Low-Cost Hardware Profiling of Run-Time and Energy in FPGA Embedded Processors

Mark Aldham, Jason Anderson, Stephen Brown, Andrew Canis
ECE Department, University of Toronto
Toronto, ON, Canada
{aldhamma, janders, brown, acanis}@eecg.toronto.edu

***Abstract*—This paper introduces a low-overhead hardware profiling architecture, called LEAP, that attains real-time cycle and energy profiles of an FPGA-based soft processor. A novel technique is used to associate profiling data with specific functions in a way that is area- and power-efficient. Results show that relative to a previously-published hardware profiler, our design uses up to 18× less area and 8.6× less energy. LEAP is designed to be extensible for a variety of profiling tasks, three of which are investigated in this paper. We also demonstrate the utility of LEAP in the context of hardware/software co-design of processor/accelerator FPGA-based systems.**

I. Introduction

Field-programmable gate arrays (FPGAs) are a widely used technology in the design of embedded systems due to their improving speed, density and power, steadily decreasing cost, and their programmability which reduces a product's time-to-market. The advent of FPGA soft processors permits the creation of hybrid systems within a single FPGA fabric, comprising software running on a processor, as well as custom digital hardware for accelerating computations and improving energy-efficiency [11], [17]. It is estimated that over 40% of today's FPGA designs contain embedded processors [28]. There is a need, then, for techniques that can profile the dynamic behavior of a program as it executes on an FPGA soft processor, with the profiling results used for two key purposes: 1) improving the performance of the program itself, and 2) in the context of hardware/software co-design to aid in partitioning a program's computations into those suitable for software versus hardware.

Profiling can be performed in either software or hardware. In the software approach, the program being profiled is modified to gather profiling data during its execution. In the hardware profiling approach, on the other hand, the program executes in its original unmodified form at full speed on the processor. The processor itself is augmented with additional circuitry that automatically gathers profiling data as the program executes. The disadvantage of hardware profiling is higher processor hardware complexity; the advantage is both superior speed and accuracy when compared to software profiling. Hardware profiling is generally a necessity in embedded computing applications, where hard real-time constraints may require the system to run at full speed, making instrumentation from software approaches infeasable.

In this paper, we introduce the Low-overhead and Extensible Architecture for Profiling (*LEAP*): a new hardware profiler for FPGA-based embedded processors. An important goal for any hardware profiler is low-overhead: the amount of additional circuitry added to the processor, and the energy consumption of that circuitry should be as small as possible. A novel aspect of our design is the use of perfect hashing in hardware to associate code segments with hardware counters that track dynamic run-time behavior. The use of hashing leads to considerably smaller overhead versus previously-published hardware profiler designs. Specifically, relative to SnoopP [24] (a hardware profiler for FPGA-based processors), our design requires up to 18× less area and 8.6× less energy. Moreover, our design is extensible in that it can be easily adapted to profile different run-time characteristics, including instruction and cycle counts, cache stalls, pipeline stalls, and energy consumption.

Recent years have seen an increased research focus on the design of hybrid processor/accelerator systems, where custom hardware is used in tandem with a processor to achieve higher computational throughput and energy-efficiency. In the design of such systems, one generally starts with a software-only solution and uses a profiler to identify compute and energy-intensive code segments that would benefit from hardware implementation. LEAP has been developed as part of a larger open source high-level synthesis (HLS) project [3], where the long-term vision is to provide automated and dynamic hardware/software partitioning and synthesis of processor/accelerator systems. In this paper, we give preliminary results for such hybrid systems, where LEAP has been used to inform the partitioning decisions.

The remainder of this paper is organized as follows: Section II presents related work. Section III introduces the profiling architecture, outlines its method of operation, and demonstrates its extensibility. Experimental results appear in Section IV. Conclusions and suggestions for future work are given in Section V.

II. Background

A. Prior Work on Profiling

As mentioned above, profiling can be performed with either software or hardware techniques. Software approaches typically incur run-time overhead due to instrumentation, and may produce results with reduced accuracy if sampling is employed. Hardware profiling is typically non-intrusive, and produces accurate results at the cost of additional circuitry. Two software tools are described below, as are four hardware profilers.

Gprof is a popular software-based profiler [1] that uses a sampling-based approach whereby the processor's program counter (PC) is "sampled" at specific intervals. Additional code that must be added to a program, in addition to the time required for sampling, is associated with a significant run-time overhead – program execution may proceed multiple

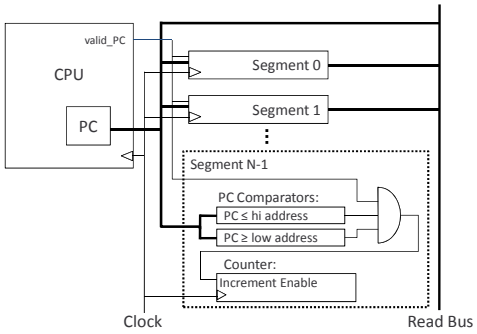

Fig. 1. Architecture of the SnoopP profiler [24].

times slower than normal. In gprof, sampling less often will reduce run-time overhead, at the expense of accuracy.

Pin [18] is a software-based dynamic instrumentation system through which, using architecture-generic APIs, instrumentation tools (including profilers) can be created. This API enables the observation of all architectural states of a process, including the contents of registers, memory, and control flow. Pin uses just-in-time (JIT) compilation, taking native executables as input, and compiling traces as required for execution.

SnoopP [24] is an FPGA-based hardware profiler designed for use with the Xilinx MicroBlaze soft processor [30]. It allows arbitrary code regions to be profiled for cycle counts. The address ranges of each region, as well as the number of regions to profile, are entered as parameters in the VHDL code of the profiler that must be set pre-synthesis. During execution, a large number of comparators check the program counter of the processor in each clock cycle to see if it falls into any of the specified regions – two wide comparators are used for *every* region profiled. If the PC falls in a profiled region, a 64-bit counter corresponding to that region is incremented. The general architecture of SnoopP is shown in Fig. 1. AddressTracer is an adaptation of SnoopP that permits measurement of the number of cycles spent in a function *plus* its descendants [23].

Other recent work includes Airwolf [26], which is a software FPGA-based profiler for the Altera Nios II soft processor [7] that inserts software drivers into each software function call/return to enable/disable individual counters. Comet [12] is another software FPGA-based profiler for the Altera Nios [5] soft processor that requires the user to modify the application to be profiled with pragma-like labels to specify the starts and ends of regions of interest.

For energy profiling, [25] describes a profiling-based methodology for power reduction within non-FPGA embedded processors using instruction-level power models. Another power estimation methodology to predict the power consumption of non-FPGA embedded processors is presented in [22], which describes the creation of a power data bank that stores the energy consumption of built-in library functions, user-defined functions, and basic instructions. The work in [21] describes a method to estimate the energy consumption of an FPGA-based processor using the Xilinx Microprocessor Debugger (XMD) TCL interface (as opposed to on-chip profiling) to combine an instruction trace with a power look-up table.

Our profiler, LEAP, both differs from and improves upon prior hardware profilers in two ways: it has very low overhead in both area and power, and it is extensibile to handle multiple different profiling metrics, including both run-time-oriented metrics and energy.

TABLE I
SYNTHESIS RESULTS FOR TIGER PROCESSOR ON CYCLONE II (2C35).

Total logic elements	13,422 (33,216)
Total registers	6,455 (33,216)
Total memory bits	225,280 (483,840)
Embedded Multipliers	16 (70)
Maximum Frequency (MHz)	75.59 (402.5)

B. Tiger MIPS Processor

To develop and test the proposed profiler, a soft processor was required. Soft processors from the commercial FPGA vendors could not be used due to their proprietary source code. ARM processors were likewise excluded for IP considerations [10], [27]. We settled on using a MIPS processor and evaluated a number of open source MIPS implementations. The Tiger MIPS processor [20], developed at Cambridge University, was selected due to its high-quality Verilog code, documentation, mature development ecosystem, and its coverage of the MIPS instruction set.

Tiger MIPS is a 32-bit 5-stage processor that supports the MIPS1 instruction set [19]. A JTAG UART-based communication interface enables command-line data transfer between a host computer and the on-chip processor. This interface is used to transfer programs and data to the Tiger system, perform GDB-based debugging, and retrieve profiling results (added for use with LEAP). Tiger MIPS is designed to work on the Altera Development and Education DE2 board, a widely-used platform comprising a Cyclone II 2C35 FPGA [6] and 8 MB off-chip SDRAM memory. Tiger MIPS uses on-FPGA block memory to implement 8 KB direct-mapped instruction and data L1 caches. Cache misses are directed to the on-board SDRAM (four words are burst on a cache miss). Implementation details for Tiger on the Cyclone II are given in Table I, with the chip capacities shown in parentheses.

Although we use a MIPS processor as the test vehicle in this research, our profiler design is not tied to MIPS and it is straightforward to adapt the design to other processor architectures. A candidate processor must provide access to its PC, instruction bus, and cache stall signals, in addition to a signal indicating whether the current PC is valid. Opcode-specific features of LEAP were developed using MIPS opcodes, but these can be easily retargeted to a new processor with knowledge of its instruction set architecture.

C. High-Level Synthesis (HLS) Framework

Hardware profiling with LEAP is a key component of a HLS design tool [3] that enables the acceleration of software through automated synthesis of custom hardware accelerators. The HLS tool requires multiple (mostly automated) steps to create a hybrid processor/accelerator system:

1) The user compiles a standard C program to a binary MIPS executable using the LLVM compiler [15].
2) The software executable is run on the MIPS processor; execution is profiled with LEAP to determine the code segments that would most benefit from hardware acceleration to improve program throughput and energy.
3) The HLS tool reads a configuration file representing the hardware/software partitioning and compiles the required segments to synthesizable Verilog RTL.

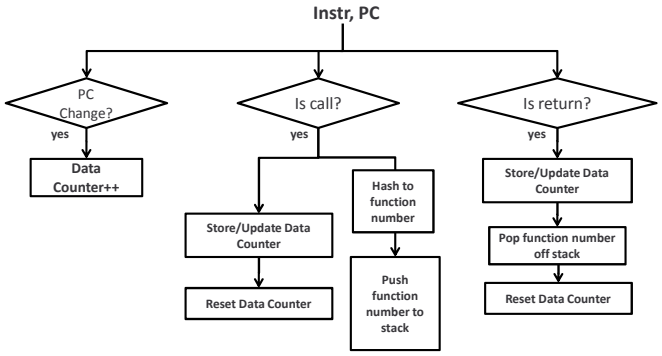

Fig. 2. High-level flow chart for instruction-count profiling.

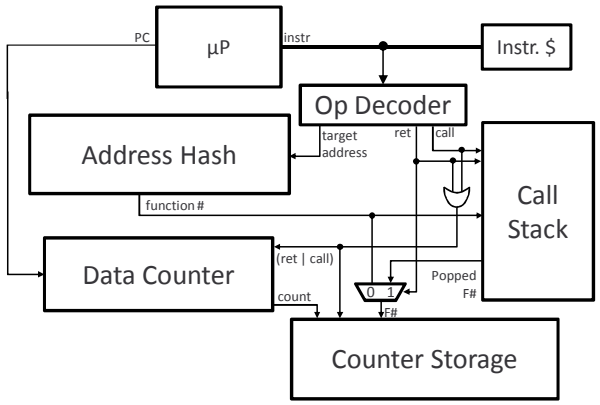

Fig. 3. Modular view of profiling architecture.

4) Verilog RTL is synthesized to an FPGA implementation using Altera commercial tools [9].
5) The original software code is modified such that calls to accelerated functions are replaced with calls to wrapper functions that communicate with the accelerators. This modified source is compiled to a MIPS executable.
6) The hybrid processor/accelerator system executes on the FPGA.

III. LEAP Design

We begin by describing LEAP's high-level functionality, and then delve into its hardware architecture and low-level details.

A. Method of Operation

LEAP profiles the execution of the program by monitoring the processor's program counter and also its instruction bus. During execution, LEAP maintains a counter, called the *Data Counter*, that tracks the number of times an *event* has occurred. In the case of instruction count profiling, each event corresponds to a change in the program counter value. For cycle count profiling, each successive clock cycle is an event.

LEAP organizes the collected data on a per-function basis by allocating a storage counter for each software function. The question that naturally arises is how to determine the counter associated with a particular function. Prior hardware profilers, such as SnoopP [24], use a large number of comparators to associate PC address ranges with individual counters. A key innovation in LEAP is the use of *perfect hashing* in hardware to associate functions with counters, resulting in significantly less hardware overhead when compared to previously published work. The perfect hash produces a function number based on the function's starting address.

Function boundaries are identified by decoding (in hardware) the executing instruction to determine if it is a function call or return. If a call is detected, the Data Counter is added to any previously stored values associated with the function containing the call instruction (from previous invocations of the function). The Data Counter is then reset to 0 to begin counting the events in the called function. If a function return is detected, the Data Counter value is added to the counter associated with the current function, and once again the Data Counter is reset.

The high-level operation of LEAP is shown in Fig. 2 for instruction count profiling. In the left branch, the PC is monitored and the Data Counter is incremented each time a different instruction executes. The middle branch detects function calls. When a function call is identified, the Data Counter is added to the counter (stored in on-chip memory) associated with the calling function; this association is determined through perfect hashing. The Data Counter is then reset so that it now represents only instructions executed in the called function. At the same time, the calling function number is stored to a stack so that when the called function returns, the profiler is able to keep track of function context. Finally, the target function address is hashed into a function number so this function can be associated with a counter. The right branch is used to detect function returns. When a function executes a return, the Data Counter is added to the counter corresponding to the returning function. Popping from the aforementioned stack yields the number of the function that will be returned to, setting the current function context. The Data Counter is again reset.

Although LEAP is designed to provide per-function profiling results, finer granularity could be achieved in future through the creation of a new profiling scheme, loop profiling. In this scheme, profilable regions are associated with loops, and an iteration of a loop is indicated by the execution of a short backwards branch (as opposed to a function call).

B. Profiling Architecture

The hardware architecture of LEAP is composed of 5 modules, as illustrated in Fig. 2.

- The **Operation Decoder** (labeled *Op Decoder*) monitors the instruction bus to determine whether a call or return has been executed. Function calls are executed by either the *jump-and-link* (JAL) or the *jump-and-link-register* (JALR) instructions, while a return is done using the *jump register* (JR) instruction when the destination register is set to register 31 (designated for return addresses).
- The **Data Counter** module performs the actual measurement of profiling data. The remainder of the architecture is used to properly associate this data with a particular function. In this way, the implementation of a new profiling task only requires the modification of this module. For cycle-accurate profiling, the Data Counter increments every time the PC changes (instruction count profiling), every cycle (cycle count profiling), or every cycle while stalling (stall cycle profiling).
- The **Counter Storage** module is used to compactly store the data collected by the Data Counter module. At any

function context switch, the entry corresponding to the current function is read, added to the current value of the Data Counter, and written back to the same entry. Through this design, all data for a particular function is accrued correctly, regardless of the number of calls and returns.

- The **Address Hash** module is used to map a function's memory address to a unique index, where the unique index is used to find the function's corresponding counter in the Counter Storage module (described above). We give details on this module in the next section.
- The **Call Stack** is used to keep track of the currently-executing function, since a return instruction does not indicate which function will be returned to, only the address to jump to. Function indices (from the Address Hash module described above) are pushed onto the stack during function calls and popped-off during function returns.

C. Address Hash Module

To store profiling data for each function in an efficient manner, the indices used to represent the functions must be contiguous over a compact range. The address of the first instruction in each function is used to identify the function, but if these addresses were directly used to index into the Counter Storage RAM, the address space of the RAM would need to be as large as that of the processor; for a 32-bit PC, this would require 2^{32} bits. Such a large address space is impossible to realize using memory bits on an FPGA and therefore, the address space must be translated to a more compact range. This translation is done through perfect hashing.

A hash function is a mathematical function that maps a large set of data to a smaller set. A perfect hash function is one which performs a unique mapping so that no two inputs map to the same output. Perfect hashing, in our case, is used to convert each function's address into an index in the range 0 to N-1, where N is the number of functions in the program rounded to the next power of two.

The perfect hash generator in [16] is used to create an application-specific perfect hash. This generator is based on a static hashing algorithm, shown in Fig. 4, which uses 6 parameters that can be tailored to the specific application to guarantee collision-free hashes. The input to the hash function, `funcAddr`, represents the address of the first instruction in a function; the output of the hash is the function index. We include the code in Fig. 4 simply to illustrate that the function is very efficient to implement in hardware as it only requires bitwise `AND` and exclusive-`OR` operations, logical shifts, and one memory access. `tab` in Fig. 4 is an N-element byte array computed by the hash generator.

During the compilation of a program to be profiled, all function addresses are extracted from the executable binary and passed through the perfect hash generator, producing the customized set of parameters for use with the hashing scheme in Fig. 4 (which we implement in hardware). These parameters must be generated for each application that is to be profiled in order to perform the unique mapping from function addresses to specific counters. The hash parameters are loaded into memory along with the application's binary. At execution time of the program, the profiler initializes the Address Hash module with these parameters. While the generation of the hash parameters is done at the compilation stage, the actual hashing from function addresses to function numbers is done in hardware. No modifications of this hardware circuit

```
// Input: funcAddr, output: funcNum
// Parameters: tab[], V, A1, A2, B1, B2
int doHash (unsigned int funcAddr) {
        unsigned int a, b, funcNum;

        funcAddr += V;
        funcAddr += (funcAddr << 8);
        funcAddr ^= (funcAddr >> 4);
        b = (funcAddr >> B1) & B2;
        a = (funcAddr + (funcAddr << A1)) >> A2;
        funcNum = (a^tab[b]);

        return funcNum;
}
```

Fig. 4. Parameterized hashing algorithm used by the perfect hash generator (shown in C for clarity).

(e.g. resynthesis or reprogramming) are needed to profile a new application.

D. Profiling Extensions

LEAP is easily extended to measure clock cycles and stall cycles per function, only requiring the Data Counter to be modified. The total number of cycles executed per function is collected by incrementing the counter every clock cycle, and the total number of cycles consumed by cache stalls is found by incrementing the counter every clock cycle in which either cache has its stall signal asserted (a signal easily accessible in Tiger MIPS).

LEAP was further extended to perform energy profiling. An instruction-level power database was created off-line using timing-driven simulations of programs running on Tiger MIPS implemented on Altera's Cyclone II FPGA. The simulations were done using Mentor Graphics' ModelSim [2], producing switching activity data files (VCD) that were then provided to Quartus PowerPlay – Altera's power analysis tool. Through this methodology, we were able to compute the average dynamic power consumed by each instruction in the MIPS instruction set across a suite of 13 C-language benchmarks (described in more detail in Section IV). Our energy profiling in LEAP works by tracking not only the number of instructions per function, but also the types of instructions executed, which we combine with the instruction-level power data to produce an overall energy estimate.

To gauge the power-per-instruction as accurately as possible, stalls must be considered. In general, during a stall, the processor consumes less power, potentially making it appear as though the instructions "in flight" during a stall consume less power than they actually do. Stalls can be defined in two ways: 1) cache stalls in which the processor waits during a cache miss, or more broadly, as 2) pipeline stalls which can be due a cache miss, or also due to data hazards, and multi-cycle instructions (division and multiply in Tiger MIPS). We discuss both types of stalls below. During the timing simulations we used to build the instruction-level power database, we use ModelSim TCL commands to pause VCD (switching activity) generation during stalls, thereby removing the stalls from the per-instruction power data.

After profiling the dynamic power of each instruction, we observed that certain instructions dissipate similar amounts of power. We partitioned the instructions into six groups, and chose a power value for each group – chosen as simply the average dynamic power of all instructions in the group. The groupings were chosen to minimize the standard deviation

TABLE II
DYNAMIC POWERS FOR INSTRUCTIONS CONTAINED IN GROUP E OF INSTRUCTION-LEVEL POWER DATABASE, MEASURED IN MILLIWATTS.

Instr.	Power
lhu	75.03
lbu	75.16
sllv	75.96
mult	77.37

TABLE III
INSTRUCTION-LEVEL POWER DATABASE FOR THE MIPS1 INSTRUCTION SET, MEASURED IN MILLIWATTS.

Group	Avg. Power	Std. Dev.	Avg. Power	Std. Dev.
A	47.27	1.65	38.45	1.20
B	57.32	1.96	57.53	2.62
C	62.14	0.99	62.82	1.43
D	66.02	1.40	68.03	2.47
E	71.07	3.51	75.88	1.08
F	83.74	2.61	101.60	3.61
STALLS	31.85		44.34	
	(a) Cache stalls		(b) Pipeline stalls	

within each group, which in turn reduces the error introduced by averaging. For each function being profiled, we use six counters to track the number of instructions in each group executed within the function. A seventh counter is used to track the number of cycles spent stalling in each function. The use of a limited number of groups of instructions allows us to use fewer counters, reducing hardware overhead. Using six groups allowed for a sufficiently low power deviation within each group, as seen in Table III. Different numbers of groups can certainly be used for different overhead and accuracy trade-offs.

As an example, Table II shows a collection of instructions that have been grouped together, and Table III shows the average power consumed by each group of instructions. Part (a) of the figure shows the power values when only cache stalls are accounted for; part (b) gives the results when all pipeline stalls are accounted for. Observe that, when only the cache stalls are considered, the power values are generally lower.

To accommodate the groupings of instructions, the Data Counter module was modified as shown in Fig. 5. Instruction decoding was added so that the correct group counter is incremented as each instruction enters the execute stage. The seven counter values are concatenated into a single wide word that is fed to the Counter Storage module (see Fig. 3). The complete description of the instructions in each group could not be included for lack of space, however, the groupings are described in [4].

Finally, further to the extensions above, LEAP is able to produce two types of profiles. In addition to the standard "flat" profile that accounts for the events occurring within a function, LEAP is capable of *hierarchical* profiling, in which the data stored for each function represents the data for that function plus all of its descendant functions.

IV. EXPERIMENTAL STUDY

We evaluate the proposed profiler design in four different ways. First, we verify its accuracy by comparing the cycle profiling results produced by LEAP with those produced by our own implementation[1] of SnoopP (described in Section II-A). Then, we examine the area, speed and energy overheads of both LEAP and SnoopP. Next, we describe the results for energy profiling with LEAP. Finally, we show LEAP's utility in hardware/software partitioning.

TABLE IV
COMPARATIVE RESULTS OF LEAP VS. SNOOPP FOR THE BENCHMARK "ADPCM".

Function	LEAP	SnoopP	Diff.	LEAP	SnoopP	Diff.
decode	63,863	63,863	0	3,656	3,652	4
encode	62,193	62,204	-11	4,245	4,271	-26
upzero	32,614	32,620	-6	948	942	6
filtez	14,708	14,708	0	695	691	4
quantl	14,177	14,175	2	737	723	14
uppol2	11,209	11,209	0	230	233	-3
uppol1	8,097	8,097	0	127	126	1
main	5,122	5,122	0	1,814	1,829	-15
adpcm_main	4,588	4,588	0	1,641	1,626	15
filtep	3,655	3,655	0	56	55	1
logscl	3,425	3,425	0	234	234	0
logsch	3,100	3,112	-12	134	133	1
scalel	2,902	2,902	0	288	292	-4
reset	2,755	2,768	-13	2,332	2,341	-9
abs	1,155	1,155	0	56	55	1
Total	233,563	233,603	-40	17,193	17,203	-10
	(a) Cycles			(b) Stall Cycles		

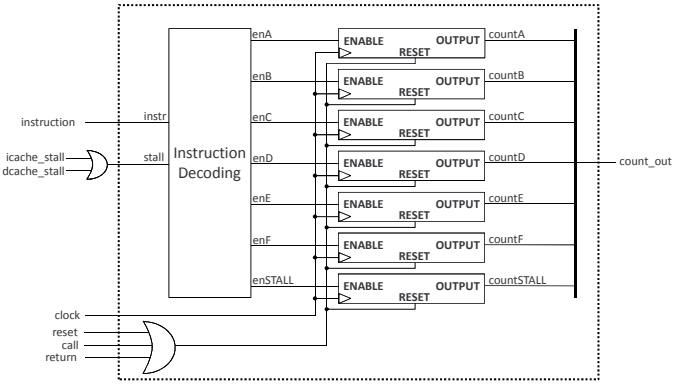

Fig. 5. Data Counter module configured for power profiling when considering all pipeline stalls.

For all results in this section, we target the Altera Cyclone II FPGA and profile programs on the DE2 board (which contains 8 MB of SDRAM). A set of 13 C-language benchmarks are used: the twelve CHStone [14] benchmarks plus Dhrystone [29], a traditional integer benchmark. CHStone represents a wide array of applications, including encoding and decoding, double-precision mathematics, encryption, and image decompression.

A. Accuracy Verification

To verify accuracy, experiments were performed using counter widths (CW) of 32 bits, with the maximum number of functions (N) set to 64, for both SnoopP and LEAP. The CW and N settings are more than enough to accommodate the benchmarks. For LEAP, the maximum stack depth was set to 32. We used both profilers to measure the total number of cycles spent in each function in each benchmark, as well as the number of stall cycles. Sample results for one of the

[1]In this case, SnoopP is applied to profile Tiger MIPS instead of Xilinx MicroBlaze.

benchmarks, *adpcm*, are shown in Table IV. The left side of the table shows total cycles; the right side of the table shows stall cycles. As shown, only slight differences in cycle counts were observed between the two profilers (less than 0.1%, on average), which we attribute to small variations in SDRAM memory access times on the DE2. Similar results were observed for all benchmarks; this data can be found in [4]. These results confirm that LEAP correctly measures cycle counts.

B. Overhead Comparison

We evaluate the area, speed and energy overheads of LEAP and SnoopP for different counter widths, CW, and numbers of counters, N (i.e. the number of functions being profiled). For this analysis, the profilers were synthesized in isolation, without the MIPS processor. Area overheads were gathered from the reports produced by Altera's Quartus II tool [9], and include the number of logic elements (LEs), registers, and memory bits. Note that an LE in Cyclone II contains a 4-input look-up-table (LUT), which is a small memory capable of realizing any logic function of up to four variables. Memory bits refers to the number of used bits in block RAMs on the Cyclone II.

The overhead results are given in Table V. Part (a) of the table gives results for LEAP, part (b) gives results for SnoopP, and part (c) compares the two profilers. Looking first at the area results in part (a), observe that the number of LEs and registers is relatively constant for LEAP as the size of the profiler is increased. Only the number of memory bits required grows significantly as CW and N are increased. In part (b) of the table, we see that SnoopP's size grows rapidly as CW and N are increased, due primarily to the number and size of comparators in SnoopP. Note that SnoopP does not use block RAMs, and therefore has no *Memory bits* column. In part (c) of the table, we see that with respect to the number of LEs, LEAP is up to 97% smaller than SnoopP for the case of 64-bit counters tracking 256 functions. The two profilers are closest in LE count when counters are 32-bits wide and the profiler is tracking 32 functions. In this case, LEAP is still 56% smaller than SnoopP in terms of LEs.

Looking solely at the number of LEs is not entirely accurate, as SnoopP does not use block RAMs on the FPGA – all of its profiling data is stored in registers. In an attempt to account for this, consider that a 4-input LUT contains 16 SRAM cells, and therefore, one could (pessimisstically) assume that each memory bit used in LEAP is equivalent to $\frac{1}{16}$ of a LUT in terms of area. A memory bit is, in fact, far less costly than this, as the decoding circuitry in an SRAM memory is amortized across thousands of bits, whereas multiplexer-based decoding circuitry is needed for each LE. Nevertheless, we can compute an *inflated* LE count for LEAP by adding LEAP's LE usage to its memory bit count divided by 16. In the *Total Area* column on the right side of Table V, we present the ratio of the inflated LE count in LEAP, to the LE count in SnoopP. Observe that even under these assumptions (which are unfavorable to LEAP), LEAP consumes at most about 2× *less* area than SnoopP.

The area data is shown graphically in Fig. 6, with LE count on the vertical axis and the profiler parameters on the horizontal. Three curves are given: LE count in LEAP, inflated LE count in LEAP (defined above), and LE count in SnoopP. The area savings of LEAP versus SnoopP are apparent in the

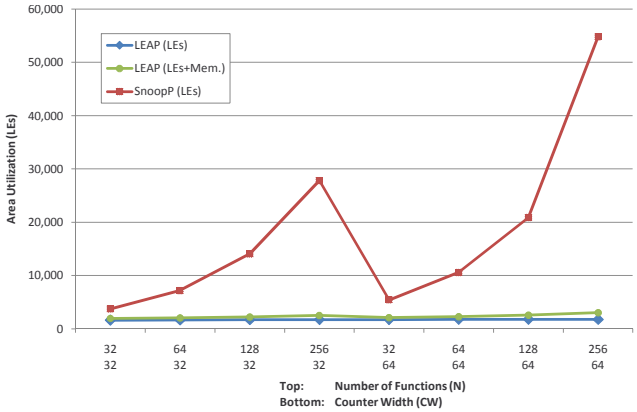

Fig. 6. Area overhead comparison of LEAP and SnoopP, both configured for cycle profiling.

figure, as is the sensitivity of SnoopP to the profiler parameter settings.

The columns labeled *Fmax* in Table V show the maximum post-routed frequency of each profiler in MHz. For LEAP, this ranges from 147 to 169 MHz; for SnoopP, this ranges from 79 to 113 MHz. The slow speed of SnoopP may present a problem if it is integrated into a soft processor with a clock rate that exceeds this maximum operating frequency.

The power overheads of LEAP and SnoopP appear in the *Power* columns of Table V. These power overhead numbers represent the geometric mean dynamic power comsumed by each profiler across all 13 benchmarks (the processor's power is not included). To isolate the profiler's power consumption, we performed a timing simulation of the entire system (including the processor) using ModelSim but only gathered switching activity for the profiler module. The activity data was then provided to Quartus PowerPlay [8]. Observe that, as with the area overhead, the power of SnoopP varies widely depending on the N and CW, ranging from 7.17-52.32 mW. LEAP's power consumption, on the other hand, is relatively flat, ranging from 5.85-6.67 mW. SnoopP's dynamic power ranges widely because the required number of comparators it uses doubles when N is doubled, which results in a 2× increase in dynamic power. When N is doubled in LEAP, only the Counter Storage RAM will double in size, however, the RAM memory blocks on the Cyclone II FPGA are 4K bits in size and have dedicated address decoding logic, making LEAP's power less sensitive to N. When comparing LEAP to SnoopP, it can be seen that LEAP consumes less power in all configurations. LEAP consumes at least 18% less power than SnoopP for $N = 32$ and $CW = 32$, but in the largest configuration it reduces the power consumption by 88%.

C. Energy Profiling

To assess the accuracy of energy profiling using LEAP, we performed timing simulations of the routed Cyclone II Tiger MIPS implementation executing each of the 13 benchmark programs. Simulation was performed using ModelSim with full routing delays, producing switching activity data (VCD files). The average dynamic power was computed for each benchmark using Quartus PowerPlay and the total execution time was then used to compute total energy. Total energy, computed in this way, represents the "golden" baseline with which we measure the accuracy of LEAP's energy profiling results. Function-level energy estimates produced by LEAP are summed and compared against the baseline. Note that LEAP

TABLE V

OVERHEAD COMPARISON OF LEAP AND SNOOPP, BOTH CONFIGURED FOR CYCLE PROFILING. NOTE: CW REFERS TO THE COUNTER WIDTH, AND N REFERS TO THE NUMBER OF FUNCTIONS PROFILED.

CW	N	Total LEs	Total Regs	Mem. Bits	Fmax (MHz)	Power (mW)	Total LEs	Total Regs	Fmax (MHz)	Power (mW)	Total LEs	Total Area	Dynamic Power
32	32	1,632	742	5,120	169.2	5.85	3,725	2,847	112.8	7.17	0.44	0.52	0.82
32	64	1,660	737	6,144	165.8	6.13	7,179	5,540	109.0	11.94	0.23	0.28	0.51
32	128	1,709	757	8,192	151.4	6.67	14,047	10,921	103.2	20.54	0.12	0.16	0.32
32	256	1,747	788	12,288	159.4	5.87	27,807	21,678	97.6	35.39	0.06	0.09	0.17
64	32	1,732	801	6,144	147.9	6.20	5,424	3,873	85.3	9.40	0.32	0.39	0.66
64	64	1,779	819	8,192	147.3	6.23	10,587	7,590	83.6	16.54	0.17	0.22	0.38
64	128	1,801	811	12,288	151.2	6.24	20,872	15,019	79.3	28.01	0.09	0.12	0.22
64	256	1,754	810	20,480	150.8	6.10	54,790	29,872	89.3	52.32	0.03	0.06	0.12

(a) LEAP **(b) SnoopP** **(c) $\frac{\text{LEAP}}{\text{SnoopP}}$**

TABLE VI

ENERGY PROFILING RESULTS WHEN ALL PIPELINE STALLS ARE CONSIDERED. NOTE: * INDICATES THE ENERGY (OR ASSOCIATED ERROR) WHEN CORRECTION FACTOR APPLIED.

Benchmark	Stalls	Cycles	Power (mW)	Energy (nJ)	Stall Factor	Energy* (nJ)	Total Time (ns)	Power (mW)	Energy (nJ)	Energy	Energy*
adpcm	37.3%	146,468	56.88	532,740.94	0.99	531,430.45	9,312,640	53.95	502,416.93	6.0%	5.8%
aes	46.0%	39,924	54.96	163,500.43	0.97	162,562.87	2,881,320	49.03	141,271.12	15.7%	15.1%
blowfish	12.3%	888,221	71.45	2,530,356.30	1.18	2,893,418.97	40,507,640	71.91	2,912,145.02	-13.1%	-0.6%
dfadd	44.7%	14,318	55.35	57,556.45	0.98	57,245.57	1,011,720	54.61	55,250.03	4.2%	3.6%
dfdiv	33.9%	52,898	57.26	183,025.38	1.00	183,036.18	3,226,360	55.51	179,095.24	2.2%	2.2%
dfmul	55.0%	4,969	53.20	23,716.31	0.96	23,550.46	431,200	49.04	21,146.05	12.2%	11.4%
dfsin	43.3%	2,180,140	55.12	8,526,386.48	0.98	8,483,392.37	153,907,120	54.80	8,434,818.19	1.1%	0.6%
gsm	42.2%	29,559	56.90	116,914.53	0.98	116,368.08	2,018,160	64.05	129,263.15	-9.6%	-10.0%
jpeg	34.6%	4,034,329	57.53	14,195,382.50	1.00	14,187,192.49	246,168,000	63.06	15,548,822.75	-8.7%	-8.8%
mips	21.1%	34,660	61.79	105,358.20	1.06	108,434.21	1,751,480	56.23	98,485.72	7.0%	10.1%
motion	33.0%	17,647	58.27	61,332.44	1.00	61,382.00	1,057,160	60.52	63,979.32	-4.1%	-4.1%
sha	9.5%	1,033,449	79.95	2,880,061.19	1.26	3,652,738.91	45,690,040	82.05	3,748,769.32	-23.2%	-2.6%
dhrystone	27.9%	23,394	59.69	76,520.37	1.02	77,149.23	1,280,440	58.68	75,136.22	1.8%	2.7%
									Abs. Average	8.38%	5.95%

(a) Energy Profiling Results **(b) Stall Correction Factor** **(c) Simulation "Golden" Results** **(d) Error**

instruction count and instruction type profiling (see Section III) is done in hardware on the DE2 board.

Table VI shows the energy profiling results. Part (a) shows the results of LEAP energy profiling, including the percentage of time spent in pipeline stalls, the total number of cycles required to execute the benchmark, the average dynamic power estimate, and the energy estimate calculated using the instruction-level power database. Part (b) shows a stall-based correction factor, described below, and the energy estimate (*Energy**) found using the correction factor. Part (c) shows the golden results, including total execution time in nanoseconds, average dynamic power, and total energy consumption. Part (d) shows the energy estimation error (with and without the correction factor), calculated using the standard estimation error formula: $error = \frac{estimate - actual}{actual}$.

The absolute value of the estimation error ranges from 1.1% for *dfsin*, to 23.2% for *sha*, with an average absolute error of 8.4%. Two benchmarks, *blowfish* and *sha*, have significantly underestimated energy consumptions. Interestingly, we saw that these two benchmarks exhibit much lower stall rates than any of the other benchmarks. On average, the pipeline stall rate across all benchmarks is 33.9%, however, *blowfish* and *sha* only stall for 12.3% and 9.5% of the time, respectively. We therefore speculated that stalls were the root of the high estimation error for these benchmarks. A small number of stalls implies that the processor is working for more of the time, leading to higher power dissipation.

Though we attempted to "factor out" stalls from the power values in the instruction-level power database (as described in Section III), it is difficult to do this precisely, as when one pipeline stage of Tiger MIPS stalls (for example, due to a data hazard), subsequent pipeline stages are allowed to proceed with execution. Such *partial* pipeline stalls are a source of inaccuracy in our power database. To compensate for this, we scale the estimated energy values by an empirically-derived correction factor that is based on a program's stall rate. The correction factor is defined to be: $factor = 1 + (\frac{Avg.Stall\% - Stall\%}{Stall\%}) \times \frac{1}{Stall\%}$, where $Avg.Stall\%$ is the average stall rate across all programs (33.9%) and $Stall\%$ is the fraction of time spent stalling in the program being profiled. Intuitively, this factor increases the power estimate when the stall rate of the program being profiled is below the average stall rate; likewise, the power estimate is reduced when the stall rate of the program is above the average stall rate.

Computed correction factors and scaled energy values are shown in columns 6 and 7 of Table VI, respectively The accuracy of the energy estimate is improved for 10 of the 13 benchmarks, with the average absolute error being reduced below 6%. Certainly, the impact of stalls on power profiling is a direction that warrants further investigation, especially when the embedded processor permits partial pipeline stalls.

D. LEAP-Based HW/SW Partitioning

As previously mentioned, LEAP has been developed as a key part of a HLS synthesis framework that targets a hybrid processor/accelerator system. This system consists of the Tiger MIPS processor as well as hardware accelerators that work in tandem with Tiger and communicate across the Altera Avalon bus interface. In this case, LEAP was used to identify the

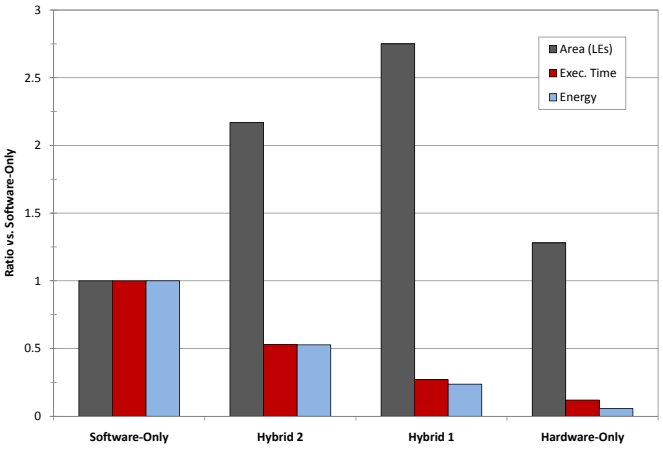

Fig. 7. Area, execution time, and energy comparison for hybrid systems in four configurations.

most compute-intensive functions in each of the benchmark programs, and thereby partition a program into software and hardware portions. We highlight a few results here for four experimental scenarios defined below. The interested reader is referred to [3] for complete results.

1) **Software Only**: this system consists of the base Tiger system, running pure software on the processor.
2) **Hardware Only**: this system consists of a pure-hardware (i.e. no processor) implementation of the benchmark, as created by the HLS framework.
3) **Hybrid 1**: this hybrid system accelerates the most compute-intensive software function (and descendants), identified using LEAP.
4) **Hybrid 2**: this hybrid system accelerates the second-most compute-intensive software function (and descendants), identified using LEAP.

Fig. 7 summarizes the area (# of LEs), execution time and energy results, normalized to the software only case. The results are geometric mean values taken across all 13 benchmarks. Circuit-by-circuit results are excluded due to page limitations. Moving from left-to-right in the figure corresponds to more computations being realized in hardware versus software. Looking first at the number of LEs, observe that the hybrid designs require significantly more LEs than either software-only or hardware-only. The reason for this is that the hybrid scenarios contain *both* the MIPS processor in addition to hardware accelerators.

Turning to the execution time and energy results, we see that both metrics are reduced considerably as computations are migrated into hardware. LEAP is effective in choosing the most compute-intensive functions that would benefit from a hardware implementation. Choosing the second-most compute-intensive function (Hybrid 2) does not offer the same energy benefits as choosing the most compute-intensive function (Hybrid 1). This gives us confidence in the fidelity of LEAP's ranking of functions.

V. Conclusions and Future Work

In this paper, we introduced LEAP, a non-intrusive hardware profiler that is both area- and power-efficient. LEAP is easily extensible to several forms of cycle-accurate profiling, including instruction count, stalls, and total cycle count. LEAP uses perfect hashing in hardware to avoid the use of a large number of address bus comparators, leading to up to 18× less area overhead than a recently published hardware profiler, and up to 8.6× less energy overhead. We extended LEAP to perform energy profiling in conjunction with an instruction-level power database, and achieved accuracy within 6% absolute error, on average, on a suite of 13 C-language benchmarks. LEAP can also be applied to the task of hardware/software partitioning for the automated design of processor/accerator systems-on-chip.

We believe hardware profiling of embedded processors to be a rich area for future research. We plan to extend LEAP to track additional run-time program behavior, including branch outcomes, semi-static variables, and memory access patterns. Such data is useful in guiding the synthesis of software to hardware. For example, branch outcomes can direct loop unrolling, and memory access pattern data can be used to drive memory architecture synthesis. Other future directions include adding support for profiling recursion, as well as incorporating techniques to enable the use of smaller data counters, such as counter saturation detection and shifting [13].

References

[1] Gnu gprof. http://sourceware.org/binutils/docs/gprof/index.html.
[2] *ModelSim - Advanced Simulation and Debugging*.
[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 33–36, 2011.
[4] Mark Aldham. Low-Cost Hardware Profiling of Run-Time and Energy in FPGA Soft Processors. Master's Thesis, ECE Department, University of Toronto, 2011.
[5] Altera, Corp. *http://www.altera.com*.
[6] Altera, Corp., San Jose, CA. *Cyclone II Device Family Data Sheet*, 2011.
[7] Altera, Corp., San Jose, CA. *Nios II Processor Reference Handbook*, 2011.
[8] Altera, Corp., San Jose, CA. *PowerPlay Power Analysis*, 2011.
[9] Altera, Corp., San Jose, CA. *Quartus II Handbook Version 10.0*, 2011.
[10] Peter Clarke. Arm clone taken offline, talks between its designer and arm. 11 2001.
[11] J. Cong and Y. Zou. FPGA-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. Reconfigurable Technol. Syst.*, 2(3):1–29, 2009.
[12] M. Finc and A. Zemva. A systematic approach to profiling for hardware/software partitioning. *Computers & Electrical Engineering*, 31(2):93 – 111, 2005.
[13] A. Gordon-Ross and F. Vahid. Frequent loop detection using efficient non-intrusive on-chip hardware. *IEEE Trans. Comput.*, 54:1203–1215, October 2005.
[14] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.
[15] http://www.llvm.org. *The LLVM Compiler Infrastructure Project*, 2011.
[16] B. Jenkins. Perfect hashing. http://burtleburtle.net/bob/hash/perfect.html, 2011.
[17] J. Luu, K. Redmond, W. Lo, P. Chow, L. Lilge, and J. Rose. FPGA-based monte carlo computation of light absorption for photodynamic cancer therapy. In *IEEE FCCM*, pages 157–164, 2009.
[18] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM PLDI*, pages 190–200, 2005.
[19] MIPS Technologies, Sunnyvale, CA. *MIPS Architecture For Programmers*, 2010.
[20] S. Moore and G. Chadwick. The Tiger "MIPS" processor. http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html, 2011.
[21] J. Ou and V.K. Prasanna. Rapid energy estimation of computations on FPGA based soft processors. In *IEEE Int'l SOC Conference*, pages 285 – 288, September 2004.
[22] G. Qu, N. Kawabe, K. Usami, and M. Potkonjak. Function-level power estimation methodology for microprocessors. In *IEEE/ACM DAC*, pages 810–813, 2000.
[23] E. Saad, M. Awadalla, and K. El-Deen. FPGA-based real-time software profiler for embedded systems. *Int'l Journal of Computers, Systems and Signals*, 9(2), 2008.
[24] L. Shannon and P. Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *ACM FPGA* pages 190–199, 2004.
[25] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Trans. on VLSI*, 2(4):437–445, December 1994.
[26] J. Tong and M. Khalid. Profiling tools for FPGA-based embedded systems: Survey and quantitative comparison. *Journal of Computers*, 3(6):1–14, 2008.
[27] L. Vaughan-Adams. ARM investigates Swedish university over 'clone' claim. *The Motley Fool Investment Page*, March 2001.
[28] V. Aggarwal W. Marx. FPGAs Are Everywhere – In Design, Test & Control. *RTC Magazine*, April 2008.
[29] R. P. Weicker. Dhrystone benchmark: rationale for version 2 and measurement rules. *SIGPLAN Not.*, 23:49–62, August 1988.
[30] Xilinx, San Jose, CA. *MicroBlaze Soft Processor Core*, 2011.