor has direct control of the scheduling. Buyuktosunoglu et al. [6] provides an adaptive issue queue for reducing processor power. Albonesi [2] provides a methodology to dynamically shut off units and processor issue-widths in super-scalar processors to save power. Unfortunately, dynamic-scheduling processors are not power-efficient for embedded systems. As per our calculations and comparisons with [41], for the same transistor technology, the scheduling logic of a superscalar alone took more power than an entire VLIW processor of the same issue-width.

Tai and John [28] proposed a method to dynamically scale processor resources such as the reorder-buffer, load-store queue and the instruction-window on a super-scalar processor. They propose using specialized instructions inserted by the operating system. We incorporate this idea into our design, however, we insert specialized instructions using a profiling compiler because many embedded systems do not have complex OS support, but a compiler is almost always available.

3 The CLAW Microarchitecture

The microarchitecture used in this processor is CLAW. CLAW is a variable-width processor whose width can be modified as necessary during design-time. Additionally, the processor's width can be reduced dynamically during execution without significant overhead. The prohibitive factor in wide-issue processors is the wire-length delays [20], however, clustered

architectures circumvent this problem by "forcing data locality into the processor [40]." Another advantage with the clustered architecture is that we have fine-grain control over the processor control path.

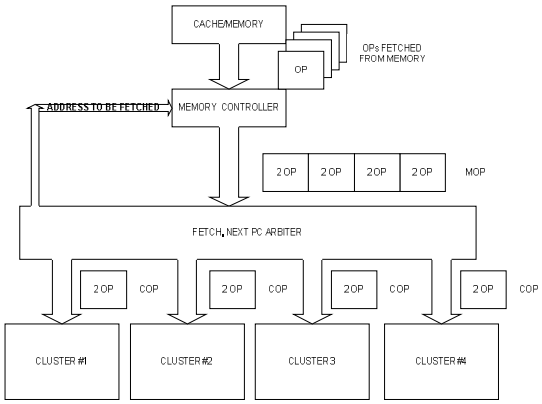

Figure 4: Top Level CLAW Block Diagram

Figure 4 shows the top-level diagram of CLAW. An entire Multi-Op (MOP) is fetched from the cache or memory using the memory controller. A MOP is synonymous to a VLIW instruction or a group in IA-64. This MOP is then sent to the Fetch unit, which divides each MOP into cluster-Ops (COP). Each COP contains two operations (OP). Each Op is synonymous to an individual instruction such as ADD or LOAD. The relationship between MOP, COP and OP is given in Figure 5. The last OP of each MOP is indicated by setting the "T" bit, also shown in Figure 5. The "X" bit is reserved for future-use. In CLAW, each cluster is able to execute two OPs. This number was chosen because our initial study of the benchmarks via simulation revealed IPC potential greater than one.

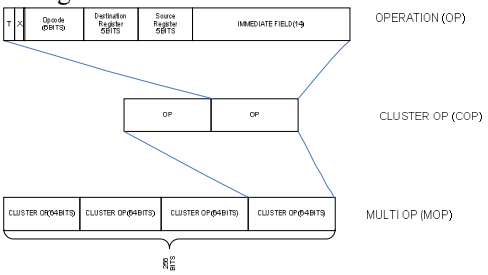

Figure 5: CLAW Instruction Granularities

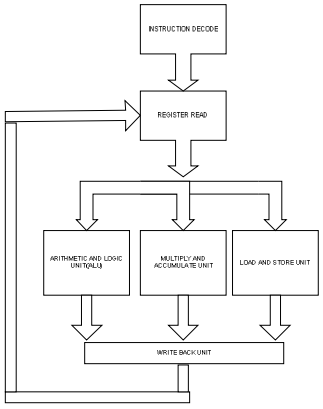

Figure 6: Components of a single Cluster

Each cluster is able to accept two OPs, simultaneously decode them, read the appropriate values from the local register-file, then execute them and write the results back to the register file or memory. Figure 6 shows the components inside a cluster. The instructions are able to see only its local register-file. Values from other clusters must be explicitly copied to the local register file using appropriate copy instructions. More information about inter-cluster copies is given in section 4.

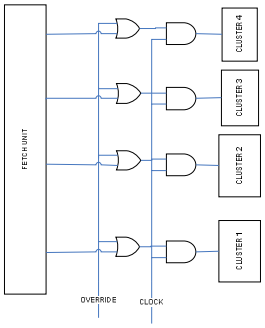

Figure 7: Cluster shutoff Mechanism in CLAW

The idea of dynamically modifying clusters came from the realization that ILP in a program's lifetime is bursty. The compiler can be used to capture such information during compile-time and provide hints for the processor to dynamically shutoff certain cores to reduce unwanted energy consumption. The shutoff instruction is required to be the first instruction inside the MOP. The fetch unit will poll the first instruction of the MOP to find a shutoff instruction. When it encounters such an instruction, the appropriate cluster indicated by the immediate field is shutoff. The shutoff mechanism in CLAW is shown in Figure 7; if necessary shutoff instructions can be overridden by the user via the appropriate input to the chip (override pin).

4 Compiling for CLAW Architecture

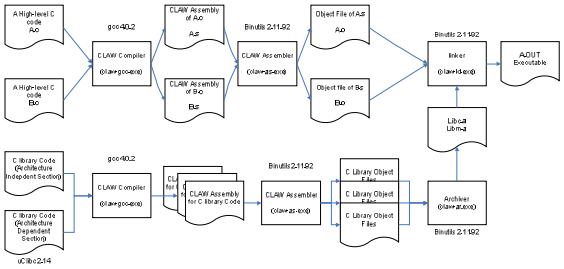

Figure 8: Steps for Translating High-level Source code to executable

Optimizing an application for different issue-widths to take advantage of ILP phase changes, while efficiently running them on the same VLIW processor is a non-trivial problem. Figure 8 shows the translation of a high-level program into an executable as well as the tool chain used toward this goal. To successfully compile and execute programs on the CLAW processor, we created a CLAW backend for binutils, gcc and uClibc. GCC is used to schedule instructions and the appropriate NOPs to

remove dependencies are inserted by the GNU assembler (part of binutils). The register-file of each cluster is represented as a register-class in GCC. For homogeneity and scalability, all instructions can be handled by all clusters. Instructions using registers 0-31 are assigned to cluster 1, and 32-63 to clusters 2 and so forth. If an RTL (after register allocation) has source and destination registers between 0 and 31, then the instruction is assigned to cluster 1. Registers in cluster 1 hold the state-information. Register 'r1,' 'r2' and 'r9' is designated as the stack-pointer, frame-pointer and return-address registers respectively. Registers r3-r8 are used for passing parameters between functions.

Since CLAW is a variable-width clustered architecture, a cluster-scheduling algorithm is necessary. Since GCC does not provide such a feature, we had to implement one over the existing scheduler. For this work, four major published cluster-scheduling algorithms were considered: Bottom-up-Greedy (BUG)[14] , Limited-Connected VLIW scheduling (LC-VLIW) [7], Unified Assign and Schedule (UAS) [34]and Combined cluster Assignment, Register allocation and instruction Scheduling (CARS) [24].

BUG takes a data-precedence graph (DPG) of a trace and traverses it from the bottom up. It recursively traverses the DPG and computes the function unit and operand availability of each instruction. Using this information, BUG assigns the operations in a trace. After this, the list scheduler inserts communication operations into the schedule as necessary. LC-VLIW focuses on partitioning code for a clustered machine that does not have full-connectivity between all clusters. This uses a multi-phase approach similar to BUG. The code is initially scheduled assuming the machine is a fully connected clustered VLIW machine. The code is then compacted locally to minimize the effect of inserted copy operations to the schedule.

UAS, unlike LC-VLIW or BUG, integrates the cluster-assignment in the instruction-scheduling phase. The schedule of operations and the DPG of the list are passed into the scheduler. Typically a list based scheduler is used with the list of operations being ordered based on a priority function. The inter-cluster buses are considered to be machine resources and are used within the scheduler when necessary. UAS claims to create a compact, efficient and nearly optimal schedule.

CARS tries to perform cluster-assignment, instruction scheduling and register allocation in a single step. CARS algorithm takes a dependence flow graph (DFG) with nodes representing operations and directed edges representing data and control flow. The CARS algorithm, unlike UAS, considers registers as a resource during cluster scheduling.

The single-phase algorithms (UAS and CARS) avoid several scheduling constraints that hinder optimal cluster scheduling. Important information such as instruction dependencies is lost between phases, which can result in a significant amount of inter-cluster copies. This in turn incurs significant performance and energy expense due to charging and discharging of long wires. Of the four algorithms, CARS seems to be the best solution since it considers scheduling, assigning and register-allocation concurrently. Unfortunately, our framework (GCC) does not allow register-allocation to be done together with scheduling, thus UAS was chosen.

To gain high-performance from UAS, an aggressive list-scheduler is necessary. Treegions [19] can provide large instruction-windows beyond basic blocks so that the list-scheduler can perform a tighter schedule. Treegion scheduling is implemented on a GCC-4.0.2 branch by Rosier and Conte [37] [1]. As a result, we decided to implement UAS algorithm on top of their Treegion-scheduler.

4.1 UAS on GCC

GCC provides several hooks that allow architects to manipulate and intercept the ready-list at different stages of scheduling [39]. The UAS was attached to the "TARGET_SCHED_FINISH_GLOBAL" hook. This hook is called immediately after the treegions are created. Figure 9 shows the flow-diagram of the major steps involved in the UAS implementation. A list of unscheduled Ops (as RTL) is taken from the Treegion scheduler and a list of instructions that are ready in the current cycle is assembled. For each instruction in this ready list, a new cluster is picked as per a priority function.

There are four different priority functions available in UAS, they are: sequential placement, random placement, magnitude-weighted placement (MWP) and completion-weighted placement (CWP). In sequential placement, the Ops are assigned in a round-robin fashion to each cluster. In Random placement, the Ops are placed to a random cluster chosen using a pseudo-random number generator (lrand48()). MWP schedules an Op to the same cluster as its predecessors. If a cluster's predecessors are assigned to two different clusters, either one can be a target for the current Op. In CWP, the Op is assigned to the same cluster as the predecessor that takes the longest to complete. The advantage CWP has over MWP is that since the current Op has to wait till the latest of its predecessor to complete, the holes in between can be used to schedule a copy instruction. For more detailed explanation, the reader is referred to [34].

The additional challenge encountered is register allocation. The register allocator tries to minimize assigning instructions to different register classes by mapping dependent-instructions into the same register class. Even though this can reduce additional cluster-usage, the register-allocator does not take cycle-time into account. To overcome this problem, the register allocator's mapping function (reg_class(..) function in passes.c [39]) was replaced with a specialized function (added using a new hook called "TARGET_MACHINE_DEPENDENT_REG_CLASS")

[1] We obtained a patch from M. C. Rosier for gcc-4.0.2 and applied the patch onto our compiler

that will assign instructions as per the UAS scheduler. The new hook implementation has been submitted to the gcc-patches mailing-list as a patch for acceptance. All the modifications described will not affect any other gcc port, and our gcc source-code can be configured for any other gcc-backend (e.g. x86) and function without any difficulty.

In CLAW it is possible to dynamically or statically shutoff (through clock-gating) certain clusters. The best way to accomplish this with minimal overhead is through a specialized instruction. In CLAW, we created such an instruction called "shutoff." The Immediate field for this instruction is a bit vector which indicates the appropriate cluster that needs to be shutoff. For example, "shutoff 0110b" implies that cluster 2 and cluster 3 should be shutoff. While the shutoff instruction can be inserted at any granularity, for this work we have studied using the shutoff instruction at the basic-block level and the function level.

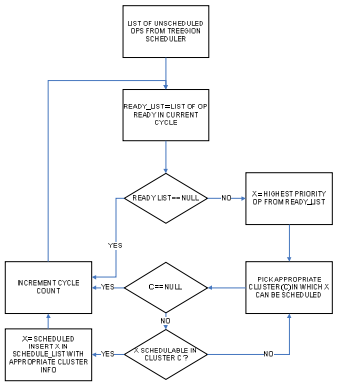

Figure 9: The Design-flow of the UAS Algorithm on GCC

```
01000084 <_t_run_test>:
 1000084:       27 08 7f cc     1.addi r1,r1,0x3fcc
 1000088:       05 40 00 ff     1.nop 0xff
 100008c:       05 40 00 ff     1.nop 0xff
 1000090:       c5 40 00 ff     1.nop.t.nv 0xff
 1000094:       35 00 44 14     1.sw 0x14(r1),r2
 1000098:       27 10 40 34     1.addi r2,r1,0x34
 100009c:       05 40 00 00     1.nop 0x0
 10000a0:       c5 40 00 00     1.nop.t.nv 0x0
```

Figure 10: Example of Unused Cluster (indicated in Blue boxes) in Viterbi benchmark[2]

Initially, the compiler inserts a shutoff instruction with '0' as its immediate field (indicating all clusters must be on). The CLAW profiler then scans the static code provided by the compiler to see if there are any empty clusters. Figure 10 shows an example of an empty cluster: if for the duration of an entire basic-block (or function depending on the granularity) there exists a common empty cluster, then the appropriate cluster is shutoff.

Figure 11 shows the algorithm of our profiler to detect idle clusters to power-down using the shutoff instruction. The algorithm accepts a block of instructions (BLK) as input. The profiler goes through every MOP to see if it can find Cluster-Ops (COP) with only NOP, that is, an unused-cluster. If such a scenario is noticed, the appropriate bit is set to '1' in the 'Unused' array. M.count indicates the MOP position in the BLK and C.count indicates COP position in the MOP M. This array is a two-dimensional array with rows indicating the number of MOPs in the block (indicated by BLK.MOP_Count) and the columns showing each cluster. This array must be dynamically allocated, but it is not shown in the figure for simplicity.

After stepping through all the MOPs in BLK, the profiler goes through the 'Unused' array to find if all the MOPs have common clusters that can be shutoff. This is done by checking if the summation of all the 1's in a column is equal to the number of MOPs in BLK. The list of empty clusters is returned back to the profiler from this function using the "Shutoff_Cluster_List" variable. The profile examines this to set the appropriate bit in the shutoff instruction. The profiler also displays the number of cases where all the clusters are turned-on for analysis.

```
Array Find_Shutoff_Clusters (BLOCK BLK)
{
  int Total_MOPS_Not_Using_A_Cluster=0;
  Array Shutoff_Cluster_List[NUMBER_OF_CLUSTERS];

  /* BLK.number changes for each block thus the arra must be but not shown for ease */
  /* BLK.number signifies the number of MOPS inside a BLOCK */
  Array Unused[BLK.MOP_Count][NUMBER_OF_CLUSTERS] = USED;

  /* Initialize all the values in Unused to '0' meaning they are used */
  for (II = 0; II < BLK.MOP_Count; II++)
    for (JJ = 0; JJ < NUMBER_OF_CLUSTERS; JJ++)
      Unused[II][JJ] = 0;

  /* Initialize shutoff_cluster_list saying we keep them all on */
  for (JJ = 0; JJ < NUMBER_OF_CLUSTERS; JJ++)
    Shutoff_Cluster_List[JJ] = 0;

  /* for each MOP in the BLK (whatever granularity we choose) */
  for (each MOP(M) in  BLK) {
    for (each COP(C) in M) {

      /* IF all the Ops inside a COP are NOPs, then we set unused to '1' */
      if (All_Ops_Are_Nops(C)) {
     Unused[M.number][C.number] = 1;
      }
    }
  }

  for (JJ = 0; JJ < NUMBER_OF_CLUSTERS; JJ++) {
    Total_MOPS_Not_Using_A_Cluster = 0;

    for (II = 0; II < BLK.MOP_Count; II++) {
      Total_MOPs_Not_Using_A_Cluster += Unused[II][JJ];
    }
    /* IF all the MOPs not use a common cluster, then turn that cluster OFF */
    if (Total_MOPs_Not_Using_A_Cluster == BLK.number)
      ShutOff_Cluster_List[II] = 1;
  }

  return Shutoff_Cluster_List;
}
```

Figure 11: Cluster-shutoff algorithm implemented in the Profiler

5 Experimental Frameworks

5.1 Design Flow & Energy Measurement

Embedded systems, unlike high-performance systems, are small and very sensitive to power and energy differences. Moreover, accurate energy values are necessary to determine the size and strength of the battery required for the embedded processor. A gross overestimation of energy can require the designers to use a larger battery, thus incurring more area and cost. An underestimate can undermine the battery requirement, which requires frequent replacement (or recharging) of the batteries making the system inconvenient for the user. In an independent study done by EE-times, a high-level processor power analyzer had up to 25% error in its calculation. Moreover, previous studies suggest that hardware level (as in RTL level) studies provide 14-24% better power and energy results than pure cycle-count studies [12].

The most accurate way to measure power and energy in an embedded system is to create a prototype of the proposed system in hardware and connect a multi-meter and measure its current. The cost and time of

[2] The ".t.nv" next to the 4th NOP signifies the TAIL bit indicated in Figure 5

fabrication and chip-design can make this a prohibitive effort. Another way to measure power is to create a transistor-level design of the circuit and use SPICE to measure power. To execute ~1,000,000 instructions through a transistor-level design of a processor using Cadence Virtuoso and measure power using SPICE is estimated to take over 31 days on a low-load SPARC Sun fire 280W. This measurement-time cost is not feasible for research today.

There are several RTL-level tools such as Primetime (formerly known as Primepower) and Power-mill [25] that can measure power within 5% accuracy to SPICE [17]. If a processor can be designed, synthesized and verified in Verilog and have its energy consumption measured in Primetime then accurate power values can be achieved. To simulate ~1,000,000 instructions through our Verilog-model (on a low-load SPARC Sun-fire 280W) using Verilog-XL [43] takes approximately 14 hours. Using the execution-time from Verilog-XL and the power values from Primetime, accurate energy estimates can be derived. This is a good compromise between accurate energy values and fast simulation.

CLAW, based on OpenRISC 1200 architecture, is written in Verilog hardware-description language and synthesized by Synopsys Design Analyzer using Artisan SAGE-X 90nm RVT standard-cell library. This standard-cell library is equivalent to low-operating power libraries described by ITRS [44]. Such libraries are most-often used for embedded processors today [35]. This synthesized Verilog is then placed and routed using Cadence Design-Encounter. The output of this step is the parasitic file (SPEF format) that gives accurate capacitance values of the wires inside the processor. The synthesized Verilog-file is then simulated with the appropriate 90nm gates and a test-bench using the Cadence Verilog Simulator. The simulator outputs the switching information in the VCD format as well as number of cycles CLAW took to execute the benchmark. This simulation-step is illustrated in Figure 12.

The VCD file, obtained from Verilog simulation, along with the SPEF file and the synthesized Verilog is input into Primetime and appropriate power-values are obtained. The product of the obtained power values along with the cycle-time and cycle-count results in energy values.

5.2 Benchmark Selection and Execution

To accurately represent embedded-system benchmarks, ten benchmarks from EEMBC benchmark set [45] were used, shown in Table 1. EEMBC is reputed to be the most representative embedded-systems benchmarks available today. These benchmarks are created by a consortium represented by engineers from both industry and academia who are experts on embedded systems. In addition, the consortium have provided specific instructions about the starting point and stopping point of all the benchmarks, along with the number of iterations required for all the loops. The starting point is marked with a "th_signal_start()" function and the stopping point is marked by "th_signal_finished()" function. The C code between these two functions must be the only part whose characteristics are measured. This isolates the actual algorithm from additional noise such as I/O, and makes sure the algorithm is executed completely and adequately.

To execute the benchmarks as per EEMBC specifications in hardware, a CLAW simulator was written in C++. This simulator is used to capture the state (mainly the register-file and the memory state) up to the starting point described by EEMBC. When the reset pin of the processor is set, the appropriate values are written in the register-file and the memory-array[3] (and data-cache). The PC is then pointed by the test bench to the starting point and the memory values are updated using the values obtained by the C++ simulator. The benchmark is executed until the stopping point described by EEMBC.

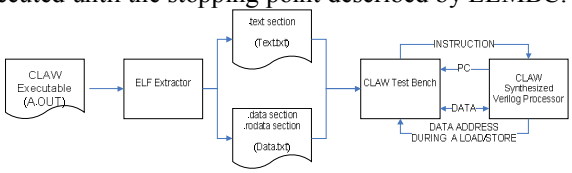

Figure 12: Simulating CLAW using Verilog-XL

Table 1: EEMBC Benchmarks

Benchmark	Description
Aifir01	FIR Filter
conven00	Convolutional encoding
Dither01	Floyd-Steinberg error diffusion Dithering Algorithm
Ospf	OSPF Dijkstra's Algorithm
puwmod01	Pulse Width Modulation Algorithm
rotate	Image Rotation algorithm
routelookup	Packet Routing Algorithms
Rspeed01	Road Speed Calculation
ttsprk01	Tooth-to-Spark tests in automobiles
viterb01	Viterbi Decoder

6. Results

For this paper, a one, two and four-cluster CLAWs were created in hardware. The clock-frequency is fixed at 75 MHz (period: 13 ns) for all the configurations for uniformity. A 13 ns period was chosen because it is the cycle-time at which up to 64 clusters can be incorporated without slack violation. While we only present results for up to 4 clusters (which could have had 2-3x smaller period), this is not an issue because the energy trends discussed in this work still hold as frequency is scaled. Single cluster standard-cell area is 0.52 mm^2. Two and four-cluster CLAWs have areas of 1.05 and 2.13 mm^2, respectively. The UAS scheduler was implemented on GCC using the four cluster-assignment priorities (Sequence, Random, MWP and CWP) for the two and four-cluster configurations. Figure 13 and Figure 14 show the speedup of 2-cluster and 4-cluster CLAW over a

[3] We model memory in the test-bench as an array and connect it directly to the data-cache

single-cluster for the 10 EEMBC benchmarks, respectively.

MWP and CWP schemes give the most speedup across all the benchmarks because they take the code-pattern into account when scheduling the instructions. Routeloop is the most parallel benchmark, while aifir01 shows the least. Figure 15 and Figure 16 show the percentage of copy-operations in the dynamic stream for 2 and 4-cluster CLAW. Sequential and Random placement of UAS results in ~8-28% of the dynamic instruction stream consisting of copy instructions, while MWP and CWP on average had only 0.2%. These copy instructions increase the number of dynamic instructions, which in turn increases the execution-time. Copy instructions create additional true-dependences between instructions between clusters, thus increasing execution time.

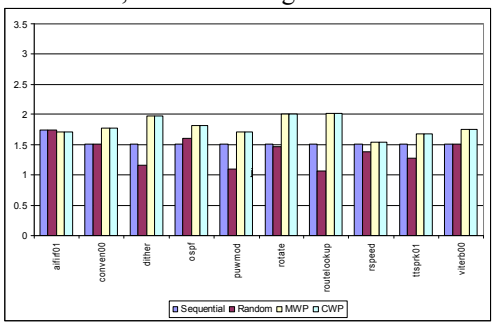

Figure 13: Speedup for EEMBC Benchmarks for 2-Cluster CLAW over single-cluster CLAW

Several predictions claim that static (or leakage) energy will dominate the processors today. Static-energy is claimed to have ~50% contribution to the total energy consumption for 90nm technology, yet as per Figure 17 the static-energy accounts for roughly 15-20% of the total energy. The primary reason for this disparity is that static energy is dominant in the memory hierarchy [26]; dynamic energy is still dominant inside the processor [26]. For CLAW the energy values are measured only for the processor. The memory hierarchy is modeled inside the test-bench and isolated from power-measurement because many embedded and real-time systems have on-chip memories and avoid caches due to their unpredictability and excessive power dissipation.

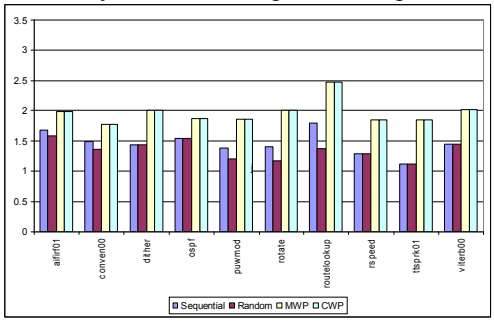

Figure 14: Speedup for EEMBC Benchmarks on 4-Cluster CLAW over single-cluster CLAW

Figure 18 and Figure 19 show the dynamic-energy consumption for two and four-cluster CLAW. Routelookup and OSPF are significantly smaller benchmarks than the rest, which impacts their energy use. Ttsprk01 has the largest dynamic code-size in the group. Static energy distribution is given in Figure 20 and Figure 21. Static energy increased for Sequential and random placement because they generated dependent copy instructions that created more holes in the trace. As the code-size gets larger, there is a potential for more NOPs, which can cause stagnant wires and units, increasing static energy. In all these data, shutoff mechanism is disabled to show the base values.

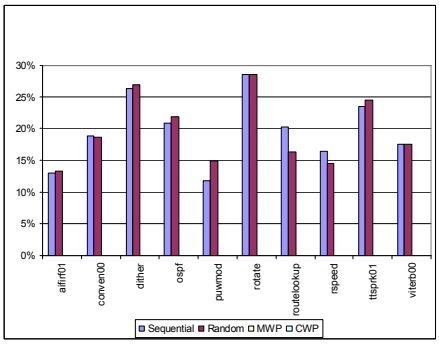

Figure 15: Percentage of Copy Instructions for 2-Cluster CLAW

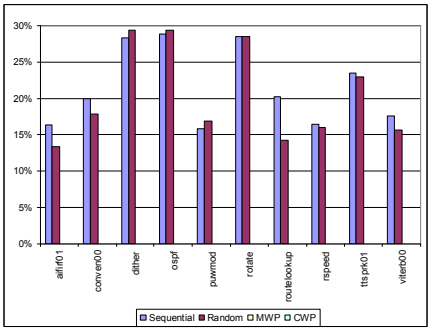

Figure 16: Percentage of Copy Instructions for 4-Cluster CLAW

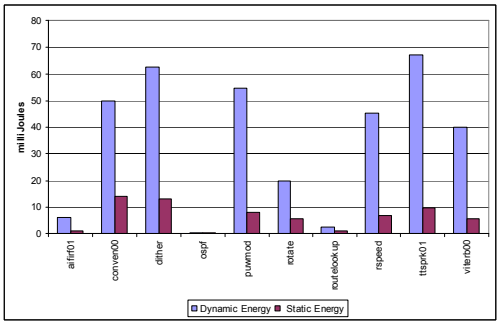

Figure 17: Static and Dynamic Energy Dissipation for Single Cluster CLAW

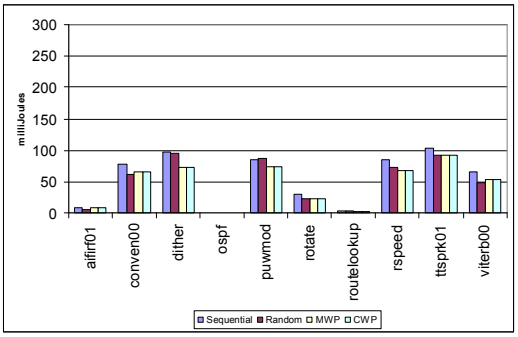

Figure 18: Dynamic Energy Consumption for 2-Cluster CLAW

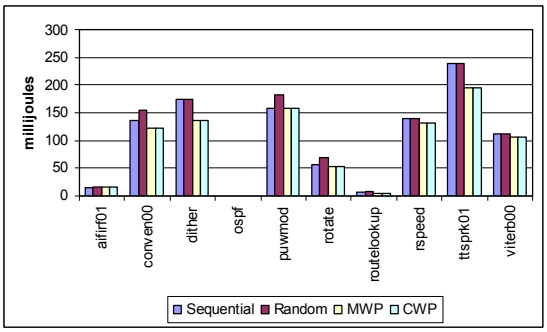

Figure 19: Dynamic Energy Consumption for 4-Cluster CLAW

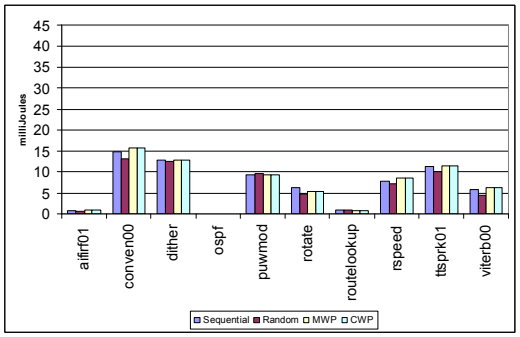

Figure 20: Static Energy Consumption of 2-Cluster CLAW

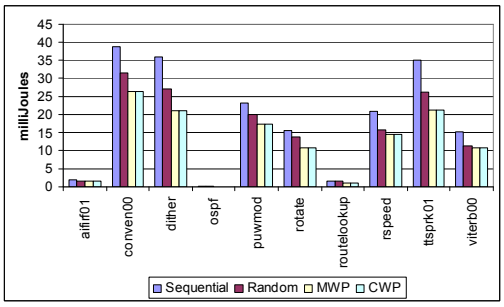

Figure 21: Static Energy Consumption of 4-Cluster CLAW

6.2 Dynamic Cluster Shutoff

To study the energy effects due to dynamic-cluster shutoff, we studied 2-cluster and 4-cluster CLAW for both basic-block-level and function-level shutoff. For the random and sequential placement, the profiler was not able to shutoff any clusters. Due to space constraints, we only show the energy effects of 4-Cluster CLAW for function level and 2-Cluster CLAW for basic-block-level experiments. We only show results for the CWP mechanism since we were unable to find any optimization scenarios for the Random and Sequential placement. MWP was skipped because it followed same utilization trend as CWP.

Figure 22 and Figure 23 show the dynamic and static energy consumption of 4 Cluster CLAW utilizing the shutoff mechanism at function granularity. The values are normalized with the base values of all clusters on (from Figure 19 and Figure 21). All the benchmarks except routelookup turned clusters 2, 3 and 4 off most of the time while routelookup only shutoff 3 and 4. The energy values did scale appropriately with the cluster-shutoff. The static energy increased by approximately 10-20% but the dynamic energy compensated this increase.

Adding shutoff instructions at the start and end of every basic-block was not effective because this resulted in code-explosion. This almost doubled the execution-time for every benchmark and offset the benefit of shutting off clusters. Figure 24 and Figure 25 show our results. The static energy also increased due to the increase in number of NOP. Similar energy increase trend is seen in 4-cluster CLAW due to code-explosion. This increase could be reduced through the use of an optimization pass to remove silent shutoff instructions (those that do not change machine state), but is not explored in this work.

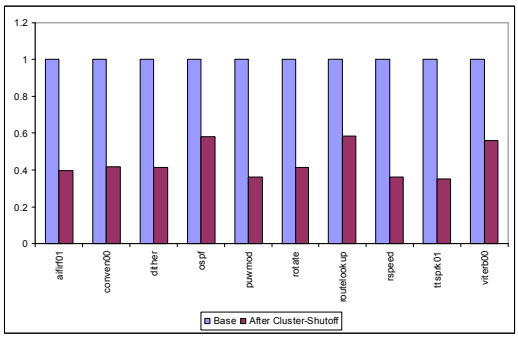

Figure 22: Dynamic Energy Distribution of Function-level Cluster-shutoff for 4-Cluster CLAW

There is another factor that is inherent to GCC that resulted in such code-explosion. GCC, unlike several proprietary compilers such as TI Compiler (TI-CC) [13] or the ARM-CC[42], does not perform any high-level optimization, such as loop-fusion, on the code[33]. The parser directly decomposes the C statements into trees in GIMPLE format then tries to apply optimization. At this stage, it is generally too late for such optimizations. Thus, GCC code on average contains more basic-blocks with fewer instructions when compared to ARM-CC or TI-CC basic blocks. This translates to more shutoff instruction insertions at this granularity, resulting in further code-explosion.

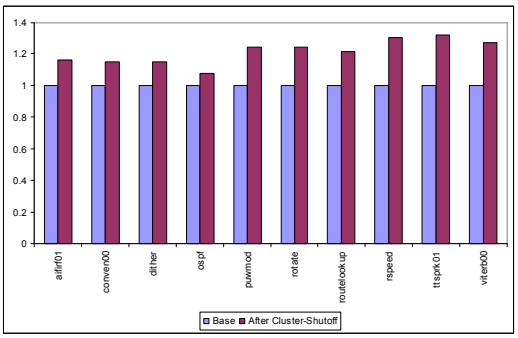

Figure 23: Static Energy Distribution with Function-level shutoff for 4-Cluster CLAW

From this research it is clear that the function level cluster shutoff heuristic is superior to the basic-block policy in terms of energy and provides significant benefits over the baseline with no shutoff. Additionally, function level shutoff has minimal impact on performance since only 1 MOP per function is inserted, translating to less than a 0.2% increase in the overall dynamic code stream.

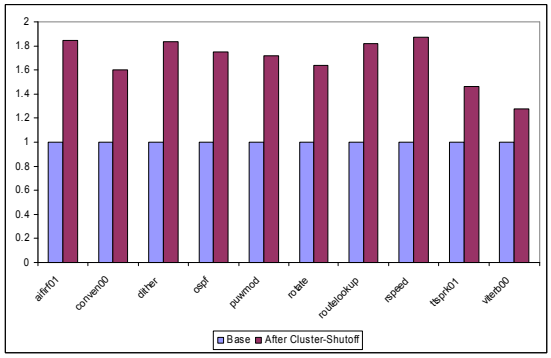

Figure 24: Dynamic Energy Distribution with basic-block-level shutoff for 2 Cluster CLAW

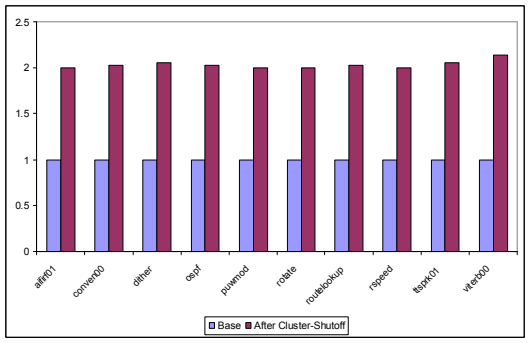

Figure 25: Static Energy Distribution with basic-block-level shutoff for 2 Cluster CLAW

To exploit the dynamic shutoff mechanism of CLAW, we wanted to see the utilization of each cluster. Random and Sequential placement uses all the clusters in approximately equal fashion, but MWP and CWP uses cluster-1 most of the time. MWP and CWP have similar results because most of the instructions in the CLAW architecture are able to execute in a single cycle. For example, puwmod using MWP has at least one non-NOP OP in cluster 1 100% of the time, and none in cluster-2 (never used).

7 Conclusions

In this work, we provided a RTL level VLIW-embedded processor, CLAW, that is synthesized and placed and routed, then measure its power and energy using industry-strength tools. This processor is able to dynamically scale its issue-width, enabling effective energy aware compilation. In addition, we provided compiler and optimizing tools that are able to exploit this scalability and utilize this energy efficient processor without sacrificing any performance. By having a dynamic scheme, we can save considerable time compared to previous approaches which require a redesigning of the processor when switching applications or adding new application.

Using this processor, we also showed that hardware-level analysis is far more accurate than software simulation, providing better insight into energy dissipation within a processor. Using this hardware-level design we were able to debunk the myth that static-energy domination is as much of a concern in embedded processors at 90 nm as it is in high-performance processors, which otherwise would not have been possible in a software-only approach.

References

[1] R. Bahar, S. Manne, "Power and Energy Reduction Via Pipeline Reduction," *ISCA,* 2001

[2] D. Albonesi, "Dynamic IPC/Clock Rate Optimization," *ISCA,,* 1998

[3] A. Bechini, T. M. Conte, C. A. Prete, "Opportunities and Challenges in Embedded Systems," *MICRO*, 2004

[4] S. Bhunia et al., "A Novel Low-Power Scan Design Technique Using Supply Gating," *ICCD,* 2001

[5] A. Bona et al., "Energy Estimation and Optimization of Embedded VLIW Processors based on Instruction Clustering," *DAC*, pp. 886-891, 2002

[6] A. Buyuktosunoglu et al., "An Adaptive Issue Queue for Reduced Power at High Performance," *Proc. of First Intl. PACS,* 2000

[7] A. Capitianio, N. Dutt, A. Nicolau, "Partitioned Register Files for VLIWs: A preliminary Analysis of Tradeoffs, " *In MICRO*, pp.292-300, 1992

[8] C. Chang, D. Marculescu, "Design and Analysis of a Low Power VLIW DSP Core," *Proc. of Emerging VLSI Technologies and Architectures,* 2006

[9] R. Chassaing, "DSP Applications using C on the TMS320c6x DSK"

[10] N. Clark et al., "OptimoDE: Programmable Accelerator Engines trough Retargetable Customization," Hot-chips, 2004

[11] O. Colavin, D. Rizzo, "A Scalable Wide-Issue Clustered VLIW with a Reconfigurable Interconnect," *CASES,* pp.148-158, 2003

[12] J. Cong et al., "Microarchitecture Evaluation with Physical Planning," *DAC,* pp. 32-35, June 2-6, 2003
[13] G. Davis, "Writing Reliable C/C++ system Code: Nut and Bolts," TI Developer Conference, 2007
[14] J. R. Ellis. Bulldog: A Compiler for VLIW Architectures. The MIT Press, 1985
[15] P. Faraboschi, G. Brown, J. A. Fischer, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," *ISCA,* pp. 203-213, 2000
[16] J. Fischer, P. Faraboschi, G. Desoli, "Custom-Fit Processors: Letting Application Define Architectures," *MICRO,* pp. 324-335, 1996
[17] R. Goering, "Synopsys launches more powerful power-analysis tool," EE-times, 2000
[18] J. Gonzalez, A. Gonzalez, "Dynamic Cluster Resizing," *ICCD*, 2003
[19] W. A. Hawanki, S. Banerjia, T. M. Conte, "Treegion scheduling for wide-issue processors," *ISCA,* 1998
[20] M. Horowitz, W. Dally, "How Scaling Will Change Processor Architecture," *ISSCC, 2004*
[21] Z. Hu, et al., "Microarchitectural Techniques for Power Gating of Execution Units," *ISLPED*, pp. 32-37, 2004
[22] A. Iyer, D. Marculescu, "Power Aware Microarchitecture Resource Scaling," *DATE,* pp. 190-196, 2001
[23] D. Jain et al., "Automatically Customizing VLIW Architectures with Coarse Grained Application-specific Functional Units," *Proc. of Software and Compilers on Embedded Systems,* 2004
[24] K. Kailas, K. Ebcioglu, A. Agrawala, "CARS: A New Code Generation Framework for Clustered ILP Processors, " *HPCA*, p.133-143, 2001
[25] M. Keating et al. "Low Power Methodologies Manual," Springer Publishing, July 2007
[26] N. S. Kim et al., "Leakage current: Moore's law meets static power," IEEE Computer, Vol. 26, Issue 12, 2003
[27] H. Li et al., "Deterministic Clock Gating for Microprocessor Power Reduction," *HPCA,* 2002
[28] T. Li, L. K. John, "Routine based OS-aware Microprocessor Resource Adaptation for Run-time Operation System power savings," *ISLPED,* 2003
[29] Y. Luo et al., "Low Power Network Processor Design using Clock Gating," *DAC*, 2005
[30] S. Manne, A. Klauser, D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," *ISCA,* 1998
[31] S. Mukopadhyay et al., "Gate Leakage Reduction for Scaled Devices Using Transistor Stacking," *IEEE Transactions on VLSI,* pp. 716-730, vol. 11, No. 4, August 2003
[32] R. Nagpal, Y. Srikant, "Compiler-Assisted Leakage Energy Optimization for Clustered VLIW Architectures," *Proc. of IEEE International Conference on Embedded Software*, 2006
[33] D. Novillo, "GCC Internals," *CGO (Presentation),* 2007
[34] E. Ozer, S. Banerjia, T. M. Conte, "Unified Assign and Schedule: A New approach to Scheduling for Clustered Register File Microarchitectures," *MICRO,* 1998
[35] L. Pickup and S. Tyson, "Hot Chips? … Not!" *Chip Design Magazine*, pp. 26-29, August/September 2004
[36] M. Powell et al., "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Micron Cache Memories," *ISLPED*, 2000
[37] M. C. Rosier, T. M. Conte, "Treegion Instruction Scheduling in GCC," *Proc. Of the GCC Developers Summit, 2006*
[38] C. Su, C Tsui, A. Despain, "Saving Power in the Control Path of Embedded Processors," *IEEE Design & Test of Computers,* pp. 24-30, 1994
[39] R. M. Stallman, "GNU Compiler Collection Internals for GCC 4.0.2," 2006, FSF Press (Obtained from GCC Source)
[40] A. Terechko, M. Garg, H. Corporaal, "Evaluation of Speed and Area of Clustered VLIW Architectures," *Proc. of International Conference on VLSI Design,"* 2005
[41] V. Zyuban and P. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures," *IEEE Trans. on Computers*, March 2001
[42] "Writing Efficient C Code for ARM," Application Note 34, Document No: ARM DAI 0034A, 1998
[43] "Verilog-XL User guide," Product Version 4.0, 2002, Published by Cadence (www.cadence.com)
[44] International Technology Roadmap for Semiconductors (ITRS), http://www.itrs.net
[45] Embedded Benchmark Consortium, http://www.eembc.org
[46] Tensilica Xtensa 7 (LX2), http:// http://www.tensilica.com/products/xtensa/index.htm