Intervals in Software Execution Cost Analysis

Fabian Wolf, Rolf Ernst
Institut für Datenverarbeitungsanlagen, Technische Universität Braunschweig, Germany
$\{wolf|ernst\}$@ida.ing.tu-bs.de

Abstract

Timing and power consumption of embedded systems are state and input data dependent. Formal analysis of such dependencies leads to intervals rather than single values. These intervals depend on program properties, execution paths and states of processes, as well as on the target architecture. This paper presents an approach to analysis of process behavior using intervals. It improves previous work by exploiting program segments with single paths and by taking the execution context into account. The example of an ATM cell handler demonstrates significant improvements in analysis precision.

1. Introduction

Accurate software running time and power analysis are key to optimized system synthesis. In general, imprecise estimation of software execution costs (such as running time and power) increases design risk or leads to inefficient designs. Profiling and simulation are the state-of-the-art in industry, but since exhaustive simulation is impractical, simulation results can only cover part of the system behavior. Static analysis is a more complicated but attractive alternative. It provides lower and upper bounds reflecting data dependent control flow as well as data dependent statement execution cost. In the past, these bounds were wide due to a lack of efficient control flow analysis and architecture modeling techniques. Significant progress in both areas has made formal analysis practical.

Intervals for software execution cost depend to a certain extend on the process control flow which depends on process input data. Execution cost of the software processes and, hence, of the overall system are context dependent. We will use an example from wireless communication, where there are several paths on which different data packets are routed through a network of software processes. Important questions of the system architect can be the power consumption for sending a data packet or the time to set up a connection in a base station. This should take the system context into account, since for each packet type the processes react with a different control flow. Of course, simulation is always possible and statistical execution cost analysis is feasible, but the first approach is not reliable and the second is just an approximation of the complex hardware activities when executing a set of communicating software processes. We will show with realistic examples that the static analysis approach provides reliable and narrow intervals for context dependent process execution cost that is automatically evaluated by the analysis tool.

We explain the influence of data dependent control flow on software execution cost in section 2. Data dependent instruction execution is explained in section 3. In section 4 we present an example before we conclude in section 5.

2. Program Path Analysis

For path analysis techniques [7], a program is typically divided into basic blocks, where a **basic block bb** is a program segment which is only entered at the first statement and only left at the last statement [1]. Any program can be partitioned into disjoint basic blocks. Then, the program structure is represented as a directed program flow graph with basic blocks as nodes. For each basic block a cost with respect to each interval is determined. A longest and shortest path analysis on the program flow graph is used to identify a global interval. This procedure does not yet provide sufficient accuracy. For acceptable analysis precision one must identify feasible paths through a program. A **feasible program path** or trace is a path in this flow graph corresponding to a possible sequence of basic blocks when the program is executed from the first to the last basic block of a program. A program segment is a sequence of nodes in a program flow or syntax graph. This definition implies a hierarchy of program segments. Not all paths in the graph represent feasible program paths. A **false program path** is a path in the graph which cannot be executed under any input condition. False path identification is essential for programs with loops since loops correspond to cycles in the graph which lead to an infinite number of potential paths and resulting infinite cost intervals.

2.1. Previous Work

The approaches by Puschner and Koza [10] and Park and Shaw [9] require iteration bounds for all loops in the program which the user must provide by loop annotation. While making formal analysis feasible, loop bounding alone is not sufficient for accurate path analysis. The approach by Gong and Gajski [5] can partially consider false paths because the user can specify the branching probabilities. As a second step in [7] and in [9], the user is asked to annotate false paths. The number of false paths can be very large. Instead of enumerating false paths or, conversely, feasible paths, a language for user annotation with regular expressions is introduced in [9]. Still, the number of required path annotations can be extremely large in practice, as demonstrated with even small examples in [7]. A major step forward was the introduction of implicit path enumeration [7]. Here, the user provides linear (in)equations to define false paths. To evaluate these (in)equations, Li and Malik map the upper and lower bound identification to two ILP problems, the one optimizing for the lower, the other one for the upper cost bound. Previous work by Ferdinand in [4] bases on this kind of interaction while abstract interpretation is used for the static prediction of cache and pipeline behavior. Abstract interpretation has also been used to reduce designer interaction for loop bounding [6].

2.2. Execution Cost

The execution time model in [7] is established as a standard model for static approaches which is called the sum-of-basic-blocks model in [13] for timing. It can be extended to power consumption [11] and data rates, abstracting the timing to the execution cost c of a program segment in general. Let a program consist of N basic blocks with x_i execution count of basic block bb_i and c_i execution cost. Then, the **sum-of-basic-blocks model** defines for the program execution cost interval:

$$C = \sum_{i}^{N} c_i \times x_i$$

For the execution count interval $[x_{i,min}, x_{i,max}]$, the designer provides an implicit description of the possible paths by means of linear equations for execution counts. The structural constraints define another set of equations: The execution count inflow d of a basic block equals its execution count x and its execution count outflow d.

$$\sum_{bb} d_{inflow} = x_{i,bb} = \sum_{bb} d_{outflow}$$

These (in)equations for the upper and the lower execution count bound are mapped to two ILP problems which can be solved to derive the widest execution count interval $[x_{i,min}, x_{i,max}]$ for each basic block or program segment.

It is assumed that all executions of one basic block have the same cost. However, data dependent instruction execution and super scalar or super pipelined architectures with overlapped basic block execution as well as cache behavior lead to widely varying local path cost with respect to latency time and power consumption. This has a substantial effect on the cost interval. For these architectures, the sum-of-basic-blocks model cannot provide close bounds, but must be pessimistic to be correct. For higher accuracy, basic block sequences in program segments must be considered. This shall be called the **sum-of-program-segments** model containing basic block sequences which is a major improvement compared to the state-of-the-art.

2.3. Path Classification

Program properties can be exploited to simplify path analysis for the determination of the execution cost through basic block sequences [13]. Large parts of typical embedded system programs have a single program path only. An FIR filter is a simple example and a Fast Fourier Transform is a more complex one. There is only one path executed for any input pattern, even though this path may wrap around many loops, conditional statements and even function calls which are used for program structuring and compacting. A program segment has a **Single Feasible Path SFP**, when paths through this segment are not depending on input data. A program segment with an SFP is an **SFP-segment**.

Previous analysis approaches give more than one execution path for SFP programs because they do not distinguish between input data dependent control flow and program structuring aids. In the best case, they may be accurate but require much designer interaction for SFP program segments and still do not deliver the path segment costs such as [7]. In case of SFP, execution would choose the one correct path and sequence for any input pattern without further designer interaction. Most practical systems also contain non-SFP parts. These have multiple feasible paths MFP. A program segment has **Multiple Feasible Paths MFP**, when paths through the program segment are depending on input data. A program segment with MFP is an **MFP-segment**. Isolation of SFP and MFP parts can help to exploit SFP.

In [13], SFP are exploited by finding SFP and MFP nodes in the control flow graph. Embedded MFP are cut out and analyzed separately using the ILP approach while SFP are analyzed by simulating the timing of the only path. Costs for cutting out the MFP and the MFP cost interval delivered by the ILP solver are added. This leads to tighter cost bounds compared to [7]. In this paper, we present major improvements. The approach in [13] can only deal with one level of embedded MFP. If several levels of hierarchy with SFP and embedded MFP are present, they have to be analyzed separately, so dependencies across the hierarchical levels are lost leading to overly pessimistic cost bounds in case of complex programs. We extend this approach to a global cost interval calculation for all levels which provides higher analysis precision. The syntax graph instead of the control flow graph is chosen because it can directly cover the hierarchy of control structures and rewriting the program to generate a control flow graph is not necessary.

2.4. Identification of Program Properties

Syntax Graph For the identification of SFP and MFP segments, the input program is mapped to a **syntax graph**.

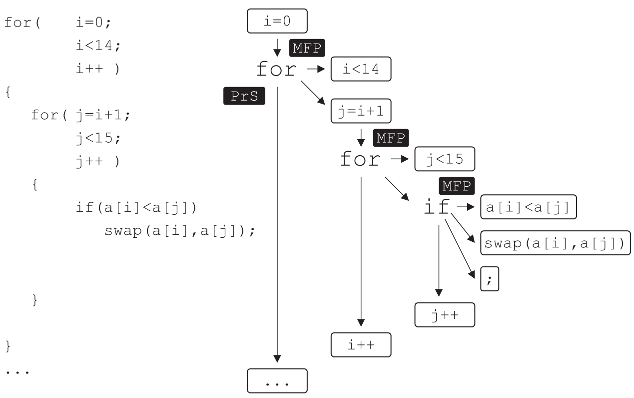

Figure 1. Hierarchical syntax graph

The syntax graph of a bubble sort algorithm is shown in figure 1. In this syntax graph, every control structure, such as if and for, is a hierarchical node. The basic blocks are the leaf nodes with the according basic block cost. Every control structure has edges with different meanings. The ”control” edge that decides which of the paths is executed and the ”successor” edge that leads to the next node are part of every control structure while the ”then” and the ”else” edge are specific for the $if/else$ program segment. The same restrictions to use structured programs are assumed as in [7]. Control flow enters and leaves an $if/else$ program segment exactly twice for the given hierarchy level, once for the control structure and once for either the ”then” or the ”else” edge like in figure 2.

Feasible Paths in the Syntax Graph A Program Segment **PrS** in the syntax graph is a sequence of syntax nodes with exactly one first and one last basic block. This follows the definition of basic blocks in [1]. A Program Path Segment **PaS** is a path through a program segment PrS. A Single Feasible Path Program Segment **SFP-PrS** is a hierarchical PrS with exactly one PaS through the PrS of the syntax graph while a Multiple Feasible Path Program Segment **MFP-PrS** is a hierarchical PrS which is no SFP-PrS.

A Maximum Program Segment **MPrS** is a PrS with all SFP-PrS and MFP-PrS on the same level of hierarchy. A Maximum Single Feasible Path Program Segment **MSPrS** is a maximum sequence of consecutive SFP-PrS on the same level of hierarchy. It has exactly one PaS for execution.

A depth first search algorithm on the syntax graph can be used to determine input data dependencies of conditions using symbolic simulation of basic blocks [13]. Every control structure which does not contain an input data dependent condition must be SFP. Leaf nodes are SFP by definition. If conditions contain input data, or symbolic execution is not successful due to the complexity of symbolic expansions, the syntax graph nodes are classified as MFP. This leads to wider cost intervals. This algorithm classifies each hierarchical node. PrS with MFP child nodes are classified as MFP because the multiple paths also enter and leave this hierarchical node even when their control structure is independent of input data. The for-PrS in figure 2 which shows the inner loop of figure 1 potentially has $2^{iterations}$ paths because control flow splits in the if-PrS.

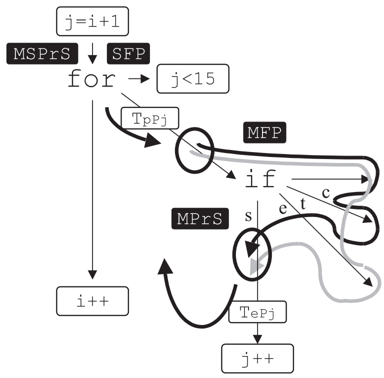

Figure 2. Execution paths in the graph

To treat such situations, we introduce a pseudo SFP-PrS. A **pseudo SFP-PrS** is an SFP-PrS with a single PaS on one level of hierarchy while lower levels may have multiple paths as in figure 3. On this level of control hierarchy, it can be treated like an SFP-PrS as we prove in [12].

Program Segment Cost Cost determination requires a Program Segment Execution **PsE**. It is an execution of a PaS through the complete PrS. Details can be found in [13] and in section 3. There can be a minimum and a maximum cost for a single PaS through the PrS because of data dependent instruction execution. The **PrScost** is the cost for the execution of a PrS. PrScost is determined according to its PrS classification.

$$\mathrm{PrScost}(\mathrm{PrS}_i) = \begin{cases} \mathrm{MSPrScost}(\mathrm{PrS}_i) & \mathrm{PrS}_i \text{ is MSPrS} \\ \text{MFP-PrScost} & \mathrm{PrS}_i \text{ is MFP-PrS} \\ \text{undefined} & \text{else} \end{cases}$$

The **SFP-PrScost** is the cost for the execution of an SFP-PrS. SFP-PrScost is determined by PsE. In practice, all consecutive SFP-PrS and pseudo SFP-PrS are joined in MSPrS, so MSPrScost is computed by PsE covering the SFP-PrScost. The **MSPrScost** is the cost for the execution of an MSPrS. Let PrS_i be MSPrS:

$$\textbf{MSPrScost}(\mathrm{PrS}_j) \text{ is determined by } \textbf{PsE}$$

for the cost of the one executed path. The **PaScost** is the cost for the execution of a PaS. The path may cover MSPrS and MFP-PrS. It can contain a lower hierarchical level MFP-PrS requiring a descend in the recursive cost analysis approach until we get to a PaScost without MFP-PrS, i.e. only containing MSPrScost delivered by PsE as above. The **MFP-PrScost** is the cost of an MFP-PrS.

$$\textbf{MFP-PrScost}(\mathrm{PrS}_j) = \sum_{PaS_i} \mathrm{x}_i \ \mathrm{PaScost}(\mathrm{PaS}_i) + T_{p/e,P_j}$$

MFP-PrScost is computed as an ILP problem using the approach of [7], delivering the execution count x_i of the distinct PaS_i plus the transition costs. MFP-PrScost has a minimum and a maximum. The **Transition Cost** $\mathrm{T}_{p/e,P_j}$ is the cost representing overlapping PrScost for the prologue p and the epilogue e in figure 2. These transition costs must be conservative. If no MFP-PrS in the PaS on lower levels of hierarchy is present, the recursive descent stops. The execution count x_i of a PaS is solved according to section 2.2 after the equations for the embedded MFP-PrS including execution count x_i have been propagated to the top level.

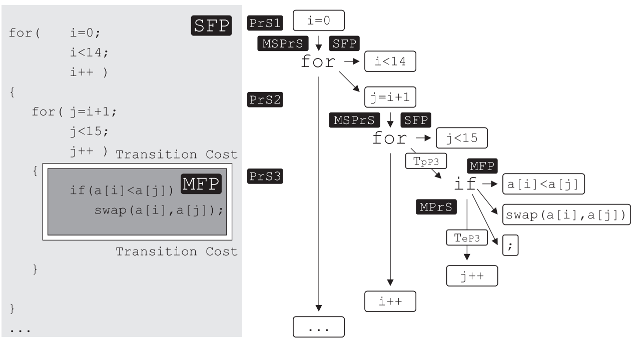

Figure 3. Pseudo SFP-PrS with MFP-PrS

In the previous approach in [13], embedded MFP cost and "cut point" cost were separately analyzed and added to the simulated SFP for every MFP on different levels of hierarchy. We do not lose the dependencies across several levels of hierarchy because the MFP-PrScost equations including the execution count x_i and transition costs are propagated to the top level of the syntax graph instead of adding values in the control flow graph. This generalizes the approach in [13]. As MFP-PrScost is based on PaS, single paths PaS through the MFP-PrS can be analyzed.

Global Cost Calculation For the bubble sort example in figure 3, the recursive cost calculation with the propagation of equations works the following way: We start on the top level of the process. For its execution cost we need the cost of PrS_1 and of the lower levels of hierarchy PrS_2 and PrS_3. After checking the two for loops, the recursive descend finds PrS_3, the $if/else$ MFP-PrS. It only contains leaf nodes. The $\mathrm{MFP\text{-}PrScost}^{if/else}$ of PrS_3 is composed by the cost of the paths $\mathrm{PaS}_{i,3}$ across "control" and "then" or "control" and "else" each of which is delivered by PsE. x_i is their execution count and $T_{p/e,P_3}$ the transition cost. Then we can calculate the cost of PrS_2, the inner for loop shown in figure 2. It is composed by the cost equation of the $if/else$ MFP-PrS, and the MSPrScost of the "*j++*"-PrS and "control"-PrS as these are leaf nodes. The cost equation for PrS_1, the outer for loop, can be given which adds the MSPrScost of the "*i++*"-PrS and "control"-PrS to the cost of PrS_2 and PrS_3.

$$\text{Execution Cost} = \sum_{PaS_{i,3}} \mathrm{x}_i \ \mathrm{PaScost}(\mathrm{PaS}_{i,3}) \ + \ T_{p/e,P_3}$$

$$+ \ \mathrm{MSPrScost}(\mathrm{PrS}_2) \ + \ \mathrm{MSPrScost}(\mathrm{PrS}_1)$$

Even with one level of hierarchy between PrS_1 and PrS_2, their MSPrScost can be delivered by the same PsE because the control flow is given by the program properties. PrScost equations for PrS_3 have been propagated to the top level where the designer can provide functional constraints bounding the x_i of PrS_3 instead of basic blocks according to section 2.2. This finally delivers the execution cost bounds.

There is a one-to-one correspondence between the basic blocks of the syntax graph and the nodes of the hierarchical control flow graph HCFG the syntax graph can be transformed to. For the following examples the HCFG is used to allow an easier modeling of control flow.

2.5. Context Dependent Control Flow

The path analysis approach presented up to this point is based on the identification of input data independent control flow. This improves the estimation accuracy compared to the approach in [7] and the first preliminary SFP analysis approach in [13]. Even MFP segments with input data dependent paths are analyzed with narrower bounds than in the previous basic block based approaches, as long as some segments on the lower levels of hierarchy are SFP-PrS.

In the introduction, we have argued that the designer is often interested in a context dependent process behavior. Here, context is defined to be a subset of input data and/or a subset of possible process states, often called process modes. In each context, only a subset of paths through a program segment can be executed. This potentially means reduced cost bounds which could be exploited for analysis. Global process representation models [14] can support process modes, such that the distinguishable contexts are known for cost analysis. A simple example for context dependent control flow in an ATM switch component is given.

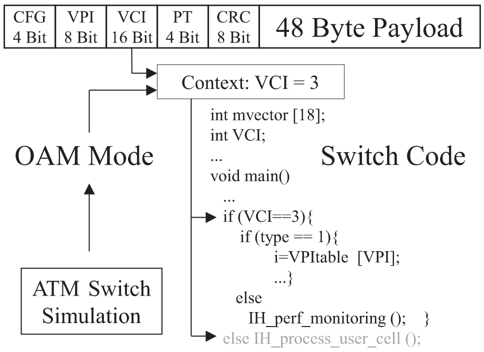

Figure 4. Path selection of the OAM mode

An ATM switch identifies some of the cells in the data cell stream as so called operation and maintenance cells, OAM, which control the ATM connection [3]. These cells do not carry user data so they are irrelevant for data transmission. Figure 4 shows a typical code segment to handle the OAM component of the switch. The control flow graph is shown in figure 5. In this "OAM mode", the shaded *else* program segment in figure 4 cannot be reached. It should not be included in further analysis of the OAM mode while in the "USER mode" only this *else* path is executed. For

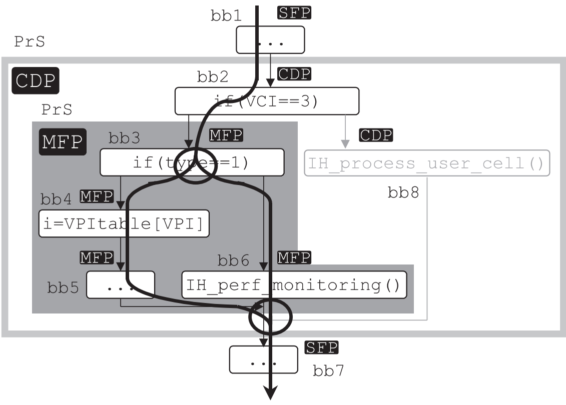

Figure 5. PrS in the ATM switch component

a given context, the "if" node bb2 has a single path only. In other words, the contexts "VCI = 3" corresponding to the OAM mode, and "not (VCI = 3)" corresponding to the USER mode turn an MFP-PrS into a PrS with a single path. We will call such a PrS a **Context Dependent Path** program segment **CDP-PrS**. For analysis of the given context, it is treated like an SFP-PrS. Where this approach is not applicable, the reduced path set of a given context can further be exploited via additional structural and functional constraints [7]. In both cases, context dependent behavior can be analyzed using the same techniques as described before. The same discussion for the gain in accuracy as in [13] applies because longer sequences are achieved than with SFP identification alone. At the transitions between SFP and CDP segments, MSPrS containing both SFP-PrS and CDP-PrS can be defined.

For different modes, SFP-PrS and functional constraints for the remaining MFP-PrS stay the same, while a different block of CDP-PrS can be extracted from the MFP-PrS. This way, average cases given as artifical modes can tighten the wide intervals. Stochastic or probabilistic distribution of input data could be considered using according cost functions and convolutions for the PrS-cost.

3. Architecture Modeling

A program segment execution PsE for the cost determination of a PrS uses one of the following two techniques:

Instruction Cost Addition ICA The instruction or statement execution costs in a basic block or PrS are added. We do not need input data. Host tracing is used while execution costs are taken from a table. This is a very computation time efficient approach. Instruction execution cost c_i can be dependent on input data. A popular example is a shift-and-add implementation of a multiplication in a processor delivering an interval for c_i. So in the tables, minimum and maximum instruction execution cost can be considered leading to an interval for the PrS cost delivered by ICA.

Program Segment Simulation PSS The basic block or PrS is simulated using known input data and a cycle true processor model [2] which can exactly deliver processor timing or power consumption. This can be any well established, off-the-shelf processor simulator provided by the processor vendor. Processor evaluation kits implemented in hardware have been successfully used for timing or power measurement with a logic state analyzer and automatic result back annotation. As an example for PSS that delivers the execution cost of the PrS, a StrongARM simulator core is combined with the DINERO III cache simulator delivering both instruction and data cache behavior. Source codes have been recompiled to one simulator. Architecture modeling regarding timing and the energy dissipation model is derived from [8] and [11]. Data rates are derived from the amount of data produced or consumed on a path and its execution count from section 2.2.

The major improvement to the architecture modeling of the first SFP analysis approach in [13] is the possibility to integrate off-the-shelf processor simulators and emulators. This enables us to determine execution cost intervals for several target architectures.

4. Experiment

Packet Receiver The approach is applied to a process that reads a packet and loads a picture as presented below. If the picture is addressed to the component, it performs a filter. A pseudo code description is given below.

```
89: header = receive(INPUT, HEADER_SIZE);
    for all pixels                   /* Size mode    */
        picture[y][x] = receive(INPUT, 1);
122:if(address == MY_ADDRESS) {      /* Ann. address */
124:    for all pixels {
            for a 3*3 pixel window {
143:            if(without_center)    /* Ann. center  */
                    average = sum/8;
                else average = sum/9;
151:            if(abs(picture[y][x]-average)>threshold)
                    send(OUTPUT, average, 1);
                else send(OUTPUT, picture[y][x], 1);
      }  }  }
```

In table 1, execution cost intervals with respect to latency time, power consumption and data rates without picture size or address match mode are given. The intervals as well as the path classification are given for every PrS that is referenced by the line number it is starting with. Due to the loop bounds for each context given by the packet size bounds in the received header, we know the minimum and maximum number of pixels leading to a CDP in line 124. SFP segments and CDP segments are merged into MSPrS, so they may not be visible in the results. The results for the complete process are given in the last line.

Line+	PrS type	Latency ms	Energy mWs	Sent bytes	Received bytes
89	SFP	[4.92,38.0]	[2.0,8.5]	[0,0]	[6197,25045]
122	MFP	[413ns,2475ns]	[50nWs,178nWs]	[0,0]	[0,0]
124	CDP	[39.5,329]	[17.5,72.6]	[0,0]	[0,0]
143	MFP	[1.54,131]	[0.65,14.7]	[0,0]	[0,0]
151	MFP	[16.7,182]	[2.85,20.4]	[0,24393]	[0,0]
$\sum$cost	-	[4.955,680.847]	[2.099,116.211]	[0,24393]	[6197,25045]

Table 1. Cost $[c_{i,min}, c_{i,max}]$ without modes

In table 2, different modes are explored. In the first three lines, picture sizes are derived from process modes. Address and luminance calculation modes follow. Table 1 can be found in line 3. We notice that known process modes lead to tighter, but context dependent cost intervals for the process that can be used for formal process representation supporting process modes on higher levels [14].

Modes/Context	Latency ms	Energy mWs	Sent bytes	Received bytes
Small Picture	[4.955,66.71]	[2.099,24.61]	[0,5865]	[6197,6197]
Large Picture	[19.24,680.8]	[8.474,116.2]	[0,24393]	[25045,25045]
No Size	[4.955,680.8]	[2.099,116.2]	[0,24393]	[6197,25045]
Small+Address	[38.49,63.62]	[21.03,23.61]	[5865,5865]	[6197,6197]
Large+Address	[264.6,572.0]	[97.3,106.5]	[24393,24393]	[25045,25045]
No Size+Address	[38.49,572.0]	[21.03,106.5]	[5865,24393]	[6197,25045]

Table 2. Cost $[c_{i,min}, c_{i,max}]$ with modes

5. Conclusion

Process timing and power consumption can be highly context dependent. Process modes are introduced to distinguish contexts with significantly different execution cost. An existing symbolic analysis approach is extended to calculate global program cost intervals and capture context dependent cost intervals of single processes. Major improvements to path analysis and architecture modeling techniques are presented. The results demonstrate a significant improvement in execution cost analysis precision.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, GB, 1988.

[2] T. M. Conte and C. E. Gimarc. *Fast Simulation of Computer Architectures*. Kluwer Academic Publishers, 1995.

[3] A. Doboli, J. Hallberg, and P. Eles. A simulation model for the OAM functionality in ATM switches. Technical report, Linköping, 1995.

[4] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of Language, Compilers and Tools for Embedded Systems*, 1998.

[5] J. Gong, D. Gajski, and S. Narayan. Software execution from executable specification. *The Journal of Computer and Software Engineering*, pages 239–258, 1994.

[6] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1998.

[7] Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[8] J. Montanaro. A 160-MHz, 32-b, 0.5W CMOS RISC microprocessor. *IEEE Journal of Solid State Circuits*, pages 1703–1714, Nov. 1996.

[9] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing scheme. In *Proceedings of Real-Time System Symposium*, pages 72–81, 1990.

[10] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):160–176, 1989.

[11] V. Tiwari, S. Malik, and A. Wolfe. Instruction level power analysis and optimisation of software. *VLSI Signal Processing*, pages 1–18, 1996.

[12] F. Wolf and R. Ernst. Execution cost interval refinement in static software analysis. Technical report, Technical University of Braunschweig, 2000.

[13] W. Ye and R. Ernst. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings International Conference on Computer-Aided Design (ICCAD '97)*, pages 598–604, San Jose, USA, 1997.

[14] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Sixth International Workshop on Hardware/Software Co-Design*, pages 9–13, Seattle, 1998.