Speedup Factor Estimation through Dynamic Behavior Analysis for FPGA

Zhongda Yuan[1], Jinian Bian[1], Qiang Wu[2], Oskar Mencer[2]

[1] Dept. of Computer Science and Technology, Tsinghua Univ., Beijing 100084, China
[2] Department of Computing, Imperial College London, SW7 2BZ, UK
yzd06@mails.thu.edu.cn, bianjn@tsinghua.edu.cn
{qiang.wu, o.mencer}@imperial.ac.uk,

Abstract

In reconfigurable platform, before convert and download program into real hardware, reliable estimation of speedup factor is of great importance for task schedulers. In this paper, a novel technique for speedup factor estimation is proposed. From the event patterns collected by hardware counters built in modern processors, a formula is given to estimate speedup factor of target process. Experiments on programs from SPEC2006 show that the speedup feature is able to be estimated at an acceptable cost.

Keywords: performance event counter, speedup estimation

1. Introduction

With the progress in Reconfigurable Computing technology, CPU+FPGA hybrid platforms become increasingly popular in both academic and industry world. Various architectures, algorithms and tools are proposed to accelerate software programs with FPGA accelerators [1]. Different speedup factors are reported, ranging from tens to hundreds, according to the application and program being transformed and the hardware platform they used.

It is widely accepted that transforming existing program into FPGA hardware code is time consuming. Before convert and download program into real hardware, reliable estimation of speedup factor will save much work by eliminating unattractive ones.

Speedup factor is defined as the execution time ratio between modified hybrid-platform and traditional Von's platform. Published researches [2-3] focus on afterward speed-up factor report. Little has been reported about speed-up factor estimation before implementing a hardware version of targeted algorithm.

Event counter is nearly a standard feature for modern processors and exist on most major processors today, such as Intel Pentium, Core, IA-64 [4] and AMD Opteron [5]. Different kinds of events can be recorded by event counters while processors are executing programs, including the clock cycles elapsed, instructions ever executed, cache misses, branch prediction failures, and so on[3].

Previous researches using event counters mainly focus on gathering performance statistics to evaluate the underlying hardware architectures or help programmers in code optimization and system management [6-9], leaving speed-up factor estimation as a blank section.

In this paper, a technique using event counters built in modern processors to estimate speedup factor is proposed. Performance event information of the program is collected at run time. By processing the records of the events, metrics of the event series are calculated and compared with the reference values to sort out the features of the monitored process.

The rest of the paper is organized as follows: Section 2 discusses the related work about determining speedup factor and the technique and applications about build-in event counters in modern CPUs. Section 3 gives a detailed design on speedup factors estimation by collecting event patterns online. Section 4 introduces the experiment including the test platform, performance counter tools, test programs and the results. Discussion and conclusion are given in section 5 with some hints to future work.

2. Speedup factor

Though there are number of reports about successful accelerating hardware / software co-designs [1][3], few have explored the potential about a hybrid-platform before implementing certain algorithm in hardware [2]. This vacuum may be explained by the following two reasons.

The main reason is wide variety of hybrid platforms and target program. Since reconfigurable computing is an emerging technique, it has a long way to become standardized. Nearly each team interested in this realm, has their own hardware layout, from integrating FPGA cells into general purpose CPU, to attach a FPGA card to the slow ISA slot. Parameters of reconfigurable resource is much more varied, including number of RCUs (Reconfigurable Units), build-in memory capacity and the data width of IO channel. The algorithm under scrutiny focuses on multimedia compression / decompression, data encryption / decryption, biology and wireless application. These application share one common characteristic – data flow is much heavier than instruction flow.

It is very common for published online scheduling research to assume that the conversion from software algorithm to hardware specification is finished beforehand. Some even assume the execution time can be deduced at the time when a hardware task is arriving. This assumption is acceptable as long as experimental platform is designed for special usage, such as data compression and or encryption [2]. But for a general purpose hybrid-platform system, online converting and scheduling shall consider the conversion cost and the performance gain before take any substantial action. In a performance-critical system, online services cannot be stopped. The kernel loop must be identified, located, converted and finally deployed into FPGA-based co-processing components. In this environment, managing module will be able to pick the most promising one, so as to improve the overall performance, when the beneficial-cost rate of a certain task can be estimated with enough accuracy.

3. Speedup factor estimation

3.1. Hardware and software Support

We setup the experimental a system with Pentium 4 processor (family 15, model 2, stepping 4) running Fedora Core 5 Linux. The processor has a working frequency of 1.8 GHz, 512KB L1 cache. It contains 18 available performance data registers, 65 performance control registers, supporting 46 types of events to be counted.

The kernel of the Linux is of version 2.6.22.9 downloaded from kernel.org [10] and patched with *perfmon* performance counter driver [11]. On this platform, a background daemon is designed to collect events from certain running process. Performance events, such as CPU cycles, instructions completed and memory access can be simultaneously stored in hardware registers while the CPU processing its instruction flows. The monitoring daemon only need to periodically checkout the value and reset the counter. The monitoring and logging overhead is limited.

3.2. Dynamic behavior observation

Cycles spent on traditional general CPUs includes instruction code memory access, memory access on data, and algorithm on data processing. Time needed for a task run in FPGA hardware consists of hardware setting up, data transfer, and calculation on FPGA. Statistics show that data transfer codes have taken up to 80% [3]. When a program was translated into hardwire circuit on a FPGA chip, the performance gain comes from the following reasons.

Calculation steps on traditional CPU are integrated into one single step in FPGA. Temporary variables that cannot be handled by registers with in CPU core, can be handled by hardwire within FPGA, for its unlimited register number. When the hardware resource to implement algorithm is enough, less IPC, may yields more speedup factor.

Data transfers are handled in a pipeline mode in traditional CPU. FPGA hardware often suffers from slower interface to memory. So, those algorithms with heavy data transfer will gain less than those without too much data access.

A sample program have three types of kernel loop, namely non dependant loop, carry dependant loop and a mixed loop, which loop through an array randomly.

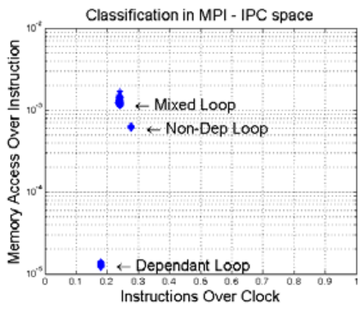

Fig. 1: classification in MPI-IPC space

In Figure 1, the blurred dots are actually clusters of samples, which are collected periodically through the build-in hardware-based Performance Counters. It can be observed that non-dependant loop, represented by the lower cluster of sampled points, enjoys better performance, or instructions completed per CPU clock, because current CPU pipelining technique suite the kind of task better. Carry dependant loop suffers from the worst performance because the pipeline is more frequently stalled for it have to wait for the output of former iteration to start a new one. In the clock to memory view, carry dependant loop have lest memory access because its iteration need only a small area to store the variables and thus encountered much less memory access. The non-dependant loop, simply iterate through a big array, have more memory access than dependant loop. At the same time, sequential access enabled the processor to handle cache more efficiently than the random version, the mixed loop, which fires event more memory access than non-dependant loop.

3.3. Speedup factor estimation

Based on these observations, we give the following formula to estimate the speedup factor of a target program.

$$F = C - A * IPC - B * \log(MPI) \quad (1)$$

In the formula above, F is the friendly level. Friendly Level can be defined as the expectable speedup factor. *A*, *B* make up a gradient in IPC-MPI space, and *C* set up a reference line. These parameters are simply estimated by experiments and may vary a little, since different hardware configurations may yield different speedup result.

4. Experiment

4.1. Test on programs from CPU SPEC 2006 benchmarks

Testing is carried on CPU SPEC 2006 benchmark programs [12] to further verify our estimation, and the results go promising as Table 1 show the event matrix from spec benchmarks. Estimation here is calculated with parameter set as $A = 9$, $B=1$ and $C = 1$, which is an empirical value set.

From the results in Table 1, we can safely point out that, kernel loops within programs like *astar* and *bzip* have complex dependent map, not very suitable for existing pipelines and FPGA accelerators; while kernel loops within programs like *bwaves* are loop carry dependent, in which most computations are restrained in local variables, and are suitable for FPGA accelerators.

It can be seen that from the experimental data collected by performance counters, the character of the target program deduced is consistent to statistical analyzes. The potential to accelerate can be estimated.

Table 1. Testing results from CPU SPEC 2006

benchmark	Samples	IPC	MPC	MPI	Speedup Estimation
Astar	860	0.69129	0.12596	0.71874	-4.8914
Bwaves	10462	0.37421	0.016608	0.05784	0.4822
bzip2	624	0.37983	0.26177	0.91946	-2.3345
cactusADM	2560	0.4148	0.058203	0.14076	-0.7725
Calculix	4280	0.70606	0.0060611	0.0070686	-0.4024
Gromacs	1636	0.48332	0.0071375	0.014758	0.8661
H264ref	1304	0.75604	0.048401	0.066257	-3.0901
Lbm	1711	0.2866	0.79047	2.7618	-2.5953
leslie3d	1780	0.47402	0.077958	0.16446	-1.4611
Libquantum	1906	0.81747	0.011527	0.015657	-2.2004
Mcf	9268	0.1857	0.017861	0.090857	1.7272
Milc	2574	0.25213	0.01202	0.077876	1.2835
Sphinx	1796	0.61894	0.022065	0.035858	-1.2423
Wrf	2307	0.63282	0.048812	0.09304	-2.3207
Xalan	1397	0.34754	0.0077023	0.025589	1.5377
Zeusmp	14794	0.31672	0.030507	0.095731	0.4957

5. Discussion and conclusion

To estimate the speedup factor of running task on running system, we use performance events data collected by PMU to pinpoint the most used codes, to determine the feature of that sequence of codes, and to estimate the potential of target process to be accelerated by means of binary modification. Experiments show that the speedup feature can be estimated with the performance event counters at an acceptable cost. From our experiment, the conclusion can be drawn that from performance pattern observed on a process, its feature can be safely estimated.

More experiment on various type programs can make this conclusion more convincing and concrete. Currently we use overall dynamic behavior as the estimation base, but actually program have phases, i.e. stages that focus on deferent operations. Dynamic behavior analysis can be implemented so as to make more accurate estimation on phrases. Detailed restrains, such as hardware capacity for algorithm, memory interface bottleneck, are simply assumed to be irrelevant in this paper. Further research effort can focus on these limits and yield a more concrete conclusion.

This work is supported by NSFC No. 90607001 and the EPSRC EP/C544706/1, EP/C544692/1.

References

[1] Tim J. Todman, A George. Constantinides, Steve J.E. Wilton, Oskar Mencer, Wayne Luk, Peter Y.K. Cheung.: Reconfigurable Computing: Architectures and Design Methods. IEE Proceedings on Computers and Digital Techniques. 152(2), 193--207 (2005)

[2] Zhi Guo, et al., A Quantitative Analysis of the Speedup Factors of FPGAs over Processors. In FPGS '04, February 22-24, 2004, Monterey, California, USA. 162—170(2004)

[3] Osman Devrim Fidanci, et al., Performance and Overhead in a Hybrid Reconfigurable Computer, Proceedings of IPDPS'03.

[4] Intel Corporation. http://www.intel.com/

[5] Advanced Micro Devices, Inc. http://www.amd.com/

[6] M.J. Serrano, Youfeng Wu: Memory performance analysis of SPEC2000C for the Intel(R) Itanium processor. In: Proc. of Intl. Workshop on Workload Characterization, pp. 184--192 (2001)

[7] Bill Maron, Thomas Chen, Duc Vianney, et al.: Workload Characterization for the Design of Future Servers. In: Proc. of Intl. Workshop on Workload Characterization, pp. 129--136 (2005)

[8] J.M. Anderson, L.M. Berc, J. Dean, et al.: Continuous Profiling: Where Have All the Cycles Gone? ACM Transactions on Computer Systems, 15(4), 357--390 (1997)

[9] E. Duesterwald, C. Cascaval, Sandhya Dwarkadas.: Characterizing and Predicting Program Behavior and its Variability. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques, pp. 220--231 (2003)

[10] The Linux Kernel Archives. http://kernel.org/

[11] HP Labs. Perfmon project. http://www.hpl.hp.com/research/linux/perfmon/

[12] Standard Performance Evaluation Corporation http://www.SPEC.org