Simulation-driven Software Performance Estimation for Fast Design Space Exploration

Seongnam Kwon
Hyunok Oh
Soonhoi Ha
The School of Electrical Engineering and Computer Science
Seoul National University
Seoul 151-742, KOREA
TEL : +82-28807292
{ksn, oho, sha}@iris.snu.ac.kr

ABSTRACT - In this paper, we propose a novel performance estimation method of software function blocks considering the effect of architecture variation, compiler optimization, and data dependent behavior. In the proposed design space exploration framework, a system behavior is specified as a composition of function blocks and the execution order of the blocks is known a priori. We run the entire application with code augmentation on the instruction set simulator of the target processor to obtain the accurate performance information of individual function blocks. In the design space exploration loop, the performance of the entire application is easily computed as a linear combination of function block performance values. Experimentation with real-life applications proves the viability and efficiency of the proposed technique.

I. Introduction

A critical step in a system level design procedure is to compare the estimated performance of a system behavior running on a number of different architectures. And there are wide performance variations for a given architecture depending on how to partition the system behavior into the architecture components. To explore the wide design space of architecture selection and partitioning early in the design cycle, fast and accurate performance estimation is essential, which is the main concern of this paper. Depending on the design objective, “performance” means execution time, power consumption, or another while we focus on execution time in this paper.

We assume that the system behavior is modeled as a composition of function blocks. Differentiation between function blocks and communications allows the system designer to explore the communication architectures independently of component selection. Also, the function blocks become the unit of partitioning to the processing elements in a given architecture.

The main theme of this paper is to estimate the performance of a function block on a processor and to store this information in a database. The usage of this information is two-fold. First, it is used for component selection and partitioning decision. Any optimal partitioning algorithm needs the execution time information of a function block on all candidate processing elements. And, the information should be accurate enough to make the partitioning decision reliable.

Second, once partitioning decision is made, we can accelerate the performance estimation of the entire application without running cycle accurate simulation of the blocks mapped to the processing elements. Instead, the system performance is estimated as a linear combination of block performances on the mapped components plus communication costs. Again, accurate performance estimation is crucial to make this approach viable.

Providing the accurate performance information of a function block on each processing element is a very time-consuming and difficult task, especially for the software side. If a function block is mapped to a predefined hardware IP, its performance is usually determined by the IP provider. Or, execution time of a hardware function module is usually predictable. On the other hand, there is no such hope for software function blocks. Software performance varies depending on the architectural features or on the compiler performance even for the same processor. Moreover, software performance is in general data-dependent. Thus it is not evident how to determine the representative performance value for a software block.

Therefore, there are two additional requirements on the performance estimation method besides the accuracy requirement for the proposed design space exploration scenario. First, it should be adaptable to reflect varying architecture features and compiler options. Second, the method should take into account the data-dependent performance variation.

Software performance estimation has been an important research topic in real-time system community ([1][2]). They mainly focused on the schedulability analysis of real time applications within hard real-time constraints. In their context, profiling and simulation methods are not adequate because simulation results can only cover part of the system behavior with no guarantee of worst case performance. On the other hand, we are more concerned about common-case performance. The other hand, we are more concerned about common-case performance. The proposed approach is applicable to soft real-time systems including most multi-media applications.

The proposed performance estimation method uses a cycle-accurate simulator at the level of instruction, or an instruction-set simulator (ISS). Since the simulator usually models all architecture features and estimation is made with the binary executable after compilation, the first requirement of adaptability is satisfied. The second requirement is met by simulating the entire application with real data, providing real data as the test vector for each block. Thus, we can obtain a real profile of data-dependent performance variation in a given application. Another benefit of running the entire application is to estimate the performance of multiple function blocks at once.

We run cycle-accurate simulation to determine the performance values of software function blocks and to record these values in a database. Since this task is accomplished before the design space exploration loop, only once for each processor core, long simulation time of the proposed method is tolerable.
The rest of this paper is organized as follows. Section 2 shows the workflow of the proposed design space exploration framework and the performance estimation method. Section 3 gives an overview of the related works on software estimation and compares them with the proposed technique. Detailed discussions on performance estimation of function blocks and design library construction follow in section 4 and 5 respectively. Section 6 discusses how to estimate the application performance during design space exploration. Some experimental results are shown in section 7. Section 8 concludes this paper.

II. Proposed Design Space Exploration Framework

The proposed design space exploration procedure, depicted in Figure 1(a), assumes that a system behavior is modeled as a composition of function blocks. We do not assume any specific computing model for block composition as long as the system behavior is defined by a non-preemptive execution sequence of function blocks. The only restriction is that the execution order of function blocks is known a priori. Dataflow specification for digital signal processing (DSP) applications, widely adopted in DSP design tools such as COSSAP and SPW, is a well-known example that meets the restriction.

In the proposed framework, the first step is to update the performance database of function blocks that are used in the application. From the functional specification of system behavior and the block performance database, the kernel of design space exploration (DSE) loop performs architecture selection and automatic partitioning. After mapping the system behavior to a candidate architecture template, the performance of the entire application is estimated. In this step, following the given execution order, we just sum up the performance values of function blocks, which is extremely efficient. We repeat this process exploring system architecture candidates and partitioning candidates.

Figure 1(b) shows the procedure of simulation-based performance estimation of function blocks. For estimating the performance of each block, a common method is to build a test bench program where a test vector generator provides input argument values to the function. This method has two serious drawbacks. First, it is very laborious to build a separate test bench program and analysis environment for each function block. Second, good test vectors are not easy to define.

The proposed approach overcomes these drawbacks by simulating the entire application. Since the entire application is already given at the specification stage, no additional effort of building a separate simulation environment is needed. And test vectors to the function blocks are all real, better than other synthetic test vectors. Before simulation, we automatically augment the code to identify the start and the end of each function execution. The proposed augmentation technique is explained later. The augmented code is cross-compiled for the target processor with all optimization options turned on.

The compiled code is simulated with the ISS of the target processor. We assume that all architectural features of the processor are accurately modeled in the simulator. With the augmented code, we obtain the run time profile of each function blocks from the simulation. The run time profile includes the execution time and memory access counts.

From the run time profile, we determine the representative performance values and store it to the database. The representative performance values can be the worst case performance or the average case performance. Since we are using the real test vectors for a function block, the average case performance value is meaningful when computing the average performance of the entire application by summing up the performance values of function blocks. Note that there is no guarantee of the worst case performance because we use the real test vectors that are not exhaustive in any sense. It just says that the performance is no worse than this value with high probability.

Now, we summarize the contributions of the proposed approach as follows.

- It provides the system designer with a convenient mechanism to update the estimated software performance of function blocks as the system architecture varies without the aid of block creators. Thus, the system designer can evaluate the system architectures with respect to their effects on the software performance.
- It considers the SW performance variation on compiler optimization and data dependency and gives an accurate estimate of each function block by simulating the executable code of application after compiler optimization using an ISS.
- It allows the system designer to accurately estimate the SW performance on a processor by simply summing up the performance values of the mapped function blocks. It results in a fast design space exploration.
III. Related Works

A common practice of software performance estimation in industrial design is simulation on an ISS [3]. Even though the simulation method is most accurate, it is often too slow to be used inside the design space exploration loop. The proposed approach also simulates the software implementation of the target system behavior. But, we do not estimate the performance of the entire application. On the other hand, we use the simulation to estimate the performance of function blocks before the design space exploration loop.

Another approach of software performance estimation is compiled simulation. A compiled simulation method annotates estimated timing to the original C code [4] or an “assembler-level”, functionally equivalent, C code [5]. While the former only guesses the effect of compiler optimization, the latter annotates the timing information after compilation, so independently of compiler optimization. Since performance estimation is performed by code execution, it is very fast. The main difficulty of this approach is to estimate the effect of processor microarchitecture, such as pipelines, multiple instruction issues and caches, on the approach is to estimate the effect of processor microarchitecture, by code execution, it is very fast. The main difficulty of this compiler optimization. Since performance estimation is performed independently of compiler optimization, the latter annotates the timing information after compilation, so independently of compiler optimization. Since performance estimation is performed by code execution, it is very fast. The main difficulty of this approach is to estimate the effect of processor microarchitecture, such as pipelines, multiple instruction issues and caches, on the performance estimation of software function blocks on processing elements is somehow given a priori. For a hardware block written in an RTL level specification, the performance value is obtained easily from the synthesis step for design validation of the function block. And, the performance value is usually stable and predictable. Therefore, hardware IPs are assumed provided with their performance values.

On the other hand, if a software function block is defined as a C function, as most system-level design environments assume, the execution time of the block on a processor heavily depends on the compiler performance. Depending on the compilation options, the performance variation can be as large as 100%. Even though block performances were already recorded in the database, we have to examine which compiler and what options were used before using those performance values. If a better compiler is developed, old performance values become obsolete and new performance estimation should be done. Therefore, the performance of a software IP cannot be determined a priori in general and the system designer should have a tool to estimate the performance on all candidate processors without the aid of the IP provider.

As discussed in section 2, the proposed framework solves these problems using ISS-based estimation. We estimate the performance after cross-compiling the C code to the target processor with the same compiler option as used in the synthesis step. And, we update the database if necessary. We have different database entries for different compiler options, even for the same block on the same processor.

Architecture features as well as compiler options affect the software performance. The most important architecture feature is memory system. If a cache system is used, cache miss rate and miss penalty both affect the software performance. As a result, there is a cyclic dependency between the performance estimation step of function blocks and the design space exploration step. The system architecture is determined from the DSE step that is performed after the performance estimation step. And, the accurate performance estimation is only possible after system architecture is determined: memory access time is dependent on the communication architecture and memory system. The architecture is determined: memory access time is dependent on the communication architecture and memory system.

Another factor to affect the software performance is data itself. If an average case behavior is of main interest, how to provide the realistic test vector to the function block is also an issue for accurate performance estimation. The proposed technique solves this problem by simulating the whole application to obtain the block performances. Then, the test vectors to a function block are real data in a specific application of interest.

To obtain the block performances from the entire application, we use the “break-point” or an equivalent command of an ISS. We set up the break points at the beginning and at the end of a block execution. At each break point, we print the time information in processor cycles into a trace file. From the trace information, we can obtain the performance statistics of a function block.
Figure 2. Code augmentation for performance estimation: (a) first trial and (b) the proposed code structure.

void Start_func(); {return;}
void End_func() {return;}
...
Start_func(); {
/* the function block */
}
End_func();
...
extern int Start_func();
extern void End_func();
...
If(Start_func()==true) {
/* the function block */
}
End_func();
...
(void)

(a) (b)

Figure 2(a) shows a naive approach to set up the break points. Two function calls, “Start_func” and “End_func”, are augmented at the boundary of a function block and set up the break points on these two functions.

Unfortunately, this naive approach does not achieve the goal in general. Any serious optimizing compiler removes the End_func() call since it does nothing. And it may change the execution order of Start_func() call and the block execution. Then, we cannot identify the correct execution time of a function block from the trace information. From this observation, a new problem appears: how to set up break points at the function boundaries regardless of which compiler option is used.

A key requirement is to keep an optimizing compiler from reordering the instructions across the block boundary. It means that there should be an ordering relation between the augmented function calls and the function block. And, the compiler may not remove the augmented function calls at will. The proposed approach satisfies these two requirements. Figure 2(b) shows the augmented code structure. To make an ordering relation between the Start_func() call and the block body, we bind the block definition inside the conditional construct and make the Start_func() return the conditional variable. Such control dependency prevents most compilers from moving the code of the function block outside the conditional construct. This is a major source of estimation error of the proposed approach. But, it is confirmed by experiments that there would be a negligible portion of the code unless the block size is too small (a few tens of assembly instructions).

Surely, the Start_func() always returns TRUE. This fact, however, should not be revealed to the compiler at the compilation step. To this end, we put the function definitions of Start_func() and End_func() outside of the application by using “extern” statement. Unless further optimization is performed in the linking step, this scheme of object code separation achieves the goal: compiler does not eliminate the function calls.

From the proposed code augmentation, we can obtain the timing traces at the block boundaries and compute the execution time of the block by a simple calculus. For each block, we compute the time difference between two break points, Start_func() and End_func() calls. And, we subtract the fixed overhead of the augmented function calls, which is pre-computed. Between two break points we also measure the memory access counts outside the processor by simulating on-chip cache behavior in ISS.

In summary, the proposed technique of block performance estimation has the following characteristics.

1. The performance value of a software block on a processor is given as a pair: (CPU time, memory access counts).

2. The proposed technique simulates the whole application to obtain the block performances with real, not synthetic, test vectors.

3. The proposed code augmentation scheme confines the compiler optimization to be applied within the function block. So, a simple technique of obtaining timing traces at the break points as well as memory access traces can be used to obtain the accurate block performance information.

V. Database Update

The estimated performance information of a function block is recorded in the database. As discussed in the previous section, there are multiple entries for each (block, processor) pair depending on the compiler options. Also, we need to distinguish the block performances by block parameters. For example, the execution time of an FIR filter is proportional to the number of filter taps. Different sets of filter coefficients are likely to be defined as block parameters while the same block definition is used. A block parameter that has an effect on the block performance is called a factor of the block. The tap count of an FIR filter is a factor for example. Then, a block has multiple database entries depending on the factor values.

Even with a given factor and a given compiler option, a function block may have a different performance value at each execution because its performance is data dependent. For data dependent performance variations, we compute the worst case performance and the average case performance and record them in the database by default. Other statistical information such as confidence interval and distribution can be obtained from the timing traces and added to the database if necessary.

To summarize, in the proposed framework, the following tuple is updated to the database automatically after block performance estimation.

(block name, factor name and values, processor, compiler options, memory access counts, worst-case performance number, average-case performance number, optional for future extension)

Up to now, we explain how to add a new block performance into the database. In case the performance information of a function block is recorded in the database already, there is a trade-off whether we estimate the block performance again for a new application or not. The performance of a block may depend on what application it is used in and what are the input value ranges.

Then, estimating the block performance again with a new application gives more accurate information for the next design space exploration step. However it costs time overhead of performing simulation with an instruction set simulation for each candidate processor. If the number of candidate processors is large, this overhead may be too huge to be tolerated within the tight budget of design time. Otherwise, it is worth paying for more accurate performance estimation.

VI. Application Performance Estimation

During the design space exploration step, fast and accurate performance estimation of the entire application is crucial for the validity of the design environment. In the proposed framework, we estimate the application performance as a linear combination of block performances for each candidate system architecture and mapping decision.

For a function block Bk, let P(k,i), m(k), and c(k,l) be the estimated CPU time on the mapped component P, the memory
access counts, and the communication requirements to the next block $B_i$ respectively. And let $n(k)$ be the number of invocations of block $B_i$ in a given execution scenario of the entire application. Then, the estimated performance becomes

$$P_{est} = \sum_{B_i \in CP} n(k) \times (P(k,i) + m(k) \times n_m + c(k,i) \times n_c)$$ (1)

where $n_m$ and $n_c$ represent the memory access overhead and the channel communication overhead of the selected candidate architecture. The summation is performed for the blocks on the critical path.

The accuracy of the estimated performance mainly depends on the accuracy of each term. For the block performance $P(k,i)$, the estimated value in the database is slightly larger than the actual value because compiler optimization is performed across the block boundary when the entire application is compiled. As the block granularity increases, such an error becomes negligible as our experimental results confirms in the next section. Another cause of inaccuracy may come from the cache behavior. When the initial state of cache is different, the simulated cache behavior is also different, to make the performance estimation inaccurate. It also affects the number of memory access counts.

For the second term of the memory access overheads, we have to differentiate memory access counts depending on the address spaces in case different memories are used. It means that the memory access count information of a block in the database needs to be more divided depending on which types of memory accesses are performed: for example, instruction, constant data, heap, and stack.

The third term may be included in the second term if the communication is performed through memory and asynchronous protocol is used. Otherwise, we need to pay extra overhead of synchronization and/or communication activities. Accuracy is also dependent on the modeling of the candidate architecture.

VII. Experiments

In this section, we show experimental results with two real-life examples, an H.263 encoder and an H.263 decoder, and a toy butterfly example. As shown in Figure 3, we specify the H.263 encoder application with a composition of software function blocks. Following the proposed design flow, we first estimate the performance of each function block. As a target processor, we use ARM720T processor with 8Kbytes write-through cache inside. And we use ARM C compiler (armcc) with –O2 option for the performance of each function block. As a target processor, we use ARMulator[8] for the processor ISS and obtained the performance of the CPU times become the performance of function blocks. The last row includes the linear combination of block performance whose value is 1,244,200,366 (cycles). We also simulated the entire application without code augmentation and obtained the total execution time of 1,243,538,245 cycles. Note that the error between two results is just 0.05%. It proves that the proposed technique of code augmentation is valid even under severe compiler optimization. The small amount of error is attributed to the fact that the proposed technique ignores inter-block compiler optimization.

Table 1. Estimated performance of H.263 encoder blocks

(a) Execution time without cache miss penalty

<table>
<thead>
<tr>
<th>Block name</th>
<th>Count</th>
<th>WCET</th>
<th>Total Execution time (TET)</th>
<th>WCET/Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLC</td>
<td>50</td>
<td>485795</td>
<td>21904583</td>
<td>1.11</td>
</tr>
<tr>
<td>MC</td>
<td>50</td>
<td>2608646</td>
<td>48842573</td>
<td>2.67</td>
</tr>
<tr>
<td>IDCT</td>
<td>29700</td>
<td>1947</td>
<td>13122365</td>
<td>4.41</td>
</tr>
<tr>
<td>Quantization</td>
<td>29700</td>
<td>1366</td>
<td>35753978</td>
<td>1.14</td>
</tr>
<tr>
<td>DCT</td>
<td>29700</td>
<td>2841</td>
<td>69553894</td>
<td>1.21</td>
</tr>
<tr>
<td>ME</td>
<td>50</td>
<td>24777951</td>
<td>986623904</td>
<td>1.26</td>
</tr>
<tr>
<td>Et.</td>
<td></td>
<td>68412069</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>50</td>
<td>1244200366</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1(b) indicates the total number of memory access counts.

(b) the total number of memory access count

<table>
<thead>
<tr>
<th>Block name</th>
<th>Count</th>
<th>Seq. Read(SR)</th>
<th>Nonseq. Read(NR)</th>
<th>Seq. Write(SW)</th>
<th>Nonseq. Write(NW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLC</td>
<td>50</td>
<td>2827615</td>
<td>942545</td>
<td>3269792</td>
<td>8607985</td>
</tr>
<tr>
<td>MC</td>
<td>50</td>
<td>2251771</td>
<td>750599</td>
<td>252300</td>
<td>1997360</td>
</tr>
<tr>
<td>IDCT</td>
<td>29700</td>
<td>732818</td>
<td>244383</td>
<td>765380</td>
<td>3067891</td>
</tr>
<tr>
<td>Quantization</td>
<td>29700</td>
<td>510153</td>
<td>170379</td>
<td>207900</td>
<td>2009181</td>
</tr>
<tr>
<td>DCT</td>
<td>29700</td>
<td>2535014</td>
<td>845800</td>
<td>237600</td>
<td>17160128</td>
</tr>
<tr>
<td>ME</td>
<td>50</td>
<td>5999358</td>
<td>1999826</td>
<td>9622420</td>
<td>13946882</td>
</tr>
<tr>
<td>Et.</td>
<td></td>
<td>3749351</td>
<td>123701</td>
<td>756487</td>
<td>10736888</td>
</tr>
<tr>
<td>Total</td>
<td>50</td>
<td>1860660</td>
<td>620599</td>
<td>15113899</td>
<td>57545865</td>
</tr>
</tbody>
</table>

If we consider the cache miss penalty, the accuracy of the proposed technique is slightly degraded. Table 1(b) indicates the total number of memory access counts that correspond to cache misses. We recorded separate counts for sequential/non-sequential(S/N) read/write(R/W) memory accesses. Figure 4 illustrates the error of the estimated performance obtained from equation (1) compared with the simulation result considering the cache miss penalty. Since all function blocks are executed in a single processor, there is no communication overhead included in this experiment. We used two different sets of cache miss penalties as depicted in the Figure 4. As larger cache miss penalty, the error grows as expected. The error is mainly caused by the inaccuracy in the estimated memory access counts as summarized in Table 2. The augmented code spoils the cache states before
block execution, so increases the cache miss counts. If the block size is large, such effect is negligible. In the H.263 encoder application, the total error is still under 0.5%.

Figure 4 also shows the experimental results with other image samples. Here, we estimate the block performances separately for each image sample since the performance values are quite different depending on the scene characteristics. It confirms that the software block performance is very data-dependent. In real system design, we should use test inputs that will result in worst case execution times.

Table 2. Error of the estimated memory access count compared with the simulated value with various image samples.

<table>
<thead>
<tr>
<th>unit : %</th>
<th>FOREMAN</th>
<th>AKIYO</th>
<th>MAD</th>
<th>GARDEN</th>
<th>SUZIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq. read</td>
<td>2.03</td>
<td>1.44</td>
<td>1.60</td>
<td>1.50</td>
<td>2.32</td>
</tr>
<tr>
<td>Nonseq. read</td>
<td>2.08</td>
<td>1.54</td>
<td>1.72</td>
<td>1.51</td>
<td>2.40</td>
</tr>
</tbody>
</table>

To examine the effect of block execution order, we change the execution order of function blocks. We will have different cache behavior, resulting in different memory access counts. Table 3 summarizes the experimental results. Compared with the original schedule, the cache hit ratio increases and therefore memory access counts are also reduced in the modified schedule. This experiment reveals that the software performance varies also on the schedule of block execution.

Table 3. Performance for a different execution sequences of blocks: S1 is an original schedule and S2 is a modified schedule.

<table>
<thead>
<tr>
<th></th>
<th>TET</th>
<th>SR</th>
<th>NR</th>
<th>SW</th>
<th>NW</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2 / S1</td>
<td>1.00</td>
<td>0.246</td>
<td>0.246</td>
<td>0.062</td>
<td>0.106</td>
</tr>
</tbody>
</table>

The experimental results with other two examples: an H.263 decoder and “butterfly” are summarized in Table 4. In particular, the “butterfly” demo consists of tiny blocks of fine granularity. As the block granularity decreases, the estimation error increases up to 0.65%. However, such limitation is insignificant, we believe, because the most time consuming function block is likely to be coarse grained. This preliminary experiment proves that the proposed performance estimation technique is very accurate and viable for fast design space exploration.

Table 4. Error of the estimated performance with H.263 decoder and “butterfly” applications.

<table>
<thead>
<tr>
<th>Unit : %</th>
<th>TET</th>
<th>SR</th>
<th>NR</th>
<th>SW</th>
<th>NW</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.263 Decoder</td>
<td>0.14</td>
<td>0.28</td>
<td>0.24</td>
<td>0.05</td>
<td>0.20</td>
</tr>
<tr>
<td>Butterfly</td>
<td>0.65</td>
<td>1.25</td>
<td>1.17</td>
<td>0.19</td>
<td>1.57</td>
</tr>
</tbody>
</table>

VIII. Conclusion

In the proposed design space exploration framework, it is critical to estimate the performance of functional blocks accurately before entering into the main loop of architecture selection and mapping of function blocks to the processing elements. We propose an accurate software performance estimation method based on instruction set simulation of processors. It is adaptable to consider varying architecture features and compiler performance. Since the proposed method runs the entire application, it also takes into account data-dependent performance variation of functional blocks for a given application.

Experimentation with real-life examples proves that the proposed method satisfies the requirements of accuracy and adaptability at the same time. Since the block performances are accurately estimated, the experimental results also confirm that a simple linear combination of performance numbers gives an accurate (within 1%) performance estimate of the entire application during the design space exploration loop.

REFERENCES