AccProf:

AccProf: Increasing the Accuracy of Embedded Application Profiling using FPGAs

Emmanouil Skordalakis, Andrew Attwood, John Goodacre, and Mikel Luján

University of Manchester, M13 9PL, UK
{emmanouil.skordalakis,andrew.attwood,john.goodacre,mikel.lujan}@manchester.ac.uk

Abstract. Accurate software profiling is an essential step in the development of embedded systems. The accuracy of profiling data collected is critically important for embedded systems that operate under fixed timing constraints, which if not met, could lead to system failure. Existing profiling solutions targeting embedded systems introduce an overhead to the running application that distorts the collected profiling data. This paper proposes AccProf for System on Chips with integrated FPGAs. AccProf is a FPGA-assisted profiling framework combining compiler extensions and bespoke hardware. AccProf is composed of (1) an LLVM pass which inserts lightweight instrumentation into the application binary running on general purpose processors , and (2) FPGA-based hardware capable of performing offloaded profiling. Offloading part of the profiling task, and supporting data-structures to the FPGA, reduces pollution of the collected profiling data leading to higher accuracy. This paper addresses on control graph profiling and evaluates AccProf on a range of benchmarks ported to SeL4 microkernel running on the AMD Zynq MpSoC. We measure performance metrics of these benchmarks across a range of processor statistics including cycles, instruction and data cache misses. We show that the impact for certain metrics is reduced for up to 5× when compared against an equivalent software-based framework.

Keywords: Embedded systems · Profiling framework · Performance counters · SeL4.

1 Introduction

Applications targeting embedded systems have their design and performance improved through iterative re-engineering aided by profiling tools. Profiling is an essential step, providing programmers with fine grained low-level information, e.g. cache misses. Software-based profiling tools typically generate call-graphs of instances of functions [10] [11], with each vertex in the graph annotated with a wider set of programmer-selected information. [7] [9] [16]. Additionally other hardware-based approaches report real-time execution or clock cycles, monitor context switches on multi-task applications or report loop statistics [20] [4] [17] [19] [20]. The problem with existing profiling solutions is that for software-based, instrumentation imposes significant overhead generating additional accesses to shared hardware resources, such as TLBs and Caches. While hardware

approaches do not have that overhead and produce in general more accurate results, they are built as processor extensions and are not widespread.

Given the growing popularity of System-on-Chip (SoC) with integrated FPGAs (AMD Zynq and Intel Aria SoC products) for embedded systems, this paper introduces AccProf, a flexible hardware assisted profiling framework. AccProf uses the FPGA to reduce the profiling intrusion and perturbation on the running application. AccProf identifies each function call and uses standard performance counters to gather events, such as processor cycles, Data and Instruction Cache statics. It also generates the call-graph and associates to each node (function call) the collected event counts. AccProf offloads both call graph generation and profiling data structure management to a flexible FPGA IP block utilising local and dedicated memory (BRAM memory in the FPGA). As a result, the instructions required for storing the profiling information as a call-graph along with the data itself, are not inserted as extra instrumentation into the application. Thus AccProf can produce more accurate profiling data by introducing reduced perturbation of the instrumented application. AccProf uses a custom instrumentation engine leveraging the LLVM plugin framework [14]. The instrumentation engine inserts into the profiled application additional instructions that gather user selected statistics from the hardware performance counters that are sent to the FPGA IP block to be processed. For out initial experimentation we integrated AccProf with the popular SeL4 [12] operating system.

SeL4 is a verified microkernel, but at the current time has limited profiling facilities. To enable the evaluation of FPGA assisted profiling, we also implemented AccProf as a software-only profiling framework herafter refer to as AccProf-soft. Several benchmarks with different configurations are used to evaluate both profiling tools. We show that AccProf can reduce application interference up to 5× for Level 1 Data/Instruction Cache refills. This is achieved while not introducing any additional noise or overhead on other information collected. When we consider applications with very simple call graphs which do not benefit AccProf, we do not impose any additional overhead compared to AccProf-soft.

2 Related Work

AccProf uses an instrumentation engine based on the LLVM framework, and profiles user-specified functions of an application running on embedded processors. It collects low-level information of those functions using the hardware performance counters provided by the underlying architecture. It stores information for every function in the form of a call-graph. AccProf's key characteristic is that the processing and storing of collected information is moved from the instrumentation engine, to the FPGA. The net result is fewer instrumentation instructions are added to the profiled application resulting in reduced pollution of the collected statistics.

The majority of profiling tools generate call graphs of functions or basic blocks. GProf is a classic profiling tool that collects execution time of every function running on an application and outputs that time in seconds along with

caller/callee relations of the profiled functions [10]. Hill *et al.* describe a call graph profiling tool that measures imbalances on computation and communication costs of parallel programmes, and visualises them in a simple manner as if they were sequential [11].

Other profiling solutions use the hardware performance counters in conjunction with other techniques, mainly to identify bottlenecks in parallel systems. PerfExpert detects performance bottlenecks in procedures and loops and provides suggestions for performance improvements [7]. Diamond *et al.* examine singlecore performance information, such as cache misses, and demonstrate how they are misleading for multicore systems. They study performance bottlenecks in multicore systems and introduce loop microfission that performs optimisations on these bottlenecks [9]. Paradyn measures performance on large scale parallel programmes by using dynamic instrumentation [16]. All the aforementioned software-based approaches are fully-intrusive, resulting in the entirety of the profiling algorithm being injected into the application, in turn polluting the measurements of the hardware counters. In contrast, a big part of AccProf's algorithm is offloaded onto the integrated FPGA, reducing both the number of profiling instructions and the impact of polution.

Previous work on profiling tools using FPGAs target soft processing cores. Gordon-Ross *et al.* proposed a detector that is able to locate critical regions of frequent loops for dynamic optimizations and report execution time [20]. LEAP attains real-time cycle profiles of an FPGA soft-processor and uses a hashing technique together with hardware counters for accurate results [4]. DAProf monitors an applications backward branches, function calls and returns to provide accurate loop execution statistics [17]. Later it was extended to monitor context switches for multi-tasked applications [19]. SnoopP obtains clock cycle accurate profiles of software programmes running on a soft-processor instantiated on an FPGA [20]. Even though these profilers are implemented solely in hardware and thus are non-intrusive, meaning they do not impose any instrumentation overhead, they require large and complex circuitry on the hardware occupying a lot of resources and they can only work with specific soft-processors, while AccProf is compatible with a variety of general-purpose processors such as x86, Arm and RISC-V by taking advantage of the API abstractions of SeL4 that manage the PMU counters.

Other notable references targetting software counters could also be supported in AccProf. TEE-PERF is a platform independent performance measurement tool for trusted execution environments [5]. That does not rely on hardware counters to count program execution, rather it uses a software counter and outputs the results of the profiling application in the form of a flame graph. Beischl *et al.* show their framework for profiling data-flow systems at higher abstraction levels effectively linking low level results of traditional profiles to higher level abstractions of the application [6].

3 AccProf - Design and Implementation

AccProf is comprised of three main modules that collaborate to enable the profiling of an application's user specified functions, delivering a fine grained function or basic block based call-graph combined with information collected from the performance counters:

- A compiler source-instrumentation engine based on the LLVM plugin framework [14] that makes use of the hardware performance counters of the underlying architecture.
- A High-Level Synthesis (HLS) design on the on-chip FPGA that receives data from the instrumentation engine regarding called functions and their collected profiling information, to generate a call-graph.
- The SeL4 microkernel as an operating system to run its applications [12]. AccProf uses two key aspects of SeL4. First, The root task which is the first thread instantiated by SeL4 after boot and is able to map the HLS address to the virtual address space of the instrumented applications. Second, the software abstractions of SeL4 for accessing performance counters which support x86, ARM and RISC-V architectures. An overview of AccProf design is shown in Fig. 1.

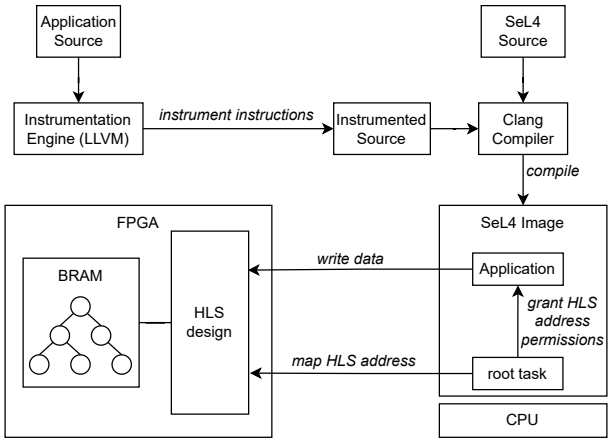


Fig. 1: An Overview of the AccProf profiling framework. The application source is first parsed by the Instrumentation engine to produce an instrumented source. This in combination with the SeL4 OS code are compiled with Clang and the output Image is loaded onto the CPU. The root task of the Image maps the HLS address and grants permissions of that address to the instrumented application. Finally the application writes profiling data to the HLS where the latter stores them to a BRAM in the FPGA in the form of a call-graph.

3.1 Instrumentation engine based on LLVM

AccProf's instrumentation engine is based on the LLVM framework. The application is instrumented at compile-time with instructions that access hardware performance counters and communicate with the embedded system's FPGA.

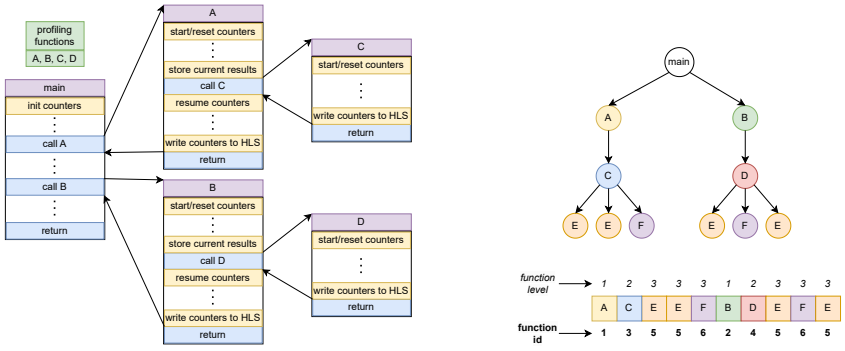
clang/LLVM [13] is a mature compiler infrastructure. The Clang front end compiles C/C++ source code into a Intermediate Representation (LLVM-IR), followed by the LLVM back end that transforms LLVM-IR into the specified architecture binary. IR code has the benefit of being platform agnostic, so it can be compiled to any supported architecture, and one can instrument additional code to an IR source, using the LLVM compilation framework for programme analysis and transformation [14].

AccProf uses the LLVM compilation framework as an instrumentation engine. It injects instructions to the application source code during compilation by using an LLVM pass. The pass retrieves a list of functions supplied by the developer that need to be profiled, and up to 6 events that need to be counted for every function. First it adds a basic block in the beginning of main that instantiates the counters and sets them to specific events. It then traverses each function and injects IR code in the beginning and in the end of each function. The pass also traverses every part of the code and searches for calls to the profiled functions, where it inserts an additional basic block before and after each call instruction.

An example of injected code by the AccProf instrumentation engine in an application is shown in Fig. 2a. In the prologue of main, counters are initialised with the specified events. In the beginning of each profiling function the corresponding performance counters are started/reset, and in the end the results are written to a virtual address pointing to the FPGA of the MpSoC. The reason we also inject instructions before and after the call instructions of these functions, is because if a profiling function (caller) is called by another profiling function (callee), the callee will reset the counters at its prologue. In that case the caller needs to store current event counts, and reset the counters once the callee has returned to continue counting its own events.

3.2 FPGA Hardware - HLS driven

AccProf is a less intrusive profiling framework, since part of its computation is implemented as a Hardware IP block on the FPGA. To increase the flexibility for the software developer, we use High Level Synthesis (HLS) to generate the FPGA IP block of AccProf. HLS abstracts hardware design details by providing the capability to synthesise hardware using C/C++, rather than hardware description languages [8]. That abstraction also simplifies porting to other devices of different specifications, although the clock frequency and the Lookup Table (LUT) usage may need reconfiguration. In our case, the total design occupies 7093 Configurable Logic Block (CLB) LUTs out of 274080 total or 2.59%, while the amount of Block Ram (BRAM) tiles used, are 65 out of 912 total or 7.13%. The HLS-generated design is responsible for retrieving the performance information of profiled functions gathered by the counters in the PMU, and then



(a) AccProf instrumentation engine example. Yellow rectangles are the injected instructions.

(b) function call tree example. An example tree is displayed at the top and its saved state in the BRAM at the bottom.

Fig. 2: AccProf instrumentation engine example on the left and a function call tree example on the right.

organise and store them in such a way, that later they can be read in the form of a function call-graph.

The information (e.g. events and processor cycles) of the profiled functions, the function IDs and the event IDs, are stored by the HLS design formatted as a call-graph in a BRAM inside the FPGA fabric. The process of call graph generation is shown in Fig. 2b. The left image shows a call-graph tree example at the top with a number of functions A-F which can be profiled by AccProf. The bottom visual array displays how that call-tree is stored in BRAM by the HLS-generated design. Functions are stored in chunks by the order they are called. Each chunk contains the function id and level in order to make associations at the end. Profiling Functions of the highest call level are assigned to level 1. In this case functions A and B. Direct callees of each function are assigned to a level determined based on their callers level. The design does not need to traverse the BRAM every time it requires to find the caller of the currently operated function. Instead it uses a stack to push function IDs when they are called, and pops them when they return. So the caller and its level can be accessed immediately with a simple stack peek.

The HLS-generated design has two phases of operation. The first when it receives data from the instrumented application at the prologue of a profiling function and then a second at the functions epilogue. The design exposes to the software eight 64-bit configuration registers. Seven of these are used for the application to write profiling information which include the number of processor cycles and up to six events specified by user. The eighth is used to store the function ID and the IDs of the 6 events to count. Event IDs to the currently supported architectures do not take up more than a byte, so six bytes are used for the event IDs. Another byte is reserved to write the function ID. AccProf

associates each profiling function with a positive integer ID. The hardware design uses another byte of the last word to store that ID. The remaining byte is not written by the application, but is used by the design to compute the function level based on the call tree that has been created up to that point. Below we explain the two sequences according to Fig. 3.

- Hardware checks the function ID. If it's positive, the data was received from the function's prologue. Then the stack is peeked, and if a parent exists, the function is assigned the parent's level + 1 or 1 otherwise. The level and metadata are stored in BRAM to the next available location indexed by a register, and that index is pushed in the stack. The the register is incremented to point to the next available index in BRAM. We move the index 64 bytes forward to make enough room to store required information.
- Epilogue sequence: If the function ID is zero, the data was received from the function's epilogue. Initially the stack is popped. The top element of the stack contains the index of the function we are currently profiling as it's index was pushed in the stack during its prologue. We assume top element of the stack is the correct element as all direct callees of that function should have returned before its epilogue. The last step is to store the profiling information of the function in their corresponding location in the BRAM.

By moving part of the instrumentation code and profiling data storage to the FPGA, several shared resources of the general purpose subsystem (and measured with performance counters) are less affected by cache or TLB pollution. For example, Level 1 Instruction (L1I) Cache will be accessed by fewer instructions associated with AccProf when compared with a software-based profiling tool. Similarly, Level 1 Data (L1D) Cache will not be accessed for reading or writing the profiling data structures, since those are now located in the FPGA memory; BRAM.

3.3 The SeL4 microkernel and Pass - HLS communication

Communication between the instrumentation engine and the HLS design is realised by taking advantage of the SeL4 microkernel [12]. SeL4 is a formally verified operating system microkernel aimed among others for safety-critical and secure-critical systems [18]. After the kernel of SeL4 boots, an initial root task thread is started. That thread is responsible for instantiating any subsequent thread. SeL4 uses the concept of software capabilities [1]. Capabilities give running threads permission to access entities in the system such as physical memory.

The root task is initially given access to all physical memory in the system, and it is responsible for mapping memory and distributing it among other threads that have been instantiated. In our testing framework, the root task is considered part of our design, and any application that is set to run in the system must be started by the root task. The root task will map the physical address of the HLS configuration ports into the virtual address space of the application that will be profiled. It is able to statically specify a virtual address

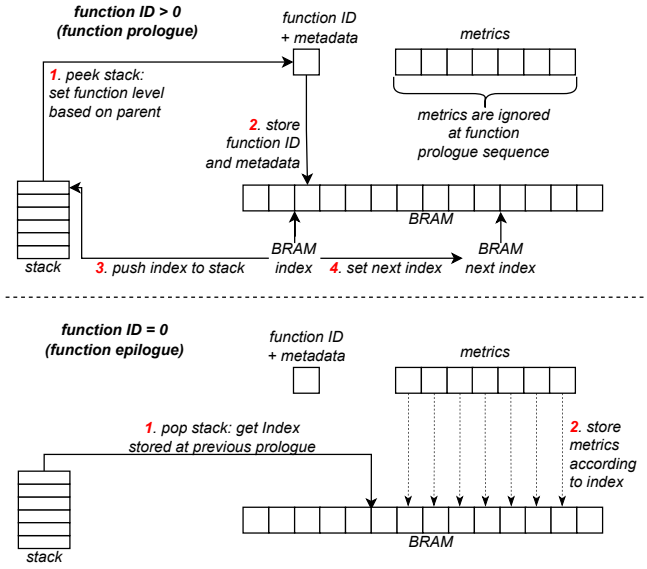


Fig. 3: Hardware design sequences. Top shows when data is received from the instrumented application during the function prologue, and bottom show the operation where data is received during the function epilogue.

which is also known by the instrumentation engine. The engine will instrument loads and stores to that same virtual address to the profiling application during compilation. The physical address of the HLS design is defined during the design phase, which allows the root task to know which physical address needs to be mapped.

4 Evaluation

The evaluation demonstrates the functionality of our profiling framework by applying it to several standard benchmarks. The key benefit of AccProf is that it lowers the impact on performance counter gathered statistics such as data or instruction cache refills during profiling. It achieves this by offloading the storage and processing of statistics and the call-graph to an FPGA IP block. Consequently, the larger the call-graph, the greater the benefit of hardware assisted profiling.

4.1 Testbed

AccProf was evaluated using the the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit made by AMD Xilinx [3]. The kit contains the XCZU9EG-2FFVB1156I MPSoC and 4GB of DDR4 SODIMM. The MPSoC is composed of four A53

ARM cores and associated two level cache system as well as an integrated FPGA connected to the bus of the last level cache (L2 cache).

4.2 The STREAM benchmark

STREAM [15] measures memory bandwidth of applications. It instantiates three arrays A, B and C of double type and then performs 4 simple algorithms:

- Copy: A(i) = B(i)
- Scale: A(i) = q*B(i)
- Add: A(i) = B(i) + C(i)
- Triad: A(i) = B(i) + q*C(i)

Through in-code defines one can configure the number of times the above computations can be executed and the size of the arrays. By default the four computations are executed raw in the main function or each within its own function. We choose the latter since we want to call and profile as many instances of functions as possible in order to generate a large call graph. We set the number of times that the computations will run to 500. This equals to 2000 function calls. As discussed in the Design section, each profiling instance of a function occupies 64 bytes in BRAM. 2000 function calls require almost 128KBytes. Then we run STREAM for 3 different array sizes. 1K (1000), 10K (10000) and 100K (100000) elements. In our system a double is 8 bytes so these 3 configurations produce a total dataset of 24K, 240K and 2.4M (2400000) bytes respectively. We then modified STREAM to experiment on a different pattern which from now on we will call STREAM-mod. In STREAM-mod, we call Scale and Add inside Copy, then Scale and Add each call Triad twice. This pattern produces multiple three-level call-graphs of seven nodes each. We ran copy 285 times which sums to 1995 instances of functions. We configured STREAM-mod for 1K, 10K and 100K array sizes. We counted 6 performance counters related to memory accesses. Those are:

- L1D (Level 1 Data) cache refills
- L1I (Level 1 Instruction) cache refills
- L1D (Level 1 Data) TLB refills
- L1I (Level 1 Instruction) TLB refills
- L2D (Level 2 Data) cache refills
- L2D (Level 2 Data) cache writebacks

We counted those metrics on all 6 configurations first using AccProf, then AccProf-soft. We compiled all versions with -O0 flag in order to get the worst case overhead on all cases, removing possible optimisations. Table 1 shows the results.

We observe around a 43% decrease in L1D cache refills for 1K array size in the hardware version compared to software only instrumentation. This is due to the memory of the array storing the call-graph, which is 128K bytes occupying most of the host memory in the software version, since the three arrays occupy only 24K bytes. This causes multiple cache evictions, while in the hardware

version, the call-graph array is stored in the FPGA. In STREAM-mod for 1K array size we count a 79.5% decrease in L1D cache refills, while we observe a 71% decrease in AccProf running on STREAM-mod compared to STREAM. This is most likely due to how the arrays are accessed. Array C is accessed more frequently in STREAM-mod causing fewer cache misses. Another reason is when the call-graph array is accessed. In STREAM it is accessed in the function prologue, then the function's computation is performed, then the call-graph array is again accessed in the epilogue where we use more indices of the array than in the prologue to store the information. In STREAM-mod since we have nested calls, the call-graph array is accessed consequently starting at the epilogue of the second Triad of Add, causing fewer misses for the software version. When we increase the three arrays' size, refills are dramatically increased for both implementations with only minimal differences. This is natural since STREAM's dataset becomes a lot larger than the call-graph, becoming the primary reason for cache evictions.

For the cache refills, AccProf behaves similarly in STREAM and STREAM-mod. All of the instructions required to access the call-graph array and stack that keeps track of the active call instances have been moved to the FPGA. In AccProf-soft, those instruction run on the host, polluting the instruction cache. Again the reason that STREAM suffers from more refills lies with the access pattern. The instructions that write the profiling information to the call-graph array and pop the stack happen consequently starting at the epilogue of the second Triad of Add, meaning that the instructions keep getting hits in the cache, however in STREAM-mod, instructions interchange with profiling instruction causing the L1I cache to continuously evict instructions. We see similar results when we increase the array size, since this does not affect the pattern in which the instructions are accessed.

When we examine the L1D and L1I TLB refills, we do not observe significant differences between AccProf and AccProf-soft on any of the six configurations. The reason is that the size of the call-graph array for Data TLB is not large enough to cause many evictions since each translation of a memory address corresponds to 4K bytes. Same goes for instructions. Counts are increased the more we increase the three arrays' size which is what primarily affects the TLB.

Regarding L2D cache refills and writebacks, L2 Cache is 1MB so presumably it cannot be filled by 1K or 10K array size configurations or the call-graph array. We observe less than 1000 refills for 1K array size and around 4000 for 10K due to prefetching.

4.3 Embench Benchark Suite

Embench [2] is comprised of 22 different benchmarks that are designed to fit in a small memory footprint of 64KB. The evaluation uses Embench benchmarks as an example of embedded applications with very simple call graphs, that cannot benefit from AccProf. The reason being that each benchmark is only ran on a single function, producing a single-node call-graph. The differences between Ac-

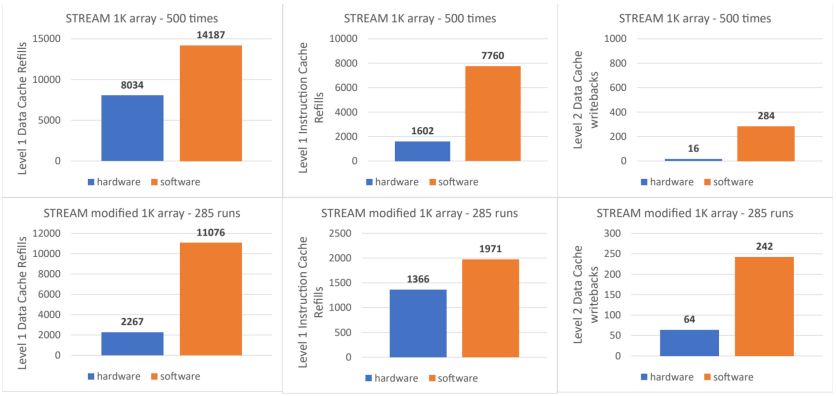


Fig. 4: Level 1 Data Cache refills for the 6 different STREAM configurations profiled by the hardware version of AccProf (blue) vs the software version (orange).

Table 1: Profiling information gathered when running STREAM(500 runs) and STEAM-mod(285 runs) on AccProf and AccProf-soft with 1K 10K and 100K array configurations.

STREAM, 500 function calls

Performance Counter	1K Arrays		10K Arrays		100K Arrays	
	AccProf	AccProf-soft	AccProf	AccProf-soft	AccProf	AccProf-soft
L1D Cache refills	8034	14187	1000917	1005146	9760997	9786194
L1I Cache refills	1602	7760	1631	5618	1356	4957
L1D TLB refills	14	10	65	66	270347	270025
L1I TLB refills	12	9	11	15	2155	2174
L2D Cache refills	986	967	4168	4201	46004152	46101908
L2D Cache writebacks	16	284	4875	5131	22139532	22142185

STREAM-mod, 285 function calls

Performance Counter	1K Arrays		10K Arrays		100K Arrays	
	AccProf	AccProf-soft	AccProf	AccProf-soft	AccProf	AccProf-soft
L1D Cache refills	2267	11076	1222654	1221552	12491279	12445746
L1I Cache refills	1366	1971	1430	1843	1031	2014
L1D TLB refills	15	13	60	67	294262	294771
L1I TLB refills	11	14	14	13	1798	1669
L2D Cache refills	761	943	3849	3989	56476217	56489255
L2D Cache writebacks	64	242	4817	4712	23832926	23841567

(a) Level 1 Data Cache refills (b) Level 1 Instruction Cache refills

Fig. 5: Level 1 Data (left) and Level 1 Instruction (right) Cache refills for the 22 benchmarks of the Embench suite profiled by the hardware version of AccProf (blue) vs the software version (orange).

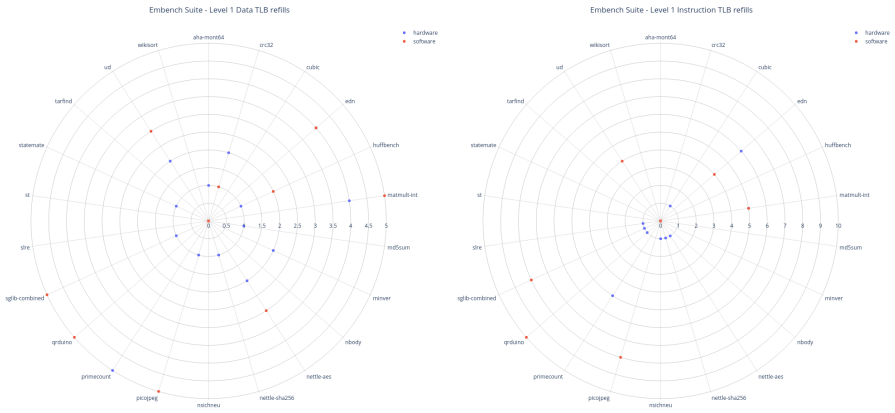
cProf and AccProf-soft are minor most probably caused by prefetching. Results are shown in figures 5, 6 and 7.

Figure 5a shows L1D Cache refills. The biggest difference we observe is for the sglib-combined benchmark. Sglib-combined uses the simple generic library for C and performs an array sorting then manipulates different types of list such as a linked list or a red black tree. The additional overhead in terms of memory for AccProf-soft would be only 64 bytes required for the the function running the benchmark. The difference between them is only slightly above 100 refills, which is expected since the dataset surpasses the 32K bytes of the L1D Cache size. Similarly for all other benchmarks the difference is negligible.

Figure 5b shows L1I Cache refills. In terms of instructions again the only difference is that AccProf-soft accesses the call-graph array twice in the prologue and epilogue of the benchmark's function. The biggest difference is again on sglib-combined which is slightly less than 400 refills. This degree of fluctuation can easily occur due to the large amount of instructions added with the system load interference, the hardware variability, cache eviction policy and so on.

For L2D Cache refills in Figure 7a, we can spot the biggest difference in ud, sglib-combined, qrduino and picojpeg benchmarks. Those differences are mostly slightly more than 200 refills which can be justified by similar reasons as the L1D cache refills.

Regarding L1D and Instruction TLBs in Figure 6 and L2D Cache writebacks in Figure 7b, we do not observe counts larger than the amount of 10, so there is no actual difference between the two systems for that case.



(a) Level 1 Data TLB refills (b) Level 1 Instruction TLB refills

Fig. 6: Level 1 Data (left) and Level 1 Instruction (right) TLB refills for the 22 benchmarks of the Embench suite profiled by the hardware version of AccProf (blue) vs the software version (orange).

(a) Level 2 Data cache refills (b) Level 2 Data cache writebacks

Fig. 7: Level 2 Data cache refills (left) and writebacks (right) for the 22 benchmarks of the Embench suite profiled by the hardware version of AccProf (blue) vs the software version (orange).

4.4 Discussion

STREAM and Embench results illustrate the type of applications that can benefit from a profiling framework that decouples profiling data processing from the instrumentation. STREAM is an example of an application with a large number of function calls each with a small number of computations. On the opposite side of the spectrum, we encounter Embench. Each benchmark in Embench contains the compute task in a single function call.

5 Conclusions

Applications running on embedded systems can be optimized by analyzing profiling information. Current software profiling tools target mainly non-embedded scenarios and impose a significant overhead in how they store and collect CPU statistics. On the other hand, hardware solutions may be more accurate but only work for the specific processor they have been implemented as an extension.

We have proposed AccProf, a hybrid profiler that instruments applications in order to count metrics of specified functions using the hardware performance counters of the underlying system. AccProf offloads profiling actions to a specialized hardware design to collect those metrics and create a call-graph of that application in FPGA storage.

We have evaluated AccProf against an equivalent all-software profiler. When offloading data processing and storage instructions from the instrumentation engine to the FPGA, the profiling framework can collect more accurate CPU statistics from the application since those instructions no longer run on the CPU.

As future work we will look to extend AccProf to support multi-threaded embedded applications. A more fine-grained design could also dynamically redistribute memory (BRAM) according to the runtime behaviour of each thread.

Acknowledgments. This work is partially funded by the UK Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme delivered by UKRI as part of the MoatE (10017512) and Soteria (75243) projects and EPSRC EP/T026995/1 (EnnCore project). Mikel Luján is supported by a Royal Society Wolfson Fellowship and an Arm/RAEng Research Chair Award.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. https://docs.sel4.systems/Tutorials/capabilities.html
2. GitHub - embench/embench-iot: The main Embench repository — github.com. https://github.com/embench/embench-iot, [Accessed 16-12-2023]
3. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit — xilinx.com. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, [Accessed 15-12-2023]

4. Aldham, M., Anderson, J., Brown, S., Canis, A.: Low-cost hardware profiling of run-time and energy in fpga embedded processors. In: ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors. pp. 61–68. IEEE (2011)
5. Bailleu, M., Dragoti, D., Bhatotia, P., Fetzer, C.: Tee-perf: A profiler for trusted execution environments. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 414–421. IEEE (2019)
6. Beischl, A., Kersten, T., Bandle, M., Giceva, J., Neumann, T.: Profiling dataflow systems on multiple abstraction levels. In: Proceedings of the Sixteenth European Conference on Computer Systems. pp. 474–489 (2021)
7. Burtscher, M., Kim, B.D., Diamond, J., McCalpin, J., Koesterke, L., Browne, J.: Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. IEEE (2010)
8. Coussy, P., Gajski, D.D., Meredith, M., Takach, A.: An introduction to high-level synthesis. IEEE Design & Test of Computers **26**(4), 8–17 (2009). https://doi.org/10.1109/MDT.2009.69
9. Diamond, J., Burtscher, M., McCalpin, J.D., Kim, B.D., Keckler, S.W., Browne, J.C.: Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In: (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software. pp. 32–43. IEEE (2011)
10. Graham, S.L., Kessler, P.B., McKusick, M.K.: Gprof: A call graph execution profiler. ACM Sigplan Notices **17**(6), 120–126 (1982)
11. Hill, J.M., Jarvis, S.A., Siniolakis, C.J., Vasilev, V.P.: Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In: Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing-PDP'98-. pp. 286–294. IEEE (1998)
12. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: sel4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220 (2009)
13. Lattner, C.: Llvm and clang: Next generation compiler technology **5**, 1–20 (2008)
14. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International symposium on code generation and optimization, 2004. CGO 2004. pp. 75–86. IEEE (2004)
15. McCalpin, J.D.: STREAM benchmark. Link: www. cs. virginia. edu/stream/ref. html# what **22**(7) (1995)
16. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The paradyn parallel performance measurement tool. Computer **28**(11), 37–46 (1995)
17. Nair, A., Lysecky, R.: Non-intrusive dynamic application profiler for detailed loop execution characterization. In: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems. pp. 23–30 (2008)
18. SeL4 Foundation: About SeL4, https://sel4.systems/About/
19. Shankar, K., Lysecky, R.: Non-intrusive dynamic application profiling for multitasked applications. In: Proceedings of the 46th Annual Design Automation Conference. pp. 130–135 (2009)
20. Shannon, L., Chow, P.: Using reconfigurability to achieve real-time profiling for hardware/software codesign. In: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays. pp. 190–199 (2004)