A Hierarchical Profiler of Intermediate Representation Code based on LLVM

Efstratios Tiganourias*†, Michail Mavropoulos†, Georgios Keramidas*†, Vasilios Kelefouras†,
Christos P. Antonopoulos†, and Nikolaos Voros†
*Aristotle University of Thessaloniki, Greece
{*etiganou@csd.auth.gr, gkeramidas@csd.auth.gr*}
†University of Peloponnese, Greece
{*s.tiganourias, m.mavropoulos, g.keramidas, v.kelefouras, ch.antonop, voros*}*@esda-lab.gr*

***Abstract*—Profiling based techniques have gained much attention on computer architecture and software analysis communities. The target is to rely on one or more profiling tools in order to identify specific code pieces of interest e.g., code pieces that slowdown a given application. The extracted code pieces can be further modified and optimized. In general, the profiling tools can be classified as deterministic, statistical-based, or rely on hardware performance counters. A common characteristic of the available profiling tools is typically based on analyzing or even manipulating (in case of binary instrumentation tools) machine-level code. This approach come with two main drawbacks.**

First, a lot of information (even GBytes of data) needs to be gathered, stored, post-processed, and visualized. Second, the performed analysis of the gathered data is platform-specific and it is not straightforward to categorize the given application-s/program phases/kernels into distinct categories that have the same or almost the same behavior (e.g., the same percentage of computational vs. control instructions). The latter stems from the fact even small changes in the source code of the applications might lead to significantly different machine code implementations. Therefore, even two specific program kernels exhibit the same behavior (e.g., they have the same number of instructions, but with a different ordering), it is very difficult for a machine-code level profiling tool to assess their similarity, simply because the generated machine level code might have significant differences resulting in many missing opportunities for the available profiling tools. To address this issue, in this paper, we present a new profiling tool that is able to operate on the machine independent intermediate representation (IR) level. The profiler (still in development phase) relies on the LLVM API and it is able to hierarchically (at various levels of the call stack) and recursively parse the IR code and extract various useful statistics. We showcase the practicality of our profiler by analyzing a subset of the PolyBench benchmarks assuming (as pointed out by a recent study) that there is a strong correlation of LLVM IR code.

I. Introduction

Profiling tools are computer aided design (CAD) tools that can analyze a given application and extract various useful statistics. Typically, the various profiling tools are used for three main reasons: i) to identify the bottlenecks of an application, ii) to identify specific problematic cases in the source code of an application (e.g., memory leaks), iii) to understand the run-time requirements of an application (e.g., heap size, cache size etc.), and iv) to help a third user to better understand a software program (e.g., by building and presenting the code-flow graph or the call graph of an application). The later category usually is accompanied by a second tool to visualize the results.

Generally speaking, the profiling tools can be split into two main categories: software-based and hardware-based. Software based tools can be further categorized into deterministic and statistical profilers. Deterministic profilers create execution traces of the application code e.g., function calls, functions returns, interrupt code etc. and for each distinct point a timestamp is assigned. The main drawback of these kind of profilers is that a lot of data must be recorded resulting into a considerable slowdown in application performance.

Statistical profilers raise the level of abstraction by performing a sampling-based approach. Instead of monitoring the events of interest continuously, the application is interrupted periodically and in each interrupt, specific statistics are gathered. The periods between two interrupts can be predefined, random or dynamically extracted (i.e., adjust the sampling rate dynamically). The latter approach typically leads to highly accurate statistics with low overheads.

In general, a plethora of open-source and proprietary profilers exist. These tools can offer CPU and memory profiling e.g., to extract the time spent in every function of the application, to reveal CPU hotspots, memory bottlenecks or even to scan the source code for memory leaks.

Probably the most widely used profiling tool for user space code is GNU gprof [9]. gprof actually utilizes the debug capabilities of gcc and is able to report function relationships (i.e., caller-callee). These relationships can be annotated, thus it is possible to identify "hot" functions. The sampling is performed by Operating System (OS) interrupts, thus significant overheads is added. Another popular tool is Valgrind [10]. Valgrind is actually an instrumentation framework and it comes with a set of various tools: memory error detector, cache and branch profiler, call graph generator, heap profiler, and thread error detector. Both gprof and valgrind are not considered accurate and robust in analyzing multi-threaded applications. On the contrary, the gperf tool [11] (developed by Google) can help the designers to analyze and enhance the performance of multi-threaded applications.

Moving now to the category of the hardware based profiling tools, these tools utilize the hardware performance counters [2][6] that are available on almost all modern processors

e.g., Intel or ARM processors. These hardware counters are dedicated in the run-time monitoring of the execution of an application. Although different processors (even from the same company) support a different set of performance counters, typically various hardware events can be exposed [2] [6]. These events include memory accesses, branch misprediction, memory fill times, register spill events etc. Hardware based profiling typically introduces minor performance overheads [8], since no instrumentation is needed. In order to read these counters, external tools need to be installed. These tools are perf [11], oprofile [14], and Performance Advance Programming Interface (PAPI) [13].

A common parameters of all the above tools is that the gather information is either too high level (i.e., at the Operating System level) or platform-specific. The latter issue renders the work of identifying and classifying applications/program phases/kernels with common characteristic a challenging task. For example, inspecting the source code of an application is difficult to tell if two or more applications exhibit the same percentage of computational vs. control instructions.

The latter stems from the fact even small changes in the source code of the applications might lead to significantly different machine code implementations. Therefore, even two specific program kernels exhibit the same behavior (e.g., they have the same number of instructions, but with a different ordering), it is very difficult for a machine-code level profiling tool to assess their similarity, simply because the generated machine level code might have significant differences resulting in many missing opportunities for the available profiling tools.

To this end, in this paper, we present a new profiling tool that is able to operate on the machine independent intermediate representation (IR) level. The profiler (still in development phase) relies on the LLVM API and it is able to hierarchically (at various levels of the call stack) and recursively parse the IR code and extract various code-level statistics. We showcase the practicality of our profiler by analyzing a subset of the PolyBench benchmarks assuming (as pointed out by a recent study [7]) that there is a strong correlation of LLVM IR code and the dynamic power figures of a given application.

Structure of this paper. Section 2 provides additional background information for modeling and/or estimating the power consumption at processor level. Section 3 presents our developed profiler and Section 4 provides the results when the proposed profiler is used to analyze the PolyBench benchmarks. Finally, Section 5 concludes the paper.

II. Processor power estimation

Knowledge of processor run-time power consumption is essential for efficiently managing energy-saving techniques and maximizing performance without exceeding the thermal and power limits of a target device [1]. Towards this approach many techniques have been proposed targeting to estimate the static and dynamic power consumption of a given processor. The majority of previous works relies on Performance Monitoring Counters (PMCs). PMCs are internal CPU registers that count (micro-)architectural events and they can be used to estimate runtime power consumption by building empirical power models. The idea is to end up with regression based power models for CPUs using PMCs as inputs.

However, in order to build and train these models the actual power consumption of the processor must be recorded. This can be done either by an external measurement device or by utilizing specific power sensors (introduced in the latest processors). Bellosa [2] presented one of the first works to use PMCs for power monitoring purposes. Isci and Martonsi [6] later presented a technique for combining PMC data with power consumption to provide run-time power estimations for a P4 CPU.

A problem in building robust power models is to select the right set of PMC events [5][8]. An ideal set should not include PMCs that are correlated to each other to avoid redundancy [3]. The problem becomes even more challenging if we consider: i) almost each processor (even from the same vendor) has a different set of performance counters and ii) the diversity of the various workloads precludes the building of power models that can be used across a wide range of applications and platforms. One potential way to address this issue is to raise the level of abstraction and try to categorize the workloads according to a set of platform-agnostic metrics. As noted, in this work we select to classify the workloads based on the mix of the machine independent intermediate representation (IR) code. However, to this end, we developed a profiler that relies on the LLVM API and it is able to hierarchically and recursively parse the IR code and extract various IR code-level statistics. The internal operation of this profiler is presented in the next session.

III. The LLVM IR Profiler

The LLVM toolchain [12] already includes a number of passes that can operate at various levels. These passes are divided into two main categories: analysis passes and transformation passes. Analysis passes extract information from the source code of the input application, while transformation passes apply specific code transformations. In this work, we develop an analysis pass that is built on top of existing passes. The new pass is modular and it can be easily integrated in the LLVM codebase, since it relies on LLVM API. As a first step, the LLVM pass takes as input an LLVM IR code and outputs the call graph. The call graph information is a useful representation of the input IR code (thus, the input application) in order to understand the basic structure and organization of the code. However, it is not suitable for more complex purposes such as power consumption analysis, security analysis etc. To accomplish the latter goals, our proposed LLVM pass is extended to record and classify the various IR opcodes of the input source codes. In addition, the proposed LLVM pass includes a classification step that is able to classify the IR opcodes into specific categories. These categories are given as inputs to the classifier. As part of this work, we assume that the IR opcodes that belong to the same category exhibit the same power profile. More details about this classification can be found in Section IV.

A high level design of the developed profiler is shown in Figure 1. As the figure illustrates the LLVM-pass takes as input LLVM IR code and a mapping between IR instructions and their power figure, and outputs: i) the different IR instructions organized in categories, ii) the callgraph (thus, the IR level statistics can be given at a hierarchical manner wrt. the functions of the program), and iii) a power analysis.

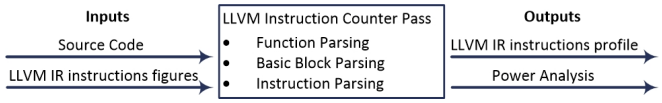

Fig. 1: Profiler overview

A. Profiler Design

The LLVM API is a well organized API that allows to add additional LLVM transformations with new features. In the context of this work, we extend the Instruction class of LLVM. We insert new hooks and features in order to classify the IR opcodes of the input application into different IR categories. More specifically, the LLVM toolchain provides a class called *Function* that represents a function in the LLVM IR. The class *Function* provides an iterator for each *Basicblock* classes and in each basic block an iterator for its Instruction classes. In Figure 2, these steps are annotated as (4) and (5). Therefore, we have at least three abstraction layers in our source code to begin with.

Moreover, all LLVM passes are subclasses of the LLVM *Pass* class and their functionality comes from overriding the virtual methods inherited from the *Pass* class. For our work, we override the function *runOnFunction* (step 1 in Figure 2). The pass iterates through the various functions in the IR, the basic blocks of each function, and finally the instructions of each basic block. During this process, the profiler holds a record for every visited instruction and updates a data structure with their total numbers of occurrences (step 6 in Figure 2). In addition, it operates recursively when the instruction "call" is visited (step 7) in order to create a call graph for the function and accordingly count all the instructions in the created call tree. Finally, the classification of each instruction, based on its power profile, is performed. The output of the profiler are a call graph for each function, a analysis of all its opcodes, and a power figure analysis (as a result of the classification of the LLVM IR instruction types).

B. Working Example

For validation purposes we present an example of an OpenCL kernel (Listing 1), on which we applied our LLVM pass.

```
__kernel void doitgen_kernel2(int nr, int nq, int
    np, __global DATA_TYPE *A, __global DATA_TYPE *
     C4, __global DATA_TYPE *sum, int r)
{
     int p = get_global_id(0);
     int q = get_global_id(1);

     if ((p < np) && (q < nq))
     {
          A[r * (nq * np) + q * np + p] = sum[r * (nq
     * np) + q * np + p];
     }
}
```

Listing 1: Source code of doitgen kernel 2

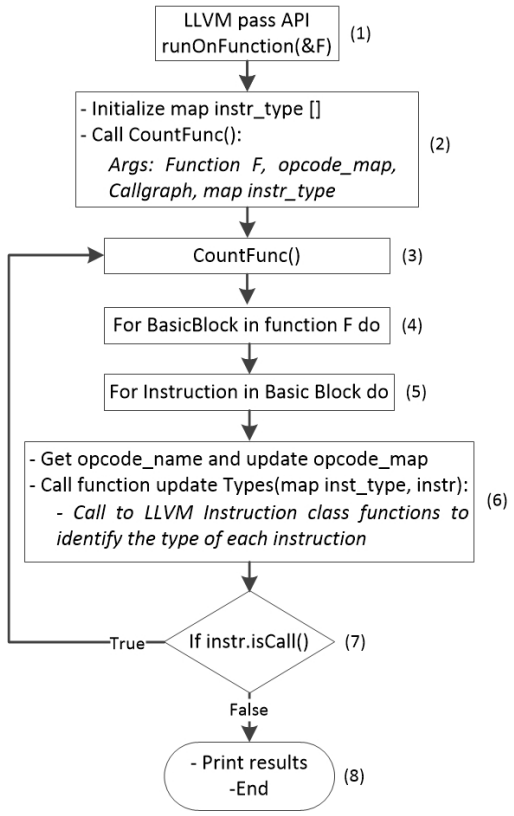

Fig. 2: Profiler block diagram

This OpenCL kernel is part of the doitgen benchmark from the Polybench Suite [4]. This kernel is used for Multi-resolution Analysis or MADNESS and is typically is used in linear algebra for solving integral and differential equations of many dimensions (more details about this kernel is out of the context of this work). This source code corresponds to the LLVM IR code (shown in Listing 2) as it is generated by the clang compiler.

```
@doitgen_kernel2(i32 %nr, i32 %nq, i32 %np, float*
    nocapture
 %A, float* nocapture readnone %C4, float* nocapture
     readonly %sum,
 i32 %r)  {
 entry:
   %call = tail call i64 @_Z13get_global_idj(i32 0)
    #3
   %conv = trunc i64 %call to i32
   %call1 = tail call i64 @_Z13get_global_idj(i32 1)
     #3
   %conv2 = trunc i64 %call1 to i32
   %cmp = icmp slt i32 %conv, %np
   %cmp4 = icmp slt i32 %conv2, %nq
   %or.cond = and i1 %cmp, %cmp4
   br i1 %or.cond, label %if.then, label %if.end

 if.then:                                          ;
     preds = %entry
```

```
15    %mul6 = mul i32 %r, %nq
16    %reass.add = add i32 %mul6, %conv2
17    %reass.mul = mul i32 %reass.add, %np
18    %add8 = add i32 %reass.mul, %conv
19    %idxprom = sext i32 %add8 to i64
20    %arrayidx = getelementptr inbounds float, float*
       %sum, i64 %idxprom
21    %0 = load float, float* %arrayidx, align 4, !tbaa
       !8
22    %arrayidx15 = getelementptr inbounds float, float
       * %A, i64 %idxprom
23    store float %0, float* %arrayidx15, align 4, !
       tbaa !8
24    br label %if.end
25
26  if.end:                                           ;
        preds = %if.then, %entry
27    ret void
```

Listing 2: LLVM IR code of doitgen kernel 2

As we can see in lines 5 and 7 in Listing 2, there are two call instructions, that correspond to the get_global_id() function calls in the source code. In addition, the if statement in source code (line 6 in Listing 1) is translated to the IR opcode depicted in lines 9, 10, and 11 in Listing 2. Line 12 is the branch instruction coming from the *if...then* label or the *if...end* label. The body of the if statement corresponds to the lines 15-24 inside the if...then label.

```
1  Function: doitgen_kernel2
2  add :: 2
3  and :: 1
4  br :: 2
5  call :: 2
6  getelementptr :: 2
7  icmp :: 2
8  load :: 1
9  mul :: 2
10 ret :: 1
11 sext :: 1
12 store :: 1
13 trunc :: 2
14
15 Power type profiling
16 NoType -> 5
17 type1 -> 3
18 type2 -> 2
19 type3 -> 4
20 type4 -> 5
21
22 Callgraph for doitgen_kernel2
23 Null
```

Listing 3: Output of the profiler

The Listing 3 illustrates the output of the proposed profiler. As we can see, the top part of Listing 3 enumerates the IR level statistics for the two functions of the studied kernel (lines 1-5 and lines 6-13). The bottom part of Listing 3 (lines 15-20) depicts the IR level instruction classification statistics. As noted more details about the classification process will be given in Section IV.

IV. Results

A. Approach

To generate the LLVM IR bitcode from the OpenCL kernels, we use the clang compiler version 12.0.0. We also use the *opt* tool from the LLVM toolchain. To perform the IR instruction classification, we accordingly extent the *Instruction* class of the LLVM codebase. Finally, we apply our profiler to the Polybench-ACC [4] benchmark suite and more particularly in the linear algebra OpenCL applications. Each application consists of 1 or 2 OpenCL kernels (annotated as K1 or K2 hereafter).

As noted, the goal of this work is to classify different OpenCL kernels wrt. the number of IR-level instructions. Therefore, the next step was to classify each IR opcode to five distinct categories. Obviously, each category contains opcodes with similar or almost similar power profiles. As part of this work, the following categories were used (we plan to extent this categorization and further validate our approach as part of our future work).

The **type1** group includes all the simple ALU operations e.g., additions, subtractions, and bitwise operations. **type2** group contains the long-latency, thus more power consuming, ALU operations like multiplications and divisions instructions. The **type3** group consists of the load-store memory operations and the **type4** includes all the control instructions e.g., branches, function calls, function returns etc. Finally, there is a last category (**NoType**) with all the remaining LLVM IR instructions.

B. Kernel Characterization

The section presents our profiler-based characterization of the following OpenCL applications from the Polybench-ACC [4] benchmark suite: 2mm, 3mm, atax, bicg, doitgen, gemm, gemver, mvt and syr2k. In total, 17 OpenCL kernels were used. Our goal, as explained, was to create groups of similar kernels depending on their IR instruction mix, thus groups with similar power behavior. Our classification algorithm works as follows: two kernels are grouped together if the difference in the same type instructions is lower than a predefined threshold. In this work we set two threshold values: 10% and 20%. Table 1 contains the extracted kernel groups. As we can see from Table 1, when the first threshold value is enforced, we end up with 12 groups, while only eight groups are extracted for the second threshold value.

Finally, Figure 3 presents the whole range of the profiler outputs for each OpenCL application and kernel (shown in the horizontal axis). The y-axis shows the absolute values of the number of instructions of each instruction type. It is obvious from Figure 3 that specific kernels (belonging to the same or to different applications) exhibit the same behavior (equal or almost equal number of instructions per instruction type) e.g., the 2mm k1 and gemm k1 kernels, while other kernels show different instruction statistics (e.g., doitgen k2 and 2mm k2), thus different power profiles.

V. Conclusions

In this work, we implement a new profiler[1], based on the LLVM framework [12]. The profiler extends the Call

[1]The profiler will be released as an open-source tool

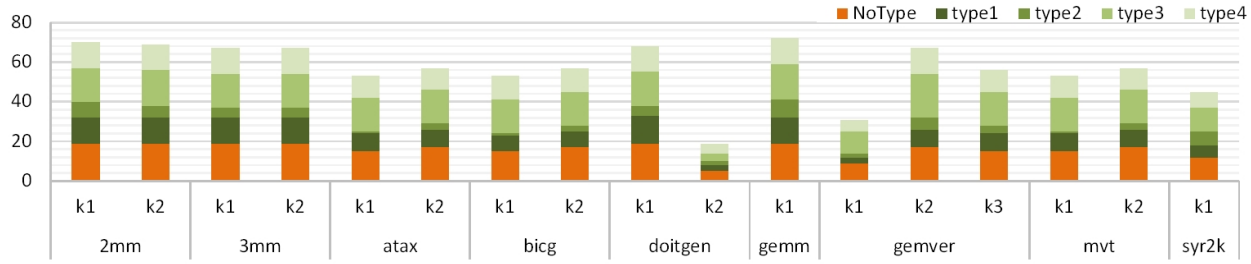

Fig. 3: LLVM IR instructions classification

kernel clustering into groups	10%	20%
group 1	2mm k1, gemm k1	2mm k1, gemm k1
group 2	2mm k2	2mm k2, 3mm k1, 3mm k2, doitgen k1, gemver k3
group 3	3mm k1, 3mm k2, doitgen k1	atax k1, bicg k1, mvt k1
group 4	atax k1, mvt k1	atax k2, bicg k2, mvt k2
group 5	atax k2, mvt k2	doitgen k1
group 6	bicg k1	gemver k1
group 7	bicg k2	gemver k2
group 8	doitgen k2	syr2k k1
group 9	gemver k1	-
group 10	gemver k2	-
group 11	gemver k3	-
group 12	syr2k k1	-

TABLE I: Kernels classification

Graph pass functionality of LLVM and also manages to create clusters of IR instructions based on the power characteristics of each IR instruction. We evaluate the proposed profiler over a subset of Polybench OpenCL kernels (17 in total) and we conclude that the studied benchmarks can be classified into into 12 groups when we set a similarity factor equal to 90% and into eight groups when the factor is equal to 80%.

VI. Acknowledgement

This work has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No 871738 - CPSoSaware: Cross-layer cognitive optimization tools amp; methods for the lifecycle support of dependable CPSoS.

References

[1] C. D. Bagwell, E. Jovanov, J. H. Kulicl, "A Dynamic Power Profiling of Embedded Computer Systems," Proc. of the Thirty-Fourth Southeastern Symposioum in System Therory, 2002.

[2] F. Bellosa, "The benefits of event: Driven energy accounting in power-sensitive systems," In Proc. of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, 2000.

[3] W. L. Bircher, M. Valluri, J. Law, and L. K. John, "Runtime identification of microprocessor energy saving opportunities," In Proc. of Int. Symp. on Low Power Electronics and Design, 2005.

[4] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes," In Proc. of Innovative Parallel Computing, 2012.

[5] G. D. Costa and H. Hlavacs, "Methodology of measurement for energy consumption of applications," Int. Conference on Grid Computing, 2010.

[6] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," In Proc. of Int. Symp. on Microarchitecture, 2003.

[7] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse and K. Eder, "Static analysis of energy consumption for LLVM IR programms," Proc. of Int. Workshop on Software and Compilers for Embedded Systems, 2015

[8] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," SIGARCH Comput. Archit. News, 2009.

[9] Gprof the GNU profiler https://sourceware.org/binutils/docs/gprof/

[10] Valgrind's Tool Suite https://www.valgrind.org/info/tools.html

[11] Gnu gperf https://www.gnu.org/software/gperf/

[12] LLVM Project https://llvm.org/

[13] Performance Application Programming Interface https://icl.utk.edu/papi/

[14] OProfile - A System Profiler for Linux https://oprofile.sourceforge.io/news/