A SOFTWARE TOOL FOR THE TIMING ANALYSIS OF EMBEDDED SOFTWARE

Gabriele Luculli
Scuola Superiore S.Anna
Via Carducci, 40 - 56100 Pisa - Italy
luculli@sssup.it

Alberto Sangiovanni–Vincentelli
Department of EECS
University of California at Berkeley, CA 9472
alberto@eecs.berkeley.edu

ABSTRACT

The presence of real–time software modules which strongly interact with specific hardware architectures is steadly growing in today embedded applications. New methods and tools are needed for the program analysis and validation of these designs. The timing analysis of software is an essential aspect because real–time requirements need to be validated and performance objectives could be missed if the software design does not fit with the hardware design. In this paper we describe a new timing analysis for software which is executed on architectures with a one–level instruction cache. The safeness of the timing estimates is guaranteed by the method and the accuracy can be traded with the processing time. The implementation details of the related software tool are reported and the practical use of the tool is shown by some experimental results.

1. INTRODUCTION

In the past, electronic design automation was mostly oriented toward hardware components, instead today there is a growing interest for methodologies and analysis techniques for the software which is regarded as the major source of flexibility. New methods and tools are needed for both the development and the validation of software modules with strong interactions with the hardware architecture [2, 5]. In particular the timing analysis of the software is an important step of any design, because the timing of the application must be validated against the real–time requirements and the system performance can be badly wasted if a good software design does not match with a good hardware design. Today, the embedded software is quickly becoming the magic potion to add to rough hardware blocks for obtaining a successful product and the new software analysis techniques are the magic spell we need to cook such potion.

In this paper, we show the use of a software tool and the related analysis method for the timing analysis of embedded software which is executed on a typical hardware architecture of a microprocessor with a one–level instruction cache. The I-cache is a fast memory which is commonly found in any high–performance microprocessor. It is necessary to obtain the required performance but it is also a huge source of uncertainty for the execution time of the software. In the past, it has been considered as an unpredictable element or at least partially predictable but with a such poor accuracy that no practical use could be granted for any real–time application. We have recently developed a method named Implicit Cache Simulation [6] which permits the estimation of the worst–case and best–case miss count with a bounded accuracy. Moreover, it is possible to trade the response time of the simulation with the accuracy of the estimates just changing a single parameter i.e. the depth level. In general, it is always possible to improve the accuracy. The same approach can be used to study the timing properties of embedded software as well as to optimize the system performance by basic blocks reordering and optimal code allocation. In this paper, we apply our method to a simple case study and we show the basic properties of the Implicit Cache Simulation.

2. RELATED WORK

The research on timing analysis of cached architectures has received a lot of attention in the past years. Two main approaches have emerged: an analytical method due to Li et al. [4] and static analysis methods due to Mueller et al. [1] as well as to Ferdinand [3].

Li method is based on an integer linear programming formulation: the code and the cache behavior are modeled by linear constraints, the optimization function is the program execution time. The formulation is general enough to consider both direct–mapped and set–associative caches. Unfortunately, integer linear programming is an NP–complete problem, so computation time may be quite large.

Muller and Ferdinand use the static analysis to predict the cache behavior. The cache is modeled in term of abstract objects and abstract functions, and the timing bounds are evaluated by fix–point approximations on the control–flow structure. Their experimental results show that the estimates are always safe but the accuracy strongly depends on the specific program and cache parameters. The presence of multiple conflicting paths is treated with a lot of pessimism and this results in a poor accuracy.

In general, these approaches do not trade the estimation accuracy with the computation time, neither they permit to improve the accuracy. In the worst–case, the analysis of a program may require a large processing time and result in a loose accuracy.

3. IMPLICIT CACHE SIMULATION

Implicit Cache Simulation (ICS) is a method for the timing analysis of embedded software which is executed on hardware architectures with I-cache. Currently, it considers the direct–mapped cache with any cache and line sizes, and it completely takes into account the effects of code allocation (i.e. code layout in main memory) as well as basic blocks reordering. The ICS is an hybrid method which is based on two different approaches: the static program analysis and the simulation of the code execution. From static analysis methods it borrows the idea of representing the behavior of the cache with an abstract model i.e. an *abstract cache state* and some *abstract functions* which operate on the abstract state. Whether, it makes use of simulation techniques for evaluating the cache performance and the performance metrics of the code.

The basic idea behind our method is that the execution time of a program is a *structural property* of the program. The presence of a I-cache obviously changes the execution time of each dynamic paths, but it is still possible to group

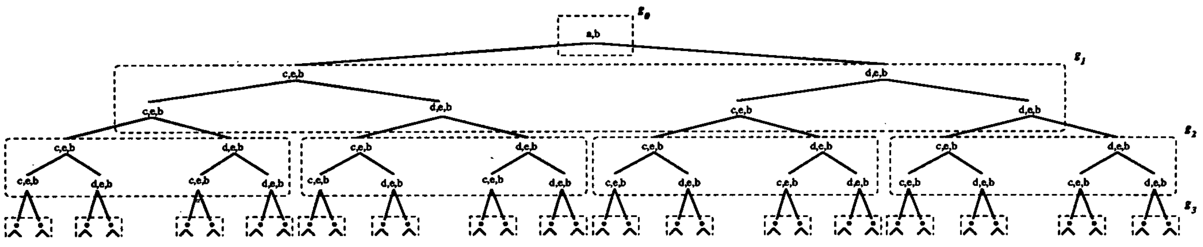

Figure 1. Partitioning of the computation tree of the Max benchmark in a set of sub–trees (depth $\delta = 2$)

together the paths which have the same execution time. How the dynamic paths get grouped depends on the structure of the program and the cache parameters. That is, it is a structural property of the program. It is not possible an exact evaluation of all dynamic paths because in general an exponential number of paths can be present, so it is not possible the exact evaluation of the worst–case/best–case execution time. Nevertheless, it is possible to approximate the exact execution time by approximating groups of paths. The ICS works exactly in this way, first it builds some approximations of path groups and then it looks for the worst–case/best–case path by moving from group to group.

Approximated groups of paths are build by partitioning the computation tree of the program in a set of sub–trees $g_R = \{g_0, \ldots, g_{\lceil \frac{N+1}{\delta} \rceil}\}$ (Figure 1), evaluating the abstract cache state at the end of each path in the sub–tree and then reducing the overall set of abstract cache states at the end of each sub–tree. The computation tree of a program is a tree where each node represents a possible program execution state and each edge represents the changing of program state after the execution of some basic blocks of the program. The reduction at the end of each sub–tree is performed by an abstract function *Reduce*() which produces a single abstract cache state from a set of abstract cache states. The overall correctness of the method is proved by the following theorem and lemma. The theorem states that a safe approximation can be computed and it suggests a possible algorithm to implement the timing analysis. The lemma characterizes the sequence of sets of abstract cache states involved in the theorem.

Theorem 1: Given a control–flow graph g with a single loop (loop bound N), a decomposition of his tree of computation $g_R = \{g_0, \ldots, g_{\lceil \frac{N+1}{\delta} \rceil}\}$, an initial abstract cache state $\hat{c}_I$ and a function f which models the program execution starting from an abstract cache state $\hat{c}_I$, $f(g, \hat{c}_I) = (\alpha, \hat{c}')$ (where α is the miss count of any complete dynamic path in the control–flow graph g, and $\hat{c}'$ is the final cache state). Two bounding function $\hat{f}^A$ and $\hat{f}^B$ can be defined in term of f, such that the following relations hold:

$$\alpha_0^A + \alpha_1^A + \ldots + \alpha^A_{\lceil \frac{N+1}{\delta} \rceil} \leq \alpha \leq \alpha_0^B + \alpha_1^B + \ldots + \alpha^B_{\lceil \frac{N+1}{\delta} \rceil} \quad (1)$$

where the α_i^A, α_i^B values are computed as, $k \in \{A, B\}$:

$$\begin{gathered} (\alpha_0^k, \hat{c}_0) = \hat{f}^k(g_0, \hat{c}_I) \\ (\alpha_1^k, \hat{c}_1) = \hat{f}^k(g_1, \hat{c}_0) \\ \ldots \\ \ldots \\ \left(\alpha^k_{\lceil \frac{N+1}{\delta} \rceil}, \hat{c}_{\lceil \frac{N+1}{\delta} \rceil}\right) = \hat{f}^k\left(g_{\lceil \frac{N+1}{\delta} \rceil}, \hat{c}_{\lceil \frac{N+1}{\delta} \rceil - 1}\right) \end{gathered}$$

◇

Lemma 1: The sequence of sets of abstract cache states $\left(\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{\lceil \frac{N+1}{\delta} \rceil}\right)$ obtained by theorem 1, has the following property:

$$\hat{c}_1 = \hat{c}_2 = \ldots = \hat{c}_{\lceil \frac{N+1}{\delta} \rceil - 1} \quad (2)$$

The approximation of f is obtained by the two bounding functions $\hat{f}^B$ and $\hat{f}^A$ which are evaluated on each sub–tree g_i. The *depth* δ of the sub–trees g_i (except g_0 which is a single node) is the key parameter of the implicit cache simulation. Greater values of the depth result on better accuracy of the estimation because a reduced number of abstract cache states are lost in the reduction process at the end of each g_i. The drawback is that the processing time increases as well, because an increasing number of basic blocks must be simulated, but fortunately in an affordable way. The proofs of Theorem 1 and Lemma 1, as well as their generalization for control–flow graphs with any number of loops, are reported in [6].

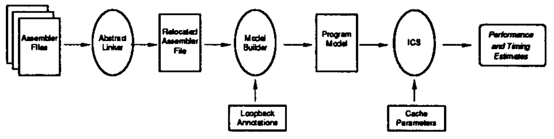

Figure 2. Overview of the ICS tool

4. IMPLEMENTATION AND USE OF THE TOOL

An overview of the tool and interfaces involved in our timing analysis is depicted in Figure 2. The implicit cache simulation is performed by the ICS module which takes as inputs a program model file and the cache parameters. The results are an estimation of the worst–case/best–case execution time (or miss count if just the cache effects are considered) as well as the accuracy of the estimates and the traces of the worst–case/best–case paths. The *program model* is a text description of the control–flow graph annotated with constant loop bounds and informations on the code allocation. An example is reported in Figure 3 for a C benchmark which computes the max value in an array of N elements. This example has been chosen because it is simple enough to permit a clear description of ICS, but it is not trivial because a huge number of dynamic paths are generated for a realistic value of array elements N. Two different code allocations are considered, so two different layouts of the benchmark are reported in Figure 3, layout L1 and layout L2. Following there is the program model of the Max benchmark for the memory layout L1:

```
Task_Definition:
;Section 1 - Global Info
Name            Task1
```

```
Num_BB          7
Start_Point     a   0
Start_Addr_All  0x0000

; Section 2 - Basic Blocks
; with Code Allocation (L1)
BB_Definition   a 0x0a
BB_Definition   b 0x06
BB_Definition   c 0x08
BB_Definition   e 0x02
BB_Definition   f 0x06
BB_Definition   X 0x10
BB_Definition   d 0x0a

; Section 3 - Control Flow Graph
CFG_Start       a
CFG_Definition  a -> b
CFG_Definition  b -> c    b -> d
CFG_Definition  c -> e    d -> e
CFG_Definition  e -> f

CFG_Loopback    e -> b    10    e
End_Task_Definition.

; Section 4 - Microarchitecture
MicroA_Definition:
ICache          32    8
End_MicroA_Definition.
```

Any program model consists of four sections. A first section with some general information such as the starting address of the code and the entry point in the control–flow graph. A section with the characteristics of each basic block that is the name and the code size. In main memory the basic blocks are allocated in the same sequence reported in the program model. It is possible to model non contiguous allocations by adding fictitious basic blocks which are not referred in the following sections (e.g. the basic block X is used just to model the allocation L1). The third section describes the structure of the control–flow graph as a set of direct links between basic blocks and a set of loopbacks. The last section describes the microarchitecture of the microprocessor which executes the code. By now, just the description of the I-cache is present in term of cache and line sizes, but in the future we plan to add others elements such as the pipeline structure and a more complex memory architecture.

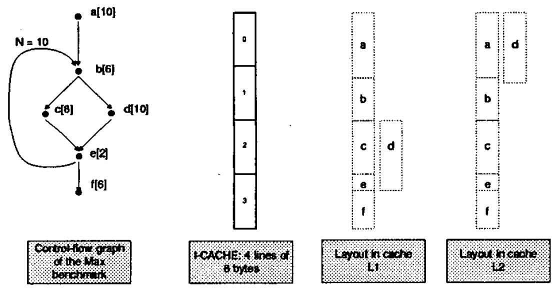

Figure 3. Annotated control–flow graph of the Max benchmark and two layouts of the code in cache

The ICS module (Figure 2) parses the program model, it builds some internal data structures and then it applies the following algorithm to perform the timing analysis:

```
void TimingEval() {
int alphaA = 0, alphaB = 0;
abstract_cache_type cHat;

  /* First Step */
  <Initialize the abstract cache cHat to empty lines>;
  <Joint evaluation of functions f_A, f_B on trees g0 and g1
   with abstract cache cHat>;
  <Update cHat with the final abstract cache state and
   the current estimates alphaA, alphaB>;
  <Reduce cHat>;

  /* Second Step */
  <Eval f_A and f_B on gi>;
  <Update cHat with the final abstract cache state and
   update ( ceil((N+1)/deepness)) - 2 ) times the current
   estimates alphaA, alphaB>;
  <Reduce cHat>;

  /* Third Step */
  <Eval f_A and f_B on the last tree>;
  <Update cHat with the final abstract cache state and
   the current estimates alphaA, alphaB>;
}
```

Thanks to lemma 1, the complexity of the algorithm becomes completely independent from the value of the loop bound N. So it is possible to implement the timing analysis in a simple three-steps algorithm rather than in $\lceil\frac{N+1}{\delta}\rceil + 1$ steps as suggested by theorem 1. In the tool implementation the evaluations of g_0 and g_1 have been joined together in the first step because we just need the timing estimates and the abstract cache state $\hat{c_1}$ at the end of $g_0 \circ g_1$. The second step is the generic evaluation of g_i starting from an abstract cache state $Reduce(\hat{c_1})$ (i.e. the reduced cache state) which is the most general cache state after the first iteration of the loop. By general cache state we mean an abstract cache state which permits to obtain a safe estimation without loosing accuracy. The results of the second step are an abstract cache state and the timing estimates of the tree g_i which must be counted $\lceil\frac{N+1}{\delta}\rceil - 2$ times. The third step ends the analysis by evaluating the last tree.

Following there is a detailed example of the timing analysis of the program model reported in Figure 3 (loop bound $N = 10$, code allocation L1, 32-byte cache size and 8-byte line size). In order to make apparent the analysis method we assume that the plain execution time of each basic block is equal to zero. So we just consider the effects of the cache on the timing analysis in term of the miss count (depth $\delta = 1$).

Step 1: The initial abstract cache state is empty $\hat{c} = \hat{c}_I = \{-,-,-,-\}$ and the initial estimated miss count is zero $\alpha^A = \alpha^B = 0$.

$$\begin{array}{ll} \hat{f}^B(g_0 \circ g_1, \hat{c}) = & (5, \{\{a,(a,b),c,(e,f)\}\{a,(a,b),d,(e,f)\}\}) \\ \hat{f}^A(g_0 \circ g_1, \hat{c}) = & (5, \{\{a,(a,b),c,(e,f)\}\{a,(a,b),d,(e,f)\}\}) \\ \alpha^A = 5 \quad \alpha^B = 5 & \\ \hat{c} = Reduce(\hat{c_1}) = \{a,(a,b),-,(e,f)\} & \end{array}$$

Step 2: The most general abstract cache state after the first iteration is $\hat{c} = \{a,(a,b),-,(e,f)\}$.

$$\begin{array}{ll} \hat{f}^B(g_i, \hat{c}) = & (3, \{\{a,(a,b),c,(e,f)\}\{a,(a,b),d,(e,f)\}\}) \\ \hat{f}^A(g_i, \hat{c}) = & (2, \{\{a,(a,b),c,(e,f)\}\{a,(a,b),d,(e,f)\}\}) \\ \alpha^A = 5 + 2 * 9 = 23 \quad \alpha^B = 5 + 3 * 9 = 32 & \\ \hat{c} = Reduce(\hat{c_i}) = \{a,(a,b),-,(e,f)\} & \end{array}$$

Step 3: Evaluation of the last loop iteration up to the end of the control–flow graph. The actual abstract cache state is $\hat{c}$.

$$\begin{array}{ll} \hat{f}^B(g_i, \hat{c}) = & (3, \{\{a,(a,b),c,(e,f)\}\{a,(a,b),d,(e,f)\}\}) \\ \hat{f}^A(g_i, \hat{c}) = & (2, \{\{a,(a,b),c,(e,f)\}\{a,(a,b),d,(e,f)\}\}) \\ \alpha^A = 23 + 2 = 25 \quad \alpha^B = 32 + 3 = 35 & \\ \hat{c} = Reduce(\hat{c}_{11}) = \{a,(a,b),-,(e,f)\} & \end{array}$$

The final result of the timing analysis is that the worst-case miss count α is $25 \leq \alpha \leq 35$. In this example, the

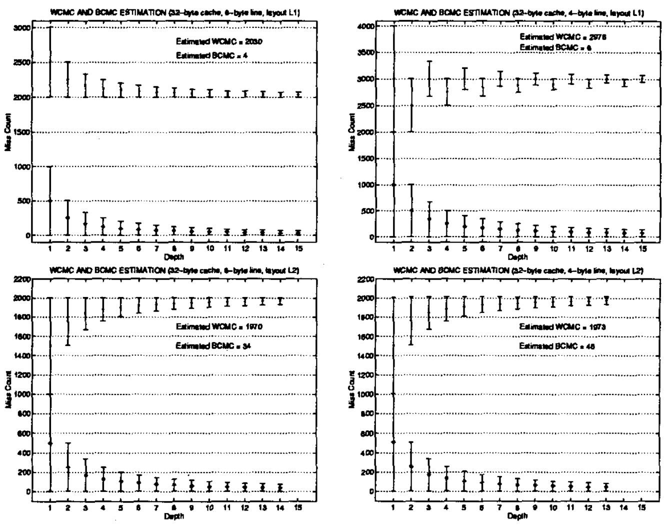

Figure 4. Timing analysis results of the Max benchmark for layouts L1, L2 and a 32-byte I-cache with 8/4-byte line size

actual value can be computed by exact evaluation of the 2^{11} paths and it results on 25 misses. So our estimate is safe (this is guaranteed by the theorem 1), but it might be too conservative for practical use. A better estimate can be obtained by adopting greater values of the depth parameter e.g. considering a depth $\delta = 2$ the timing analysis produces $25 \leq \alpha \leq 30$ and $\delta = 3$ results on $25 \leq \alpha \leq 28$.

The ICS tool can be used for analysis purposes of the code as well as for the I-cache design and the optimization of the code allocation. Following we consider the previous example with a loop bound $N = 1000$, 2^{1001} different dynamic paths are present and no other existing method is able to furnish tight estimates with a bounded accuracy in such case. We will perform a complete analysis by changing the code allocation (a general layout L1 and an optimized one L2) and the cache parameters (direct-mapped cache, 32-byte size, 8/4-byte line size). The results produced by our tool are shown in Figure 4. The following hints can be drawn:

- The ICS analysis always converges to estimates with 1% accuracy after few values of the depth parameter. The processing time in the worst case was few seconds in a Pentium II machine with Linux OS.
- Reducing the line size to 4-byte can be quite dangerous for the performance of the system. In the worst case it results on a 50% increase of the miss count for the code allocation L1. Instead the code allocation L2 is quite insensitive to the line size.
- From the performance view point the two code allocations are equivalent, no one permits a true reduction of the worst–case miss count. Nevertheless, the allocation L2 is to be preferred because it needs the few amount of memory for the code (40 bytes instead of 56 bytes).
- The best–case miss count is quite independent from both the code allocation and the cache parameters.

In conclusion, we believe that the ICS analysis and the related software tool can be efficiently used for the analysis of embedded code. Timing analysis for real–time requirements validation, performance analysis of I-cache, I-cache design and optimal code allocation are just few examples of what it is required by the design methodologies for embedded systems and they are not completely available today. The ICS tool is flexible and powerful enough to be used for all these applications.

5. CONCLUSIONS

We have described a timing analysis of embedded software for hardware systems with cache memory and presented the related software tool. The method permits to trade the accuracy of the analysis with the processing time. The implemented tool can be efficiently used to analyze the complex behavior of programs on cached architectures as well as to optimize the design of embedded software. A case study on bounding the miss count by the ICS analysis has been presented. Future research includes the extensions to set-associative caches and in general to hierarchical memory architectures.

REFERENCES

[1] R. Arnold, F. Mueller, D.B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th Real Time Systems Symposium, San Juan*, pages 94–105. Dec. 1994.

[2] F. Balarin, E. Sentovich, A. Sangiovanni-Vincentelli, et al. *Hardware-Software Co-Design of Embedded Systems.* Morgan Kaufmann Publishers, 1998.

[3] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Universitat des Saarlandes, 1997.

[4] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. In *International Conference on Computer Aided Design*, pages 380–387, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[5] S. Lorenzini, G. Luculli, and C. A. Prete. A fast procedure placement algorithm for optimal cache use. In *Proceedings of 9th Mediterranean Electrotechnical Conference.* IEEE Press, 1998.

[6] G. Luculli. *Software Performance Estimation of Real-Time Embedded Systems.* PhD thesis, University of Pisa, 2000. (Italian language).