Software Estimation Using A Generic-Processor Model

Jie Gong, Daniel D. Gajski and Sanjiv Narayan
Department of Information and Computer Science
University of California, Irvine, CA, 92717, USA

Abstract

Previous work in hardware-software co-design has addressed issues in system modeling, partitioning, and mixed module simulation and integration. Software estimation, which provides software metrics to assist hardware-software partitioning, has not been extensively studied. In this paper we present a generic-processor model for estimating execution time and program-memory size of a specification to be executed on a given processor. Experiments have shown that our estimator is fairly accurate and fast when applied to a wide variety of designs.

1 Introduction

System-level design is a set of tasks which convert a system-level specification into a set of interconnected modules implementing the specification. Each module could be implemented in hardware or as software executing on a processor. A hardware implementation has better performance whereas a software implementation has lower cost, shorter development time and allows changes late in the design cycle.

Two opposite approaches have been proposed for hardware-software partitioning to determine which part of a specification should be implemented in software and which in hardware. Gupta and De Micheli [1] use a partitioning algorithm that starts with an initial partition where all operations, except for the unbounded delay operations, are assigned to hardware. The partition is refined by migrating operations from hardware to software in search for a lower cost feasible partition. The approach used by Ernst and Henkel [2] starts with a complete software implementation from which those portions that violate the performance constraints are extracted for the hardware implementation. These two approaches start from different directions but work towards the same goal of minimizing the amount of application-specific hardware required. Hardware-software partitioning requires a software estimator that will predict the execution time of the software implementation in order to identify which portions of the specification can be migrated from hardware to software while not violating the constraints or which portions need to be implemented in hardware to satisfy the timing constraints.

The most accurate method to obtain the execution time of a software implementation would be to compile the given specification to each target processor and measure it both statically [3] and dynamically [4]. However, such a processor-specific approach is very time-consuming and requires extensive infrastructure such as various compilers, estimators and target processors, which is often not available. Due to the fact that few systems will spend a huge amount of money to acquire various compilers and target processors just to determine the software performance as well as the fact that the processor-specific approach is too time-consuming to allow designers to consider more than a few design alternatives, it is indispensable to generate reasonably accurate estimates using a cheaper and faster approach.

In this paper, we propose a generic-processor model which generates reasonably accurate software estimates in a cheaper and faster way. In contrast to using different compilers and estimators for different target processors in the processor-specific approach, the generic model proposed in our approach will use only one compiler and one estimator in conjunction with different processor technology files which characterize the corresponding processors' instruction sets. This makes our estimator fast and easy to extend to different target processors. Besides the performance, our estimator also generates estimates for program-memory size for a given specification and a given target processor. The input to our software estimator is a system-level specification in SpecChart [5]. The target processors we consider in this paper are those used mostly in the embedded systems. Processors with cache memory and/or instruction pipelining are beyond the scope of this paper.

In the next section, we present the underlying model used for software estimation. Performance and memory size estimation for system-level specifications are discussed in Section 3 and Section 4 respectively. The results of our experiments are presented in Section 5 followed by conclusions and future work in Section 6.

2 Model for Estimation

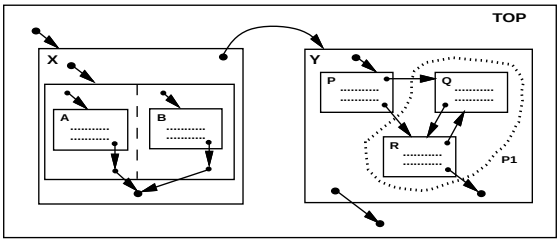

Figure 1: A sample system-level specification.

The estimation model we propose is targeted towards system-level specifications which consist of hierarchical concurrent/sequential behaviors. A behavior, which is a set of actions and a set of conditions describing when each

action is to occur, can in turn contain sequential or concurrent sub-behaviors. For example, in Figure 1, behavior TOP consists of two sequential sub-behaviors X and Y. Behavior X in turn contains two concurrent sub-behaviors A and B. Behavior Y contains three sequential sub-behaviors P, Q, R. Concurrency is represented by the dashed line like the one between behavior A and B whereas sequencing is represented by transition arcs. A behavior which does not contain any sub-behaviors is called a leaf behavior. In SpecCharts, each leaf behavior consists of a set of VHDL sequential statements. Behavior A, B, P, Q and R in Figure 1 are leaf behaviors. Each behavior in SpecCharts has a *start dot* indicating the starting point of the behavior. The *start dot* of a leaf behavior is the first statement in that behavior. A behavior is said to have completed when control reaches a *stop dot*. Behaviors without *stop dots* will never finish executing. Our estimator is intended to estimate the software metrics for any given leaf/non-leaf behavior of the specification as well as any given partition (a set of behaviors) in the specification. P1 in Figure 1 is a partition which contains two behaviors Q and R.

2.1 Estimation Model for Leaf Behaviors

In order to obtain the estimates for leaf behaviors, we may need to compile the code in the leaf behaviors into the instruction set of the target processor. For example, if a leaf behavior will be implemented on an Intel 8086 processor, it may need to be compiled into the 8086 instruction set. Using the timing and size information associated with each type of instruction such as how many clock cycles each 8086 instruction executes and how many bytes it takes, we can obtain the performance and program size of the behavior. Similarly, if the leaf behavior is going to be implemented on a Motorola 68000 processor, it may need to be compiled into the 68000 instruction set. Based on the 68000 instruction timing and size information, the estimator can obtain the software metrics for the behavior. We call this model the processor-specific model shown in Figure 2(a).

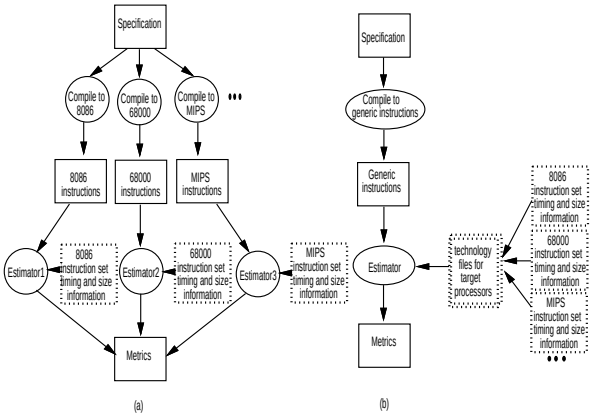

Figure 2: Approaches to software estimation: (a) processor-specific model, (b) generic-processor model.

Instead of using different compilers and estimators for different target processors in the processor-specific model, we propose a generic-processor model (Figure 2(b)) in which the leaf behavior specification is converted into a set of generic three-address instructions described in [6]. After that the estimator computes the software metrics for the leaf behavior based on the generic instructions and the technology files for the target processors. For example, if the leaf behavior is going to be implemented on an Intel 80286 processor, then the technology file for the 80286 processor will be used during the estimation. The technology file for a target processor supplies information about how many clock cycles and bytes each type of generic instruction requires for that processor.

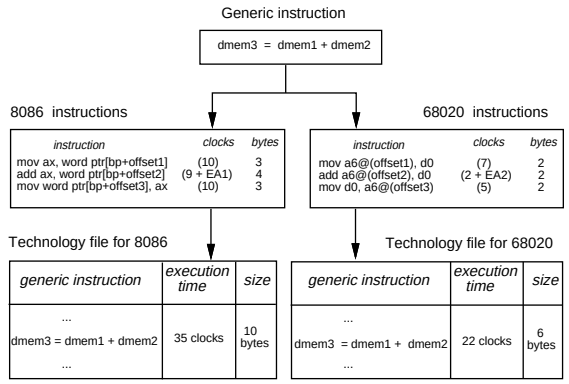

Figure 3: Deriving technology files for generic instructions.

The technology file for each target processor can be derived from the timing and size information of the processor's instruction set. Figure 3 shows the computation of the number of clock cycles for a generic instruction of the type $< dmem3 \leftarrow dmem1 + dmem2 >$. Here, $dmem$ indicates a direct memory addressing mode. The generic instruction is first mapped to a sequence of target processor instructions following which the total number of clock cycles of the generic instruction is obtained by summing the clock cycles of each individual instruction in the sequence. $EA1$ and $EA2$ in Figure 3 are the effective address calculation times used for displacement memory addressing mode, which are 6 and 8 clock cycles on the 8086 and 68020 respectively. Thus, the generic instruction will take 35 and 22 clock cycles on the 8086 and 68020 processors respectively. Using a similar approach we can derive the number of bytes each type of generic instruction will take if it is executed on the 8086 or 68020 processor. Presently the technology files for 8086, 80286, 68000 and 68020 processors are supported in our estimator. The 8086, 80286, 68000 and 68020 technology files are derived from the timing and size information of their corresponding instruction sets given in [7, 8, 9, 10]. All technology files can be found in the appendix of [6].

Compared with the processor-specific model, the generic model has several advantages. First, the generic model does not require different compilers and estimators for different target processors. Instead, only a single compiler, estimator and a set of technology files is required for software estimation. Second, the generic model makes retargeting the estimator to a new processor much easier. Retargeting consists of providing a technology file for the new processor. In the processor-specific model, we would require a compiler for the new processor in addition to the timing and size information of the processor's instruction set. Therefore, the generic model is more flexible and

can be extended to perform estimation for new processors/microcontrollers even if compilers are not available on systems running the estimators. In other words, the estimators based on the generic model are more portable since they can run on any machine, not just those with necessary compilers. Finally, it is much faster to compile the specification into the generic instruction set than those associated with specific processors since the generic three-address instructions are free of instruction idiosyncrasies. A disadvantage of the generic model is the lower accuracy of its estimates largely because the generic instruction set represents only a portion of the processor's entire instruction set.

2.2 Estimation Model for Non-leaf Behaviors and Partitions

To evaluate the software implementation of a given non-leaf behavior or a partition on a specific processor, we must first flatten the hierarchy and sequentialize the specification to diminish the concurrency since our target machine is a uni-processor. In other words, the specification needs to be mapped (flattened/sequentialized) into a program written in a language which can be directly compiled to the instruction set of the given processor. Based on the machine instructions generated, the software metrics such as performance and memory size for the specification can thus be computed. The software metrics obtained in such a way are accurate since they are computed from the actual implementation of the specification on the given processor. However, due to the fact that automatic partitioning tools will evaluate hundreds or even thousands of partitions, this approach is too costly and time consuming since we would have to actually implement each partition on the given processor through flattening, sequentializing and compiling in order to get the software estimates for that partition. To get fast estimates while not sacrificing too much accuracy, the estimation model we propose combines two different approaches: an accurate approach for estimating leaf behaviors and a fast approach for estimating non-leaf behaviors and partitions. Prior to the partitioning process, each leaf behavior is compiled and estimated using the approach described in the previous section. During the partitioning process, the software estimates for each partition are constructively computed bottom up from the estimates of the leaf behaviors. Such a combined approach is very fast because it does not involve flattening, sequentializing and compiling for each partition during the design process. It only requires some computation based on the pre-obtained estimates for the leaf behavior specifications. Therefore, this model allows rapid evaluation of different design alternatives.

3 Performance Estimation

There are two different ways for obtaining performance metrics – dynamic simulation and static estimation. Given a set of input data, dynamic simulation actually executes the program and records the clock cycles used in each execution. Static estimation, on the other hand, is insensitive to input data. It just computes the average number of clock cycles required to execute the program. Static estimation can yield good results if the number of loop iterations is known and the conditional branching probability can be predicted correctly. Besides, static estimation takes much less time and space than dynamic simulation. In this section, we will describe a standard static estimation technique and its application to the performance estimation for our system-level specifications.

3.1 Flow Analysis

Flow analysis is a technique used in the static estimation of performance for design with conditional branching (including loops). Given a control flow graph $G = (V, E)$, where V is the set of vertices v_i, and E is the set of *directed* edges e_{ij} connecting vertex v_i to v_j and indicating sequencing between v_i and v_j, we wish to determine the execution frequencies of each of its nodes based on the branching probabilities. By determining the execution frequencies of the nodes, we can obtain useful information about the design by associating with each node in the graph, a *weight* representing some design parameter.

3.1.1 Determining the Branching Probabilities

Branching probabilities are associated with the edges in the control flow graph. The following ways have been used in our estimator: (1) *Equal Probabilities* : When there are n edges branching out from a node, all of them are assigned a probability of $1/n$. (2) *Loop Related Probabilities* : When the number of loop iterations is known, say n, the exit edge is assigned a probability of $1/n$ while the back edge is assigned a probability of $(n-1)/n$. (3) *User Defined Probabilities* : User can specify the branch probabilities using annotations in the input specification.

3.1.2 Determining the Node Execution Frequencies

The execution frequency of a node is defined as the number of times on the average that the node will be executed in a single execution of the graph. It is determined by the following procedure: (1) Determine the branch probabilities using one of the methods outlined above. (2) A start node, s, preceding the first node in the graph, is added. Its execution frequency, $F(s)$ is set to 1 since this node is executed exactly once whenever the control flow graph is executed. (3) Let $\mathrm{P}(e_{ij})$ be the branch probability of the edge between v_i and v_j. The execution frequency $F(v_j)$ for any node v_j is formulated in the following equation:

$$F(v_j) = \sum_{all\ immediate\ predecessor\ nodes\ v_i\ of\ v_j} F(v_i) \times P(e_{ij}) \quad (1)$$

(4) Solve the set of equations formulated in step 3 to obtain the individual node execution frequencies. There are a variety of methods such as Gaussian Elimination, LU decomposition, and Chomsky's method which can be used for solving a set of linear equations. We have selected the Gaussian Elimination method in our estimator.

In situations where the edge has an associated weight (as is the case when conditions exist on arcs between nodes), we may need to know the execution frequency of each edge, $F(e_{ij})$. It is obvious that the execution frequency of an edge is the same as that of its target node. Therefore we have $F(e_{ij}) = F(v_j)$.

3.1.3 Determining the Performance of Control Flow Graph

Let $W(v)$ and $W(e)$ be the weights associated with node v and edge e in control flow graph G. The performance

$P(G)$ of the graph G can be calculated as follows:

$$P(G) = \sum_{for\ all\ v \in V} (W(v) \times F(v)) + \sum_{for\ all\ e \in E} (W(e) \times F(e)) \quad (2)$$

3.2 Applying Flow Analysis to Performance Estimation

The VHDL code in each leaf behavior is divided into *basic blocks*. The details of obtaining basic blocks from VHDL code and compiling VHDL code to the generic instruction set are described in [6]. By using the execution times of generic instructions specified in the technology file, the execution time (i.e. weight) of each basic block is computed by summing that of each generic instruction in that basic block.

The basic block structure of a leaf behavior is mapped to an equivalent control flow graph G. Each basic block B_i is mapped to a node v_i in G. Each edge connecting two basic blocks B_i and B_j is mapped to an edge connecting node v_i and v_j in the graph. The weight of v_i is the same as weight of B_i. Each edge in G has a weight which is the same as the weight of the condition associated with the corresponding edge in the basic block structure. By applying flow analysis, the execution time for the leaf behavior can be computed using equation 2 in section 3.1.3.

Once execution times have been estimated for each of the leaf behaviors, we can merge the performance estimates of the leaf behaviors to yield the performance estimate of the next higher behavior in the hierarchy. To estimate the performance for B_{parent}, a non-leaf behavior with sequential sub-behaviors, we create a control flow graph $G = (V, E)$ for its sub-behaviors whose performance estimates are already known. For each of the sub-behaviors, B_i, of B_{parent}, there exists a corresponding vertex v_i in the graph G. For every transition arc between the two sub-behaviors B_i and B_j, the set E has a directed edge e_{ij} from vertex v_i to vertex v_j in G. After the control flow graph model has been constructed for the sub-behaviors, we can apply the flow analysis (section 3.1.3) to obtain the performance of the parent behavior B_{parent}.

In case a behavior at any level of the hierarchy has concurrent sub-behaviors, the execution time of that behavior is computed as the sum of that of its sub-behaviors since the concurrent sub-behaviors have to be sequentialized to execute on a uniprocessor. It must be mentioned here that a non-leaf behavior may have a descendant sub-behavior which does not have a *stop dot* in SpecChart. In this case the behavior will never finish executing and consequently the execution time returned for that behavior is an arbitrarily large number.

4 Memory Size Estimation

Given a behavior, the goal of memory size estimation is to determine how much program-memory (i.e. bytes used to store the compiled program representing the behavior) are needed.

The size of each type of generic instruction is specified in the technology file for target processor. Based on the size of each generic instruction, the program-memory size of each basic block is computed as the sum of that of all generic instructions in that basic block. The program-memory size of a leaf behavior in turn is the sum of that of all its basic blocks as well as the size of all conditions associated with the edges between basic blocks. Analogously, the program-memory size of a non-leaf behavior is the sum of that of all its sub-behaviors as well as the size of all conditions associated with the arcs between its sub-behaviors.

5 Results

We have compared the estimation results for different target processors including 8086, 80286, 68000, 68020 with those results obtained by compiling the designs directly into the instruction set of the target processors. Since there is no VHDL compiler available for those target processors, we first manually converted the VHDL code to its equivalent C code. Following that we compiled the C code into the instruction set of 8086, 80286, 68000 and 68020 processors. Based on the machine instructions generated from the C compilers and the instruction timing and size information provided in [7, 8, 9, 10], we have manually computed the actual performance and program-memory size for the designs. These were then compared with the estimates based on the generic three-address instruction compiler supplemented by corresponding technology files for the target processors.

The first two designs we choose were the fifth order elliptical filter and the differential equation example adopted from the high level synthesis benchmarks. The loop body in the differential equation is assumed executing 10 times (i.e. we have specified the branch probability for the exit edge of the loop as 0.1, the branch probability for the back edge of the loop as 0.9). The estimation results for the performance and memory size are shown in Figure 4.

Application	Target Machine	Actual Perf / Mem (cycles/bytes)	Estimated Perf / Mem (cycles/bytes)	Perf / Mem Estimation Errors
Elliptic filter	8086	2569 / 336	2488 / 319	−3.2% / −5.1%
Elliptic filter	80286	712 / 336	662 / 319	−7.0% / −5.1%
Elliptic filter	68000	1692 / 230	1632 / 214	−3.5% / −7.0%
Elliptic filter	68020	924 / 229	888 / 213	−3.9% / −7.0%
Differential equation	8086	9446 / 151	10586 / 143	12.1% / −5.3%
Differential equation	80286	2244 / 147	2304 / 141	2.7% / −4.1%
Differential equation	68000	6112 / 106	6452 / 98	5.6% / −7.5%
Differential equation	68020	3416 / 105	3676 / 97	7.6% / −7.6%

Figure 4: Performance (Perf) and memory size (Mem) Estimation.

Generally, compilers optimize the object code by using different optimization techniques such as global optimization, loop optimization and register allocation. Users can invoke those optimizations by passing special flags to the compiler. In the experiments of Figure 4, we have disabled those optimizations during the C compilation since our generic compiler does not use optimization heuristics.

The next design we experimented is a real-time embedded medical system used to measure a patient's bladder volume. Its SpecChart description is described in [6].

There are two timing constraints imposed on the system. One is associated with the leaf behavior DATA_ACQUISITION, which requires that the acquisition and conversion of 1000 data points take place in

less than 1 ms. The other is associated with the non-leaf behavior ONE_SCAN, which requires that the maximum time between two scans, i.e. the time used to execute MOTOR_CONTROL, DATA_ACQUISITION, VOLUME_COMPUTATION and DATA_STORAGE, is 1 second. We have estimated behavior DATA_ACQUISITION and behavior ONE_SCAN using our estimator. The estimates are compared with the actual results obtained from the (non-optimized) target machine instructions (Figure 5).

Application	Target Machine	Actual Performance (in clock cycles)	Estimated Performance (in clock cycles)	Estimation Error
DATA_ACQUISITION	8086	82015	71004	−13.4%
DATA_ACQUISITION	80286	27056	24002	−11.3%
DATA_ACQUISITION	68000	92089	88012	−4.4%
DATA_ACQUISITION	68020	44044	41006	−6.9%
ONE_SCAN	8086	189914	153641	−19.1%
ONE_SCAN	80286	65931	55580	−15.7%
ONE_SCAN	68000	198665	175620	−11.6%
ONE_SCAN	68020	98982	84531	−14.6%

Figure 5: Performance estimation.

If the behavior DATA_ACQUISITION were to execute on an 8086 microprocessor with 12 MHZ clock rate, the estimator predicted 71004 clock cycles which is equivalent to 5.9 ms in this case. Since the timing constraint (1 ms) imposed on DATA_ACQUISITION was violated, the partitioner will conclude that custom hardware must be designed for this behavior. The timing constraint of 1 second imposed on the behavior of ONE_SCAN has not been violated since it only requires 12.8 ms (153641 clock cycles) to execute all behaviors in ONE_SCAN on the chosen microprocessor. If behavior DATA_ACQUISITION is extracted out and implemented by some faster design, the execution time for behavior ONE_SCAN can be expected to be less than 12.8 ms. Therefore, all behaviors in ONE_SCAN except DATA_ACQUISITION can be implemented as software running on the microprocessor.

It took 0.54, 0.33, 0.96 and 1.65 seconds on a Sun4 system to estimate the performance of the elliptic filter, differential equation, DATA_ACQUISITION and ONE_SCAN respectively in conjunction with the 8086 technology file. This can be compared to the several days required to manually compute the same information.

6 Conclusion and Future Work

Starting from a system-level specification, we have presented techniques for estimating the performance and memory size of software compiled to execute on a given processor. The experiments have shown that our estimator has an average error of 7.4% and has a maximum error of 19.1% on designs spanning from straight line code (elliptic filter) to code with branches and loops (differential equation and behavior DATA_ACQUISITION) and even hierarchical specification (behavior ONE_SCAN). Experiments also show that our estimator is very fast and can provide rapid feedback to the designers or system partitioning tools and enable them to explore a larger design space, which may lead to faster and/or cheaper designs.

Since the generic three-address instructions and the technology files can only characterize the target machine instructions to some extent, there is always a difference between the estimates obtained from our estimator and the results obtained directly by compiling to target machine instructions. We can expect more accurate estimation by enhancing the technology files to include more information about the target machine instruction sets. Currently our generic instruction set has limited formats, especially in terms of memory addressing modes. If we can incorporate more memory addressing modes in our compiler to close the gap between the generic instructions and the target machine instructions, we can expect better estimation results. However, this may increase the complexity of the generic instructions. Increasing complexity of the generic instructions may increase the compiling time and hence increase the total estimation time. Therefore, more studies are needed to determine what constitutes a suitable generic instruction set.

References

[1] R.K. Gupta, and G. De Micheli, "System-level synthesis using re-programmable components," Proc. of the European Conference on Design Automation, 1992.

[2] R. Ernst, and J. Henkel, "Hardware-software codesign of embedded controllers based on hardware extraction," The International Workshop on Hardware/Software Codesign, Estes Park, Colorado, September, 1992.

[3] W. Ye, R. Ernst, Th. Benner, and J. Henkel, "Fast timing analysis for hardware-software cosynthesis," Proc. of ICCD, 1993.

[4] Th. Ball, and J.R. Larus, "Optimally profiling and tracing programs," ACM Sigplan Symp. Principles of Programming Language, 1992.

[5] S. Narayan, F. Vahid, and D. Gajski, "System specification and synthesis with the SpecCharts language," Proc. of ICCAD, 1991.

[6] J. Gong, D. Gajski, and S. Narayan, "Software estimation from executable specifications," Technical Report 93-5, Dept. of Information and Computer Science, University of California, Irvine, March, 1993.

[7] G. Gorsline, 16-Bit Modern Microcomputers: the INTEL I8086 Family. Prentice-Hall, 1985, pp. 473-512.

[8] L. Scanlon, 8086/8088/80286 Assembly Languages. Simon & Schuster, 1988, pp. 361-368.

[9] J. Bennett, 68000 Assembly Language Programming: A Structured Approach. Prentice-Hall, 1987, pp. 445-455.

[10] MOTOROLA, MC68020: 32-Bit Microprocessor User's Manual (Second Edition). Prentice-Hall, 1987, Chapter 9.