Rapid Techniques for Performance Estimation of Processors

Abhijit Ray, Thambipillai Srikanthan and Wu Jigang
School of Computer Engineering
Nanyang Technological University
Singapore, 639798
Email: {PA8760452, astsrikan, asjgwu}@ntu.edu.sg

Current techniques for processor performance evaluation, which rely on using instruction set simulators to estimate the performance for each of the processors, may not be feasible due to the large amount of time taken to run the simulations. Moreover the simulators have to be modified to incorporate the new processor features that get implemented. In this paper we present efficient techniques to estimate processor performance by examining the intermediate level code and approximating the machine level code from it. These include a way to estimate library functions and a technique to estimate the IPC (instructions per cycle) for an application running on a processor. Unlike simulator based approaches our technique does not require the building of a compiler tool chain for the processors. Results in terms of the IPC and the number of instructions executed are also provided in order to fully validate the effectiveness of the proposed techniques.

ACM Classification: B.8.2

1. INTRODUCTION

The software content of the embedded systems grows exponentially, mostly from the migration of application-specific logic to application-specific code, to cut down product cost and time to market (Lajolo *et al*, 1999). The demand for more functionality along with need for shorter development have also led to larger portions of the application to be moved from hardware to software. As a result microprocessors are increasingly becoming more important in embedded system design. While earlier the desired high performance could only be achieved by using ASICs, now the same performance can be obtained using microprocessors. Microprocessors are also more flexible than ASICs and hence significant parts of the application are being moved to software, i.e., programs running on microprocessors. With the rapid advances in microprocessor technology (following the famous Moore's Law), the number of processors available in the market have multiplied many times and this number can only grow in future. Today, the number of microprocessors shipped for embedded applications far outstrips the number shipped for personal computers. Thus the embedded system designers are faced with the increasingly difficult problem of having to decide on a suitable processor for an application.

We have to select a processor which meets the required performance constraints. The performance constraints can usually be achieved using a faster processor. But, this is not desirable as faster processors are more expensive. Embedded systems are also under very tight cost constraints. Hence, an embedded system designer would want to achieve the desired performance using the cheapest possible processor.

Manuscript received: 18 May 2008
Communicating Editor: Zoran Salcic

The current methodology of using simulators for performance estimation is time consuming, which will get worse as more and more microprocessors become available in future. Embedded systems also are under very strict time to market considerations. Simulating an application on a large number of microprocessor simulators would be prohibitive due to the time taken. Moreover the use of simulators requires the development of necessary compiler tools. Hence, the cost of compiler tool-chains for each architecture is added to the development cost. Simulators also have to be modified whenever new processor features are implemented.

In this work, we present a framework for performance estimation of processors which do not require building of all the development tools and also does not require time consuming simulation on an instruction set simulator. We use the intermediate code of an application to arrive at the performance estimates. Performance characteristics of standard library functions are used to arrive at the estimates for standard library functions for which the source codes may not be available. IPC (instructions per cycle) estimates were obtained by observing the relation between the average basic block length and the IPC through experiments. Experimental results on the MiBench benchmark show that error rates as low as 2% can be achieved with our technique. The best results are within 1–2% of the actual values, and almost all benchmarks have error rates of less than 20%.

The rest of the paper is organized as follows. In Section 2, we discuss previous works in performance estimation of processors. In Section 3, we describe our estimation framework and in Sections 3.3 and 3.4, we describe in detail the estimation technique for standard library functions and the procedure employed to estimate the IPC respectively. Experimental works and results are provided in Section 4 and conclusions and future works are discussed in Section 5.

2. RELATED WORK

Most of the previous works on software performance estimation 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 (Fauth *et al*, 1995), sim-nML (Moona, 2000; Rajesh and Moona, 1999; Chandra and Moona, 2000) and ISDL in Hadjiyiannis *et al* (1997) are the different processor description languages that have been used in previous works. Machine description files are written in these languages, which can be used to generate assemblers, compilers and simulators for the processors. The application can be compiled and simulated with these tools to arrive at the performance estimates. The only difference in the above methods is the amount of detail that can be expressed in the respective languages.

In Gupta *et al* (2000), the authors focused on an intermediate representation of the application from which they extract parameters which can affect the processor performance. They also used a parameterized model of a processor. In Baghdadi *et al* (2002), the system is described in SDL, a system-level specification language. The system is then modeled and partitioned by mapping the functions of the system onto interconnected hardware/software processors. For the software realization, the SDL-specified process will be translated into C code. And the execution time is computed by the analysis of the assembled code.

In Stolberg *et al* (2002), 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 extracted data, a complexity profile of the application is generated and the performance estimate is then obtained by weighing the execution frequencies *F(i)* of core task *i* with the number of clock cycles *C(i)* required per task, resulting in an overall performance figure:

$$p_{total} = \sum F(i) \cdot C(i) \tag{1}$$

Suzuki and Sangiovanni-Vincentelli (1996) abstract their application in a form of representation called co-design finite state machines (CFSMs). They use the POLIS framework (POLIS, 2005) to partition the CFSMs to identify the components of design that are candidates for software implementation. The CFSMs corresponding to this partition are then mapped into another representation called a software graph(s-graph), which is then optimized. For software performance estimation the s-graph is converted into C code. The execution time is modeled by adding the time for entering and exiting each function, the 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.

In Worst-case Execution Time (WCET) analysis, the upper bound for the time needed to execute a program is calculated. This is important for hard real time systems, where failure to perform within the timing constraints can be fatal. An operation performed after the deadline is, by definition, incorrect. WCET analysis therefore tries to provide a time value within which the application will finish execution under any circumstances.

Puschner (2002) contends that WCET analysis has had hardly any industrial impact due to the high complexity of implementing and using the proposed WCET approaches. In addition WCET research tends to lag behind microprocessor technology. By the time WCET techniques manages to incorporate the features in one generation of microprocessors, the next generation of advanced architectures with novel speedup features are already in the market. WCET analysis also suffers from complexity of the analysis. In general, the number of paths to be analyzed for an exact WCET analysis of a piece of code grows exponentially with the number of consecutive branches in the control flow of the analyzed code. WCET analysis is also made difficult due to the presence of pipelining and caches (Lundqvist and Stenstrom, 1999; Ferdinand, 2004). As processor manufacturers optimize for the average case of the major benchmarks, the WCET becomes hard to predict, even more so when the different features interact (Puschner, 2002). For advanced processors, there is a difference between average case and observed worst case due to the speed up introduced by the advanced processors like caches, pipelining and out of order execution. These differences increase when applications have data dependent memory accesses or paths. There is inherent variability of the execution time of programs. These difficulties in estimating the WCET leads to pessimistic estimation of the performance.

In all the previous methods, either generation of compiler tool-chains is required or the source code of the application is required. In Fauth *et al* (1995), Moona (2000), Rajesh and Moona (1999), Chandra and Moona (2000) and Hadjiyiannis *et al* (1997), the compiler tool-chain and also an instruction set simulator have to be built before the program can be compiled and then executed on the instruction set simulator. While in Baghdadi *et al* (2002) and Suzuki and Sangiovanni-Vincentelli (1996), the system has to be described in some methodology specific formats, which require the internals of the application to be known in advance. Stolberg *et al* (2002) requires the source to be known so that the instrumentation code can be inserted. Another problem with the simulator based evaluation is that the simulators have to be modified as new processor features are implemented. Similarly the machine description languages have to be changed to allow new features to be described. WCET analysis suffers from complexity of the analysis and also pessimistic estimation. Our work relies on the intermediate format for estimation to estimate the compiled code. Hence, the time consuming simulation is avoided.

3. ESTIMATION TECHNIQUE

3.1 Estimation Overview

In our estimation technique, we convert the application to the intermediate format and use the intermediate code to guess the compiled code. We use the performance characteristics to obtain the estimates for the standard library function. We also estimate the IPC (instructions per cycle) of the application by analyzing the variation of IPC with average basic block length. We perform our estimation using an intermediate format which is low-level enough and still processor independent. We used the Low Level Virtual Machine (LLVM) format (Lattner and Adve, 2004). LLVM is a compilation framework for research in 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. LLVM incorporates a virtual machine which can run the LLVM object codes from which profile data can be easily extracted. Below we describe the methodology in detail:

We assume that the application is provided as a C/C++ program. This does not limit our methodology in any way. There are front-ends available to convert C/C++ programs into LLVM format. Front-ends for Java and Scheme are under development. The application is converted into the LLVM format using the gcc front-end available from the LLVM webpage (Lattner, 2004). The front-end converts the input programs in C/C++ into a bytecode file and after the initial intermediate code is passed through processor independent optimizations, the resultant intermediate format instructions are used to make an educated guess of the final machine code that would be produced. Many programming constructs were compiled, both till intermediate format and right up to the final executable format. The results were analyzed. The intermediate and the final object codes produced were compared. This helped us to form an idea of what kind of intermediate format results in what kind of object code. The process of estimation for different LLVM instruction is explained in the following sections.

3.2 Instruction Classification and Estimation

For our estimation purpose, we have classified the LLVM instructions into the following three categories.

1. *Simple Instructions:* Instructions which have a direct implementation in the target architectures instruction set.
2. *Compound Instructions:* Instructions which do not have an equivalent instructions in the target architectures instruction set and hence is broken down into simpler instructions from the target architecture's instruction set.
3. *Functions Calls:* These are instructions which invoke a function.

This classification of the LLVM instructions is for a particular target. The classification may be different for different targets. Simple instructions for the arm architecture are shown in Table 1. The LLVM instructions on the left hand side of the table can be implemented by the arm equivalent instruction on the right. Hence these LLVM instructions result in one instruction in the final object code after compilation.

To estimate the number of instructions generated for each LLVM in an application, the instruction is checked to determine its category. For simple instructions there are direct equivalents in the target instruction set and hence they produce one instruction per LLVM instruction in the compiled object code.

For the compound instructions, programs containing these instructions were compiled to the final object code and the resulting instructions from these LLVM instructions were studied. In the

LLVM instruction	ARM equivalent
br	b, bl
add	add, addc, qadd, qdadd
sub	sub, sbc, rsb, rsc, qsub, qdsub
mul	mul, mla, umull, umlal, smull, smlal
setcc	cmp, cmn
and	and
or	orr
xor	eor

Table 1: LLVM instructions and their ARM equivalent

```
switch(a){
       case 1: printf("1");break;
       case 2: printf("2"); break;
       default: printf("none");break;
}
return 0;
```

Figure 1: test program for estimating the switch instruction

```
switch uint %tmp.5, label %label.2 [
          uint 2, label %label.1
          uint 1, label %label.0 ]
label.0:
     ... %printf( ... )
     r et int 0
label.1:
     ... %printf( ... )
     ret int 0
label.2:
.    .. %printf( ... )
     ret int 0
```

Figure 2: LLVM intermediate code for the switch statement

following we demonstrate the procedure for the 'switch' LLVM instruction. The C program shown in Figure 1 was used as a test program (only relevant portion is shown):

The code shown in Figure 1, when compiled by the LLVM compiler resulted in the LLVM intermediate format shown in Figure 2 (only relevant portions are shown). The same code in ARM assembly is shown in Figure 3 (only relevant portions are shown). As can be seen from the assembly

```
    cmp r3, #1
    beq .L4
    ldr r3, [fp, #-28]
    cmp r3, #2
    beq .L5
    b .L6
.L4:
    ldr r0, .L8
    bl printf
    b .L3
.L5:
    ldr r0, .L8+4
    bl printf
    b .L3
.L6:
    ldr r0, .L8+8
    bl printf
.L3:
    mov r0, #0
    ldmea fp, {fp, sp, pc}
```

Figure 3: ARM assembly code for the switch statement

listings, one ‘`switch`’ statement in the LLVM format results in, as many ‘`cmp`’ instructions as there are ‘cases’ in the switch statement, twice as many ‘`branch`’ statements as there are ‘cases’ and one more ‘branch’ if there is a ‘default’ case. Hence a ‘switch’ statement in the LLVM format can be estimated to be equivalent to n_a instructions in the compiled code for ARM, where n_a is given by

$$n_a = 3 * n_c + d \tag{2}$$

where n_c is the number of cases in the ‘switch’ statement and d is *1* if there is a default case and *0* if there is no default case. Similar experiments were performed for the other compound instructions. Other 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 for each of the compound instructions.

To estimate function calls, the call instruction is estimated first. In the LLVM intermediate format we have the call instruction as

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

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

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

Now we can see that a call instruction results in a few `ldr` instructions (which move 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 `ldr` 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.

$$n = num\_args + 1 \tag{3}$$

where *num_args* is the number of arguments.

Now for internal functions, the estimation process is easy as the function code is available with the application itself and hence such functions can be estimated by the method already described. But the same cannot be done for external function calls which are basically calls to library functions as it can be a very time consuming process, and we would also need to get the source code of the whole standard library, which would have 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 in Section 3.3.

Figure 4 gives the flow for estimation of LLVM instructions for the ARM. The instruction is checked first if it is a simple instruction, i.e., those which have a direct equivalent in the target architecture, if so the estimate is one, for other instructions the model derived from our experiments are used as an estimate.

LLVM has a profiler, which can execute the bytecode file and can give the execution count of each basic block in the bytecode. After we have obtained the execution count of basic blocks, we use our previously described method to obtain the estimates for each basic block. The estimates for each basic block along with the profiler outputs is used to calculate the estimates for the execution of the whole application under a given set of inputs.

3.3 Estimation of Library Functions

Commonly used routines are provided as library functions which can be called from an application. This saves the developer's time as he/she does not need to write code for all these routines instead they can use previously tested implementations. The developer just inserts function calls whenever the services of such routines are required. The file is then compiled and linked with the library files so that those function calls are resolved correctly. For example for a call to the math library function sqrt(), the intermediate format inserts a call to the function. But the performance of a processor should also include the estimates for the standard library functions. Thus one way to obtain the estimates would be to compile all the standard library functions with LLVM and then use a similar technique as described above.

Occasionally these libraries are provided as binaries, with no source code given. As the source code is not available, we cannot use our methodology to compile the functions to intermediate format to obtain the estimates. In this section we describe our workaround for the performance estimation of standard library functions.

For the purpose of obtaining software estimates of standard library functions, we studied the GNU C Library (2005), as it is one of the most popular C libraries and it has been ported to a wide range of architectures. Also we had linked the benchmark programs with glibc. From the study of

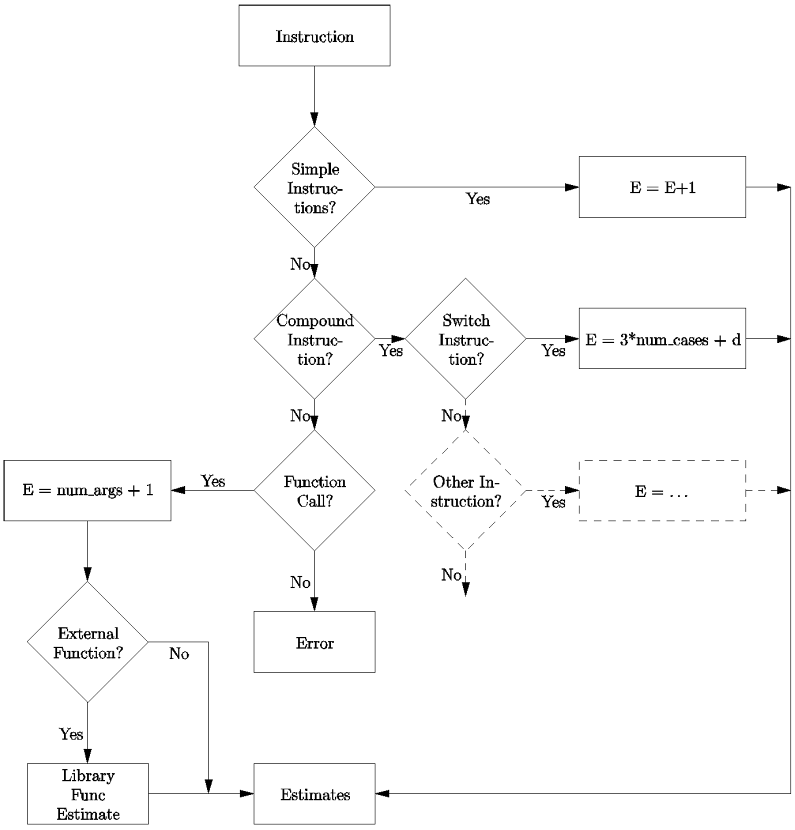


Figure 4: Estimation for LLVM instructions

the library functions, we have classified the standard library functions into two categories: 1) the ones whose performance does not depend on the input or input size and, 2) the ones whose performance depends on the input or input size. It is clear that the performance depends on the input. Hence, this classification is subjective in the sense that, the variation in performance with the input is small for some of the functions, while for others the variation is quite large.

We have found that functions like sqrt() belong to the first category, i.e., the runtime of the function sqrt() does not vary too much with the argument. On the other hand, functions like printf(), scanf() depend very much on the argument. This is clear from the fact that the time taken for the printf() function to finish execution will depend on the number of input parameters it is provided. In fact even when the number of inputs is constant, the execution time depends on the value to be printed. For example, if the application calls printf() with an integer argument, the time taken will be smaller if "1" has to be printed, compared to the time taken if "1000000" had to be printed. Hence, for functions belonging to the first category we model them using a fixed number of instructions being executed.

For the functions in the second category, we use a different technique. In the case of printf(), for example, the number of instructions executed would obviously be dependent on the number of characters printed, and we hypothesize that the relationship would be linear. As shown in our experimental results later this is quite true. So we modeled the function printf() using the following formula.

$$\boldsymbol{N = F + n * c} \tag{4}$$

In the above formula, N is the total cost of invoking the function, F represents the fixed cost of invoking the function, c represents the cost of printing a single character and n is the number of characters to be printed by the function call. Using the formula we arrived at the total cost of the function invocation. As we will show in our experimental results in the next section, this formula quite accurately models the function printf(). Similarly, we modeled other functions which are expected to show linear relationship, like scanf(), fprintf() etc.

Some functions in the standard library do not vary linearly with the argument size, e.g., functions like qsort(). The number of instructions executed when qsort() is called depends not only on the size of the argument (here it is the number of arguments passed to the function), but also on the relative ordering of the argument based on the sorting criteria. It has been shown that the running time of quick-sort is $O(n^2)$ and $\Omega(n \log n)$. But when we studied the qsort() implementation in glibc (GNUC Library, 2005), we found that the qsort() function in glibc is not a pure quick-sort implementation. Instead, it uses interesting modifications, like it uses insertion sort once the partition size gets smaller than a particular threshold value. It also uses the median-of-three to choose the pivot value; this reduces the probability of choosing a bad pivot value. This results in a runtime of $O(n)$ for almost all cases (QuickSort, 2005).

To obtain the number of instructions executed for standard library functions, test programs were written which invoke the standard library functions and then ran the program through an instruction set simulator. The SimpleScalar instruction set simulator was used to simulate the compiled programs and the count of number of instructions executed was obtained from the simulation results. We took care to write our test programs in such a way that the targeted library function gets invoked a large number of times. This is done so that the different instructions getting executed, which are not part of the standard library function code, e.g., the program startup instructions etc., gets amortized, resulting in more accurate readings. After the program is run on the instruction set simulator, we get the number of instructions that got executed from the simulation results. The process was repeated for the different standards library functions.

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. The file is later used to directly obtain the estimates for the given application. Till now we have only the static program estimates, which just tell us what the compiled code might look like. The actual performance will obviously depend on the input data. Hence, we need to run the code and obtain the execution data. The program was run with some input data and the program profile results were extracted. The program was executed under the LLVM environment.

Even this process is cumbersome considering the fact that there are many standard library functions that have to be considered. But fortunately there are many standard library functions which we can safely ignore. Some functions like fopen(), fclose(), are generally executed only at program start and program end. Similarly, functions like exit(), abort() can get executed only once and when the program exits. For non-trivial programs (i.e., programs which do not just open files and close them or which exits as soon as it starts) these functions have negligible impact on the overall program performance, and hence can be safely ignored.

3.4 Estimation of IPC

As mentioned earlier in our previous work, we have estimated the number of instructions executed. While it can give a rough indication of how much time a processor will take in executing a given application, the indications may not be always accurate. This is due to the differences in IPC (instructions per cycle).

Ideally, the IPC should be 1 for a RISC processor, as there should be one instruction getting executed every clock cycle due to the pipelining. But IPC is less than 1 mainly due to the following reasons, which leads to pipeline stalls:

- ***Cache misses:*** During a cache miss the pipeline has to be stalled until the missed data arrives.
- ***Control Hazards:*** Control hazards arise whenever any control instruction is executed and the wrong branch is taken leading to the pipeline needing to be flushed.
- ***Data Hazards:*** Data hazards arise when an instruction depends on the results of a previous instruction in the pipeline and is not available yet.

In this work we have not considered cache misses. In order to ignore the effect of cache size, we have used a large enough cache size, so that cache size ceases to be a limiting factor. The details of arriving at the cache size are described in Section 4.2.

The remaining factors which affect the IPC are control hazards and data hazards. In a pipelined execution, control breaks are the points where a pipeline needs to be flushed, in case a wrong branch is taken. Hence, applications with longer basic blocks are expected to have higher IPC, as the endpoints of basic blocks are the points at which control breaks occur. To incorporate the data dependency factors, in our estimation process, we wrote test programs of different average basic block lengths. One set was written in such a way as to minimize any chance of data hazards. Another set of test programs were written which are expected to result in pipeline stalls. The test programs were simulated on the SimpleScalar. The first set of programs has a higher IPC since it was written in a way which minimizes the chances of data hazard. The IPC values of the first set of programs will be the upper limit of IPC for the given average basic block length and the cache size. Similarly, the second set of programs will have a lower value of IPC, which we designate as the lower limit for IPC. We shall later see that this lower limit is not accurate. The test programs were written in high level language and the compiler was able to optimize the data dependencies. Hence, a few of the observed IPCs are less than the lower limit. The upper limit and lower limit of the IPCs for different basic block length was plotted. The average of the two limits was used as the estimate for the IPC.

The basic block length used for the different benchmarks was obtained by taking the weighted average of all the basic blocks in the program, the basic block lengths being weighted by their execution count. Equation 5 gives the formula how average basic block length was computed, where bb_len_{avg} is the average basic block length we are interested in, B is the number of basic blocks in the application, n_i is the number of times basic block i is executed and bb_len_i is the length of the basic block i.

$$bb\_len_{avg} = 1/B * \sum n_i * bb\_len_i \quad (5)$$

4. EXPERIMENTAL WORK

Our experimental results are based on the MiBench benchmark (MiBench version 1, 2005). 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 compiler pass to extract the application characteristics. The information extracted was the basic blocks and the intermediate format instructions of each of these basic blocks. The codes for each basic block were estimated and then the execution counts of each basic block were obtained after executing the bytecodes through the profiler.

The same benchmarks were then compiled for ARM and the PISA architectures by the gcc cross compiler, provided with the SimpleScalar tool set, to produce machine codes. The executables were then executed on the SimpleScalar instruction set simulator, to obtain the number of instructions executed. We compare our estimates with the SimpleScalar output. The error was calculated with respect to the SimpleScalar results.

4.1 Results for Library Functions

For estimation of library functions, we wrote test programs to find the number of instructions executed for the function sqrt() for different iterations. The results are shown in Figure 5. The number of iterations are plotted on the x-axis with a log scale as we have varied the number of iterations across a large range. As can be seen from the graph, the number of instructions executed during the program startup and exit gets amortized with the increase in the number of iterations. Hence, the average number of instructions executed settles down to a stable value, thus confirming our hypothesis that the number of instructions executed in a sqrt() call is independent of the argument. The final stable value of the number of instructions executed is used as our estimate for the function sqrt(). In our estimation process, we multiply this count by the number of times sqrt() invoked and add the product to our estimates.

Next we present our results for the estimation of printf(). We wrote test programs to print a varied number of characters to the standard output. The results from the experiments are shown in Figure 6. As can be seen, the relationship between the number of characters printed and the number of instructions executed is almost linear. The same can also be inferred from the reasoning that, the number of instructions executed when printf() is invoked is the sum of a fixed number of instructions executed during function set up and a variable number of instructions executed for printing of the actual input. The variable part should be directly proportional to the number of characters printed. Assuming linear dependence, we used curve fitting to obtain the linear formula for the number of instructions executed from the plot Figure 6. The formula obtained was used for estimation of the number of instructions executed when printf() is invoked. The results are shown in Table 2. The values in the second column were obtained from the test executions of printf() on

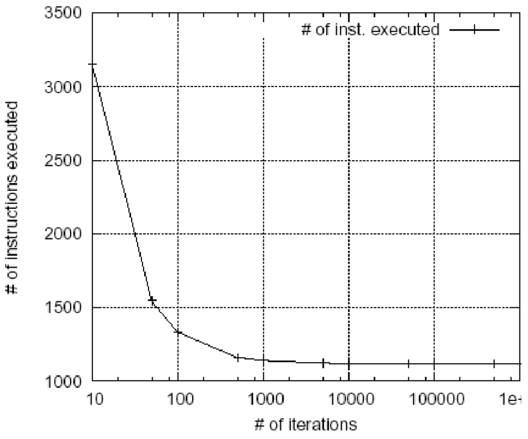


Figure 5: Number of instructions executed for sqrt() vs. number of iterations

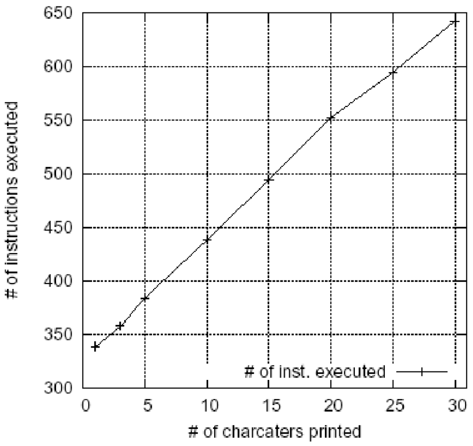


Figure 6: Number of instructions vs. number of characters printed for printf()

# of characters printed	# of instructions executed	#of instructions estimated	error%
1	339	342	-0.98
3	358	362	-1.39
5	384	383	0.1
10	438	435	0.64
15	494	486	1.46
20	552	538	2.46
25	594	590	0.67
30	642	641	0.06

Table 2: Estimation results for printf()

the instruction set simulator. The data in the second column corresponds to the graph shown in Figure 6. The estimated values, shown in the third column of Table 2, are the estimated values we obtained from our formula obtained from curve fitting of the simulated data of Figure 6. Table 2 also shows our estimates are within ±2.5% of the actual values.

We also wrote test programs to call qsort() with varying numbers of items to be sorted. The simulation results of qsort() execution for different input sizes are shown in Figure 7. As discussed earlier in Section 3.3, the qsort() implementation in glibc has linear performance for almost all cases. The same can be verified from the plot of number of instructions executed vs. the input size. We used curve fitting to arrive at the formula for qsort() estimates like we did for printf(). The results are shown in Table 3. The values in the second column were obtained from the test executions of qsort() on the instruction set simulator. The data in the second column corresponds to the graph shown in Figure 7. The values shown in the third column of Table 3 are the estimated values. Table 3 also shows that our estimates are approximately within ±2.5% of the actual values.

Figures 5, 6 and 7 and Tables 2, 3 shown in this section are for ARM. Similar experiments were performed for the PISA architecture. And the same estimation techniques were used for estimation of standard library functions in the case of PISA. Our estimation results for the different programs of the MiBench benchmark suite are shown in Tables 4 and 5 for ARM and PISA respectively. In Tables

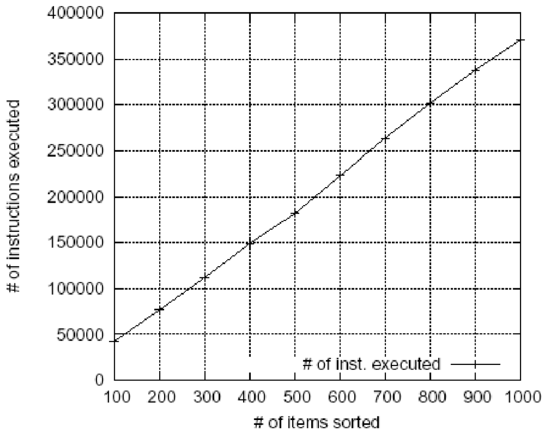


Figure 7: Number of instructions vs. number of items sorted

# of items sorted	# of instructions executed	#of instructions estimated	error%
100	43080	42000	2.51
200	77121	78000	-1.14
300	112170	114000	-1.63
400	149525	150000	-0.32
500	182133	186000	-2.12
600	222957	222000	0.43
700	263202	258000	1.98
800	301283	294000	2.42
900	337319	330000	2.17
1000	370175	366000	1.13

Table 3: qsort() estimates

4 and 5, the first column is the benchmark; the second column is the actual number of instructions executed which is obtained after simulating the benchmark on the instruction set simulator. The third column is our estimated value for the number of instructions executed and the fourth column gives the percentage error compared to the actual values. The error for ARM is about ±15% from the actual values, and for PISA it is about ±20% of the actual values. Errors for some of the benchmarks like `bitcount` are low as these programs do not have many external function calls. And most of the functions calls to standard library functions are calls to the functions of the first type, i.e., the ones for which the number of instructions executed does not vary much with the input.

4.2 Estimation for IPC

First we need to find the cache size to be used for our simulations. We simulated the benchmarks from the MiBench Benchmark suite for different cache sizes ranging from 1K to 128K. Figure 8 shows the variation of IPC with cache size for ARM. Figure 9, shows the same plot for the PISA architecture.

As expected, IPC increases initially as cache size increases. But beyond a certain size the plot flattens out, there is no or very little improvement in IPC if the cache size is increased. This can be

Benchmark	# of instructions simulated	# of instructions estimated	error (%)
basicmath_small	34337252	33807827	1.54
basicmath_large	153968114	147769221	4.02
bitcount_small	49671054	47736543	3.89
bitcount_large	743681421	714951488	3.86
qsort_small	43608758	39828189	8.67
qsort_large	737923418	648426723	12.13
crc_small	73981239	69812195	5.63
crc_large	1437936500	1357171331	5.61
susan_small	68075222	59736962	12.25
susan_large	1130126888	993681995	12.07
dijkstra_small	64930886	68714178	-5.82
dijkstra_large	272660486	264889316	2.85
blowfish_small_e	52415857	53195574	-1.49
blowfish_small_d	52408177	53195707	-1.5
blowfish_large_e	544060765	552636294	-1.58
blowfish_large_d	543979692	552636427	-1.59
sha_small	13544323	13195094	2.58
sha_large	140892866	137382880	2.49
stringsearch_small	161498	140741	12.85
stringsearch_large	3682705	3428762	6.9
rawcaudio_small	37695304	42476088	-12.68
rawdaudio_small	30162442	28819377	4.45
rawcaudio_large	732520508	825729166	-12.72
rawdaudio_large	586079072	568012320	3.08
patricia_small	103926712	92613136	10.89
patricia_large	640423162	539617650	15.74

Table 4: Estimates for ARM

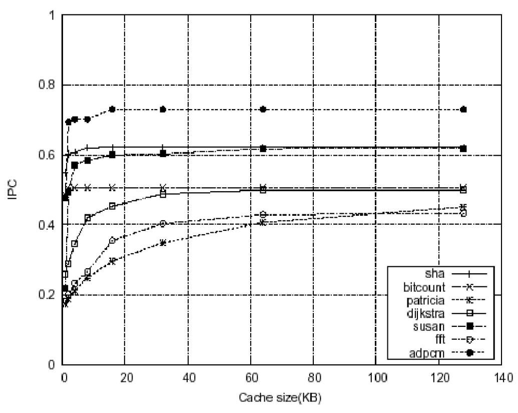


Figure 8: IPC vs. cache size for ARM

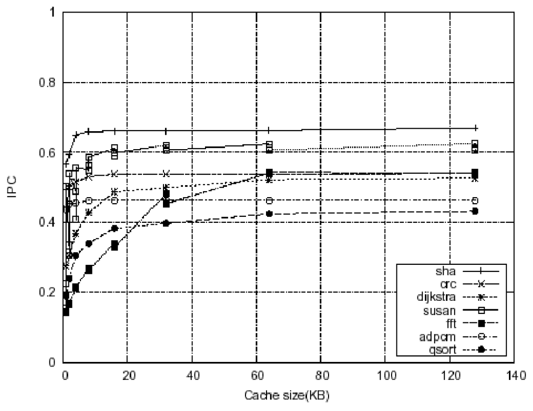


Figure 9: IPC vs. cache size for PISA

Benchmark	# of instructions simulated	# of instructions estimated	error (%)
bitcounts_small	43607079	44709924	-2.53
bitcounts_large	659000010	669387858	-1.58
susan_small	28655709	29371773	-2.5
susan_large	466581542	494538457	-5.99
stringsearch_small	186057	151779	18.42
stringsearch_large	4449396	3691922	17.02
blowfish_small_e	37373591	43725435	-17
blowfish_small_d	37054038	43725531	-18
blowfish_large_e	387245044	453773239	-17.18
blowfish_large_d	383916370	453773335	-18.2
sha_small	13200925	14661675	-11.07
sha_large	137290035	152652647	-11.19
qsort_small	41868586	31899122	23.81
qsort_large	568948084	637629717	-12.07
dijkstra_small	54879500	48900534	12.23
dijkstra_large	255787178	204229593	20.16
patricia_small	134125678	109499955	18.36
patricia_large	826922998	646257876	21.85

Table 5: Estimates for PISA

explained by the fact that, increasing the cache size beyond a certain value does not lead to much increase in the cache hit rate. As per the 90/10 rule, 90% of the execution time is spent in 10% of the code. Hence having a cache size of more than 10% of the code size should not lead to a much increase in hit rate. Thus, increasing the cache size beyond a certain value does not lead to much increase in IPC. From Figures 8 and 9, we can see that there is not much increase in IPC, for cache sizes above 64 KB. We have also plotted the miss rates for different cache sizes for the different benchmark programs. Figure 10 shows the miss rates for ARM. The miss rate is the ratio of number of cache misses and the total number of references. The same plot for PISA is shown in Figure 11. It can be seen that the cache miss rates are quite low for cache sizes larger than 64KB. Hence, for both ARM and the PISA architecture, beyond 64 KB the cache size ceases to be a limiting factor for the IPC.

Test programs were written for average basic block lengths varying from 4 to 25. One set of programs were written with no data dependency between the different instructions. The IPC of these programs was plotted against the average basic block length. The curve gives the upper limit of IPC for a given average basic block length. Similarly, the set of programs with data dependency gives the lower limit. Figure 12 shows the upper and lower limit on IPC for ARM and Figure 13 show the same for PISA architecture. In these graphs, we have also plotted the actual IPC for the different benchmark programs of MiBench against their basic block length. To obtain the average basic block length, the application was compiled using the LLVM compiler (Lattner and Adve, 2004). LLVM has a profiler, which was used to execute the compiled bytecode file and the execution count of each basic block in the bytecode was obtained from the profiler output. The average basic block length was calculated from the execution count of each basic block using Equation 5.

From Figures 12 and 13, we can see that some of the actual IPC values do not lie within the upper and lower limit. This is because the test programs were written in C, i.e., a high level language and the compiler can optimize away some of the data dependency inserted into the test programs.

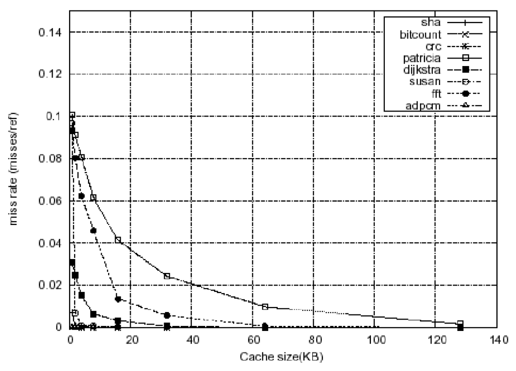


Figure 10: Cache miss rate vs. cache size for ARM

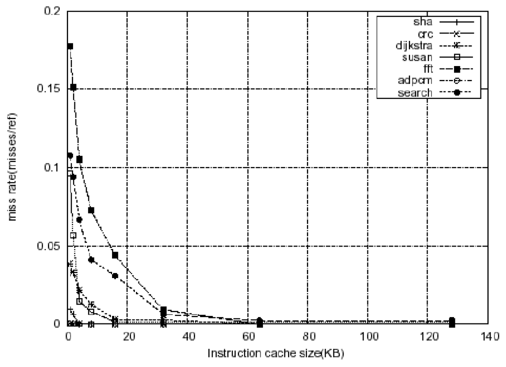


Figure 11: Cache miss rate vs. cache size for PISA

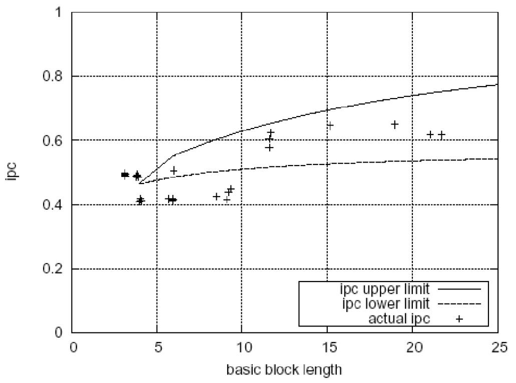


Figure 12: Upper and lower limits of IPC for ARM

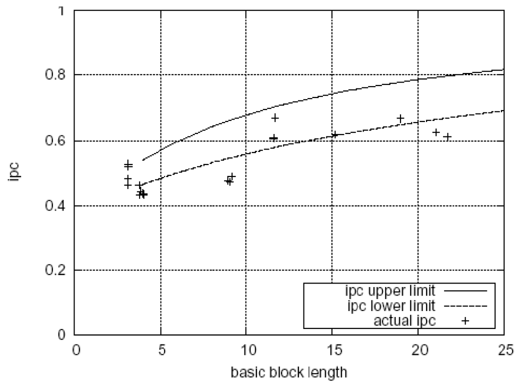


Figure 13: Upper and lower limits of IPC for PISA

So the actual lower limit may be slightly less than the one shown by the plot. This inaccuracy can be reduced if the test programs are written in the assembly language.

Next we use this upper and lower limit to obtain the IPC estimates for the different average basic block lengths. The estimates we use are the midpoint of the upper and lower limit. Figures 14 and 15 show the IPC estimates and the actual IPC for the MiBench Benchmark programs for the ARM and PISA architectures respectively.

The estimates were also compared with the actual IPC values after running the benchmark programs on the SimpleScalar simulator. The percentage errors of our estimates are given in Tables 6 and 7. For ARM, the maximum error is 18% and for PISA the maximum error is 22%.

As mentioned earlier, one reason for the errors was that we used a high level language for writing the test programs while obtaining the upper and lower limits of the IPC. Second reason for the errors in our estimation is that we have counted the basic block length from the intermediate format code. The actual basic block length might be slightly different in the machine level code. The effect of the first reason can be reduced by hand coding the test programs we used to calculate the upper and lower limits on the IPC, in assembly. For the second case, we cannot use assembly level code to obtain the basic block lengths, as it would go against our aim of not using compilation for the estimation process.

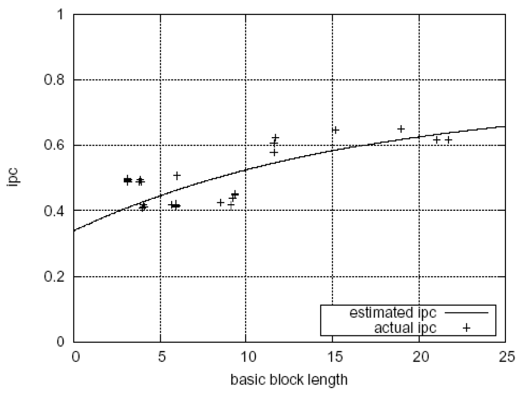


Figure 14: IPC estimates for ARM

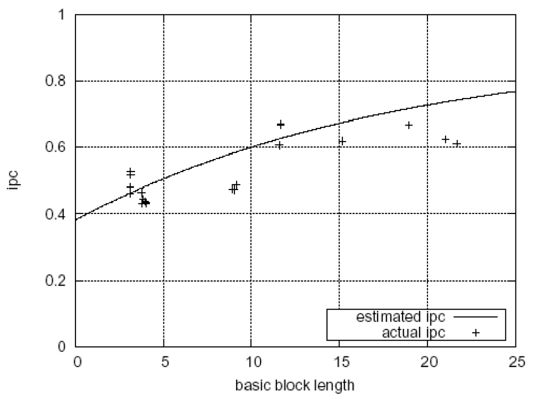


Figure 15: IPC estimates for PISA

Benchmark	Basic Block Length	IPC Simulated	IPC Estimated	Error (%)
rawcaudio_large	3.13	0.4972	0.4089	17.77
rawcaudio_small	3.13	0.4917	0.4089	16.85
dijkstra_large	3.15	0.4944	0.4093	17.22
dijkstra_small	3.15	0.4884	0.4093	16.2
rawdaudio_small	3.82	0.4872	0.4227	13.24
rawdaudio_large	3.84	0.4933	0.4231	14.23
qsort_large	3.89	0.4866	0.4241	12.85
qsort_small	4	0.409	0.4262	-4.21
search_large	4.06	0.4174	0.4274	-2.39
search_small	4.07	0.4112	0.4276	-3.98
basicmath_large	5.69	0.4174	0.4574	-9.59
patricia_small	5.91	0.412	0.4613	-11.96
basicmath_small	5.95	0.4193	0.462	-10.18
patricia_large	5.96	0.4153	0.4621	-11.28
bitcount_large	6.01	0.5063	0.463	8.55
bitcount_small	6.01	0.5061	0.463	8.52
sha_large	11.68	0.6234	0.5458	12.45
sha_small	11.68	0.6227	0.5458	12.36
susan_large_s	11.6	0.6057	0.5448	10.06
susan_small_s	11.6	0.5777	0.5448	5.7
susan_large_e	15.19	0.6463	0.5845	9.55
susan_small_e	21.7	0.6166	0.638	-3.48
susan_large_c	18.98	0.6493	0.6182	4.79
susan_small_c	21.05	0.6179	0.6336	-2.54

Table 6: IPC Estimates for ARM

Benchmark	Basic Block Length	IPC Simulated	IPC Estimated	Error (%)
dijkstra_large	3.1	0.5269	0.4633	12.06
dijkstra_small	3.1	0.5189	0.4633	10.71
qsort_large	3.8	0.4631	0.478	-3.21
qsort_small	4	0.4374	0.4831	-10.44
search_large	4.1	0.4354	0.4847	-11.31
search_small	4.1	0.4319	0.4847	-12.22
sha_large	11.7	0.6698	0.6269	6.41
sha_small	11.7	0.6688	0.6269	6.27
susan_large_c	19	0.6668	0.7172	-7.56
susan_large_e	15.2	0.6187	0.6747	-9.05
susan_large_s	11.6	0.607	0.6257	-3.07
susan_small_c	21	0.6243	0.7368	-18.01
susan_small_e	21.7	0.6113	0.7425	-21.46
susan_small_s	11.6	0.606	0.6257	-3.24

Table 7: IPC Estimates for PISA

5. CONCLUSION

In this paper we have presented a non-compilation based strategy for performance estimation of processors. Our work is based on estimating the machine code generated by analysis of the intermediate code. A new technique has been proposed and implemented for the estimation of library functions. Tables 2 and 3 shows that an error of less than ±2.5 can be achieved for library functions using the proposed technique. The estimation technique for library functions requires that the performance characteristics of each function must be obtained for each processor under consideration beforehand. However, this needs to be done only once for each processor. Accurate estimates for applications without any external function calls are possible as can be seen from Table 4 for the benchmark `bitcount`. Applications written in all the high level languages supported by the LLVM front-end can be estimated using the proposed framework. The proposed estimation framework provides for the effective selection of the most appropriate processor for a given application. It can also be combined with simulator based strategy to support fine grained processor selection.

ACKNOWLEDGEMENT

Part of the work titled "Practical Techniques for Performance Estimation of Processors" was presented at the 5th IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC), 2005, Canada.

REFERENCES

BAGHDADI, A., ZERGAINOH, N-E., CESRIO, W.O. and JERRAYA, A.A. (2002): Combining a performance estimation methodology with a hardware/software codesign flow supporting multiprocessor systems. *IEEE Trans. Softw. Eng.* 28(9): 822–831.

CHANDRA, S. and MOONA, R. (2000): Retargetable functional simulator using high level processor models. In *Proceedings of the 13th International Conference on VLSI Design*: 424–429.

FAUTH, A., PRAET, J.V. and FREERICKS, M. (1995): Describing instruction set processors using nML. In EDTC '95: Proceedings of the 1995 European conference on Design and Test, *IEEE Computer Society*, Washington, DC, USA: 503–507.

FERDINAND, C. (2004): Worst case execution time prediction by static program analysis. In *IPDPS: Proceedings of the 19th International Parallel and Distributed Processing Symposium*: 125.

GNU C LIBRARY. (2005): GNU C Library. URL: http://www.gnu.org/software/libc/libc.html

GUPTA, T.V.K., SHARMA, P., BALAKRISHNAN, M. and MALIK, S. (2000): Processor evaluation in an embedded systems design environment. In VLSID '00: Proceedings of the 13th International Conference on VLSI Design. *IEEE Computer Society*, Washington, DC, USA: 98–103.

HADJIYIANNIS, G., HANONO, S. and DEVADAS, S. (1997): ISDL: An instruction set description language for retargetability. In DAC '97: Proceedings of the 34th annual conference on Design automation, *ACM Press*, New York, NY, USA: 299–302.

LAJOLO, M., LAZARESCU, M. and SANGIOVANNI-VINCENTELLI, A. (1999): A compilation-based software estimation scheme for hardware/software co-simulation. In CODES '99: Proceedings of the seventh international workshop on Hardware/software codesign, *ACM Press*, New York, NY, USA: 85–89.

LATTNER, C. (2004): The LLVM compiler infrastructure project. URL: http://llvm.cs.uiuc.edu

LATTNER, C. and ADVE, V.S. (2004): LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the international symposium on Code generation and optimization*: 75–88.

LUNDQVIST, T. and STENSTROM, P. (1999): Timing anomalies in dynamically scheduled microprocessors. In RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium, *IEEE Computer Society*, Washington, DC, USA: 12.

MIBENCH VERSION 1.0 (2005): MiBench. URL: http://www.eecs.umich.edu/mibench

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

POLIS. (2005): A framework for hardware-software co-design of embedded systems. URL: http://embedded.eecs.berkeley.edu/Respep/Research/hsc/abstract.html

PUSCHNER, P. (2002): Is worst-case execution-time analysis a non-problem? – Towards new software and hardware architectures. In Proc. 2nd Euromicro International Workshop on WCET Analysis, *Technical Report*, Department of Computer Science, University of York, York YO10 5DD, United Kingdom.

QUICKSORT (2005): Inline qsort() implementation. URL: http://www.corpit.ru/mjt/qsort.html

RAJESH, V. and MOONA, R. (1999): Processor modeling for hardware software codesign. In VLSID '99: Proceedings of the 12th International Conference on VLSI Design – VLSI for the Information Appliance, *IEEE Computer Society*, Washington, DC, USA: 132–137.

STOLBERG, H-J., BEREKOVIC, M. and PIRSCH, P. (2002): A platform-independent methodology for performance estimation of streaming media applications. In *Proceedings 2002 IEEE International Conference on Multimedia and EXPO (ICME2002)*: 105–108.

SUZUKI, K. and SANGIOVANNI-VINCENTELLI, A. (1996): Efficient software performance estimation methods for hardware/software codesign. In DAC '96: Proceedings of the 33rd annual conference on Design automation, *ACM Press*, New York, NY, USA: 605–610.

BIOGRAPHICAL NOTES

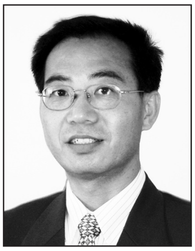
Abhijit Ray received his B Degree in electrical engineering from the Regional Engineering College, Kurukshetra, India in 1998. He received his PhD in computer engineering from Nanyang Technological University, Singapore. He is currently with Singapore Technologies Electronics (Training & Simulation) Pte. Ltd. His research interest include hardware-software co-design, micro-processors.

Abhijit Ray

Thambipillai Srikanthan has been with Nanyang Technological University (NTU), Singapore, since 1991, where he holds a joint appointment as professor and director of the Centre for High Performance Embedded Systems. He received his BSc (Hons) in Computer and Control Systems and PhD in System Modelling and Information Systems Engineering from Coventry University, United Kingdom. His research interests include system integration methodologies, architectural translations of compute intensive algorithms, high-speed techniques for image processing and dynamic routing. Dr Srikanthan has published more than 180 technical papers and has served a number of administrative roles during his academic career. He is the founder

Thambipillai Srikanthan

and Director of the Centre for High Performance Embedded Systems, which is now a University level research centre at NTU. He is a corporate member of the IEE and a senior member of the IEEE.


Jigang Wu

Jigan Wu received his BS degree in computational mathematics from Lanzhou University (China) in 1983. He received his PhD in computer software and theory from the University of Science and Technology of China. He was successively an assistant professor, lecturer in Lanzhou University from 1983 to 1993. He was an associate professor in Yantai University (China) from 1993 to 2000. He has been with Nanyang Technological University (NTU), Singapore, since 2000. He is currently a research fellow in Centre for High Performance Embedded Systems, NTU. Dr Wu has published more than 70 technical papers. His research interests include hardware/software co-design, reconfigurable computing and parallel computing.