Practical Techniques for Performance Estimation of Processors

Abhijit Ray, Thambipillai Srikanthan and Wu Jigang
Centre for High Performance Embedded Systems
Nanyang Technological University, Singapore
Email: {pa8760452, astsrikan, asjgwu}@ntu.edu.sg

Abstract

Performance estimation of processor is important to select the right processor for an application. Poorly chosen processors can either under perform very badly or over perform but with high cost. Most previous work on performance estimation are based on generating the development tools, i.e., compilers, assemblers etc from a processor description file and then additionally generating an instruction set simulator to get the performance. In this work we present a simpler strategy for performance estimation. We propose an estimation technique based on the intermediate format of an application. The estimation process does not require the generation of all the development tools as in the prevalent methods. As a result our method is not only cheaper but also faster.

1 Introduction

A typical embedded program runs on a processor with parts of the program running in hardware so that performance requirements are met. It is important to select a processor whose performance is close to the performance requirements of the application. Selecting a processor which under-performs by a huge margin means a very large piece of hardware have to be used to meet the requirements. On the other hand, an over-performing processor would be very expensive which would reflect in the higher unit cost of the product. The current estimating methods rely heavily on compiling and then executing the application on a instruction set simulator. This is a time consuming process and even more so when you have to consider a large number of processors and have to evaluate their performance. There is the added cost of the development tools itself.

This paper proposes a simple estimating methodology which does not require the full compilation of the source code. The rest of the paper is organized as follows. Section 2 gives a brief description of related work. In section 3 we describe the methodology in detail. In section 4 we provide some of our results, followed by the conclusion and future work in section 5.

2 Related Work

Most of the previous work depend upon generation of the development tools like compilers, assembler and instruction set simulators. Their main contribution is a processor description language which contains enough information required to generate compilers, assemblers, debuggers and instruction set simulators. Using the above tools the given application is compiled and the executable file is run through the instruction set simulator to get the performance estimates. nML is the processor description language described in [1], sim-nML in [2–4] and ISDL in [5]. The only difference in the above methods are the amount of detail that can be expressed in the respective languages.

In [6], the performance estimation methodology involves changing the low level code by inserting counters on those input stream elements that are decisive for the computational flow within the application. From the data extracted a complexity profile of the application is generated and the performance estimate is then obtained by weighing the execution frequencies $F(i)$ of core tasks i with the number of clock cycles $C(i)$ required per task, resulting in an overall performance figure:

$$p_{total} = \sum_i F(i) \cdot C(i)$$

Suzuki et al [7] abstract their application in a form of representation called codesign finite state machines(CFSMs). They use the POLIS framework to partition the CFSMs to identify the components of design that are candidates for software implementation. The CFSMs corresponding to this partition is then mapped into another representation called a software graph(s-graph), which is then optimized. For software performance estimation the s-graph represented into C code. The execution time is modeled by adding the time for entering and exiting each function, the

Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS'05)
1098-8068/05 $20.00 © 2005 **IEEE**

time to initialize the variables and the time spent in executing the conditional statements. The time required for execution of different control structures are obtained by running sample benchmark programs.

3 Estimation Technique

We perform our estimation using an intermediate format which while being low-level enough is still processor independent. For this we have used the Low Level Virtual Machine (LLVM) format [8]. LLVM was designed for use as a compilation framework for research into compiler optimizations across the entire lifetime of a program. The LLVM virtual instruction set is a low level program representation using simple RISC-like instruction. The key reason why we have chosen LLVM instead of a more popular compiler system like the SUIF is, LLVM intermediate format do not need to be converted back to high level language like C after transformation. Also, LLVM incorporates a virtual machine which can run the LLVM object codes from which profile data can be easily extracted. The application is first converted into the LLVM intermediate format and the program details are extracted from the intermediate format and then it's executed on the virtual machine to generate profile output. The output of the above two steps is used by the estimator to provide the performance estimates. Below we describe in detail the estimation framework.

We assume that the application is provided as a C/C++ program. This does not limit our methodology in any way. The application is converted into the LLVM format using the gcc front-end available from the LLVM web page [9]. The front-end converts the input programs in C/C++ into a bytecode file. From this bytecode file, we extract the intermediate format instructions for it. For this we have written a small pass, which iterates through all the instructions and prints out the relevant information, like the number of arguments in a function call.

After the initial intermediate code is passed through processor independent optimizations, the resultant intermediate format instructions is used to make an educated guess of the final machine code that would be produced. For this many programming constructs were compiled, both till intermediate format and right up to the final executable format. The results were analyzed, and the intermediate code and the the final object code produced were compared. This helped us to form an idea of what kind of intermediate format result in what kind of object code. Right now we have done this only for the **arm** processor. Once the methodology is validated for it, it can be extended to a range of processors.

Following is the logic how we arrive at a object code guess from the intermediate code in brief. Let's take the call instruction. In the LLVM intermediate format we have the call instruction as

```
%tmp.6.i15 = call int %fib(int %tmp.i14)
```

The above intermediate format instruction calls a function called test which returns an integer and has an integer argument. The same code in the arm machine code is,

```
ldr     r0, [fp, #-16]
bl      fib
```

Now we can see that a call instruction results in a few **mov** instructions (which moves the arguments to the registers and then a branch instruction which results in the actual transfer of control to the called subroutine). From this we speculate that any call instruction will result in a few **mov** instructions(the actual number being equal to the number of arguments) and a branch instruction. So we come up with the formula for the number of machine instructions for a call instruction in the intermediate format, which is given below.

$$n = num\_args + 1 \tag{1}$$

where num_args = number of arguments.

This is fine for user defined functions as we have the source code, which can be compiled to the intermediate format and estimates obtained. But, that would be a very time consuming process for the many standard library functions and moreover you would also need to get the source code of the whole standard library, which would need to be compiled first and then estimated. Hence for obtaining the estimates of standard library functions we have used a simpler and a more direct technique, which we have explained later.

Similarly different programming constructs were analyzed, which could possibly give rise to different machine codes. The results were used to come up with a framework to guess the code that a compiler will produce.

While this worked for all different intermediate language instructions, there were problems estimating the code for standard library functions. For example for a call to the math library function $sqrt()$, the intermediate format just puts in a call to the function. But the performance of a processor should also include the estimates for the the standard library functions. For this one way would be to compile all the standard library functions with LLVM and then use similar technique as described above to obtain estimates. Below we describe the methodology used for estimating the performance for the standard library functions.

As noted above, one way to get the estimates for all the standard library functions would be to, compile all of the standard library functions with the LLVM compiler infrastructure and then use the intermediate format to obtain the estimates using exactly the same process we use to get the estimates for a normal function. This would obviously be

Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS'05)
1098-8068/05 $20.00 © 2005 **IEEE**

a very cumbersome process. Instead we try to make use of the fact that standard library generally comes as a already compiled package, which can be executed directly.

For estimation of standard library functions, we wrote test programs which calls the standard library functions and then ran the program through an instruction set simulator. For this we have used the simplescalar [10]. So we executed the compiled program on simplescalar and obtained the count of number of instructions executed. We use the fast sim-safe/sim-fast etc which runs very fast compared to the detailed simulator like sim-outorder, but gives somewhat less accurate result than sim-outorder. But since our estimates from the rest of the code is not as accurate as those produced by sim-outorder, we don't really gain any accuracy by using a detailed but a very slow simulator. We took care to write our test programs in such a way that the targeted library function gets called a large number of times. This is done so that the number of different instructions getting executed which cannot be attributed to the function for example the program startup instructions etc, gets amortized and we obtain a more accurate result. After the program is executed with the instruction set simulator we get the result in the number of instructions that got executed and this we do for other standard library functions.

Even this process looks cumbersome considering that there are many standard library functions that needs to be taken care of. But fortunately there are many standard library functions which we can safely ignore. Some functions like $fopen()$, $fclose()$, are generally executed only once at program start and program end. Similarly functions like $exit()$ can get executed only once , i.e., when the function exits. So all these functions have negligible impact on the overall program performance, and hence can be safely ignored.

Using the above results, we create a file that describes how each LLVM intermediate language instruction and the standard library function expands to the machine instructions. This we use later to directly obtain the estimates of the given application. Till now we have only the static program estimates, which just tells us what the compiled code might look like. The actual performance will obviously depend on the input data. For this we need to run the code and see. To obtain these we have run the program with some input data and obtain the program profile results. This again we do under the LLVM environment.

This part of the work is simple. LLVM has a profiler,which can run the bytecode file and can give the execution count of each basic blocks in the bytecode. After we have obtained the execution count of basic blocks, we just use our previous results to obtain the estimates for each basic blocks and together with the profiler outputs we can obtain the estimates for the execution of the whole application under a given set of inputs.

Table 1. Error in estimates

Benchmark	Estimates	Actual	error(%)
bitcounts(small)	51357580	49670963	-3.40
bitcounts(large)	768948503	743684377	-3.40
basicmath(s)	62609424	65462194	4.36
basicmath(l)	2434075089	2515566670	3.24
qsort(s)	37645656	43608848	13.67
qsort(l)	590027672	737923507	20.04
patricia(s)	79376292	103926829	23.62
patricia(l)	457172451	640423385	28.61
stringsearch(s)	167262	161605	-3.50
stringsearch(l)	3856090	3682798	-4.70

4 Experimental Works

For our experimental work we have used MiBench [11], a free, commercially representative embedded benchmark suite, as our application. Some of the benchmarks did not compile with LLVM due to LLVM not having support for inline assembly and other reasons. The benchmarks were compiled using the LLVM compiler and the bytecode file was obtained. We wrote a small compiler pass, to extract the application information like the basic blocks in different functions and the intermediate format instruction in each of these basic blocks. The codes for each basic block estimated and then the execution count of each basic blocks were obtained after executing the bytecodes through the profiler.

The same benchmarks were then compiled for arm by gcc to produce arm machine codes. The executable was then executed on the simplescalar tools, to obtain the number of instructions executed. We compare our estimates with the simplescalar output. The percentage error was calculated with respect to the simplescalar results.

The experimental results comparing our estimates with the actual values obtained using a instruction set simulator is given in table 1. The estimated and the actual values are in number of instructions executed. For example, for the benchmark qsort(s), the number of instructions estimated is 167262, while the actual value as obtained by the simulator is 161605. And the percentage error is 3.50.

As can be seen for some of the benchmarks the percentage error is quite high. This is due to the limitations of our methodology wherein we do not consider the processor dependent optimization of the compiler and also other processor characteristics, like the number of registers available. We are currently in the process of incorporating such features into our framework.

We also measured the time taken to arrive at the estimates using our method and compared it with the time taken while using an instruction set simulator. In particular we

Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS'05)
1098-8068/05 $20.00 © 2005 **IEEE**

Table 2. Comparisons of time taken

Benchmark	Proposed technique	sim-outorder
bitcounts(small)	2 sec	$>$ 4 hrs
bitcounts(large)	2 sec	$>$ 4 hrs
basicmath(s)	2 sec	$>$ 4 hrs
basicmath(l)	2 sec	$>$ 4 hrs
qsort(s)	2 sec	$>$ 4 hrs
qsort(l)	2 sec	$>$ 4 hrs
patricia(s)	2 sec	$>$ 4 hrs
patricia(l)	2 sec	$>$ 4 hrs
stringsearch(s)	2 sec	$>$ 4 hrs
stringsearch(l)	2 sec	$>$ 4 hrs

have used the simplescalar tool set. We compare the time taken using our technique with the time taken for running the same application on simplescalar and obtaining the performance results. The timing results are tabulated in table 2.

We can see that our method is considerably faster than using a instruction set simulator. When a lot of processors have to be considered, this can lead to large amount of time being used up for executing the application through a simulator.

5 Conclusion

In this paper we have proposed a simple methodology for performance estimation of processors. The method is simple and does not require the generation of all the development tools as in the case of most of the current methods. Hence we can do away with compiling and simulation for every processor. This allows for inclusion of more processors in the selection process. The estimates obtained can be helpful in selection of a ideal processor for an application. The presented methodology is cheaper and faster than what is used currently.

Overall we can say that our methodology sacrifices accuracy for speed. While the simulator based approach is accurate, the large amount of time it takes may make it unsuitable. Our methodology does have a few limitations. We are however working to bring down the errors in our estimates by taking into consideration the factors affecting a processor performance, like the numbers of registers, pipeline etc.

References

[1] A. Fauth, J. V. Praet, and M. Freericks, "Describing instruction set processors using nml," in *EDTC '95: Proceedings of the 1995 European conference on Design and Test*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 503–507.

[2] R. Moona, "Processor models for retargetable tools," in *RSP '00: Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 34–39.

[3] V. Rajesh and R. Moona, "Processor modeling for hardware software codesign," in *VLSID '99: Proceedings of the 12th International Conference on VLSI Design - 'VLSI for the Information Appliance'*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 132–137.

[4] S. Chandra and R. Moona, "Retargetable functional simulator using high level processor models." in *VLSI Design*, 2000, pp. 424–429.

[5] G. Hadjiyiannis, S. Hanono, and S. Devadas, "Isdl: an instruction set description language for retargetability," in *DAC '97: Proceedings of the 34th annual conference on Design automation*. New York, NY, USA: ACM Press, 1997, pp. 299–302.

[6] H.-J. Stolberg, M. Berekovic, and P. Pirsch, "A platform-independent methodology for performance estimation of streaming media applications," in *Proceedings 2002 IEEE International Conference on Multimedia and EXPO (ICME2002)*, 2002.

[7] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient software performance estimation methods for hardware/software codesign," in *DAC '96: Proceedings of the 33rd annual conference on Design automation*. New York, NY, USA: ACM Press, 1996, pp. 605–610.

[8] C. Lattner and V. S. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation." in *CGO*, 2004, pp. 75–88.

[9] "The llvm compiler infrastructure project," June 2004. [Online]. Available: http:/llvm.cs.uiuc.edu

[10] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[11] "Mibench version 1.0, a free, commercially representative embedded benchmark suite," June 2004. [Online]. Available: http://www.eecs.umich.edu/mibench/index.html

Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS'05)
1098-8068/05 $20.00 © 2005 **IEEE**