Worst Case Execution Time Analysis for Synthesized Hardware

Jun-hee Yoo
ihavnoid@poppy.snu.ac.kr
Seoul National University,
Seoul, Republic of Korea

Xingguang Feng
fengxg@poppy.snu.ac.kr
Seoul National University,
Seoul, Republic of Korea

Kiyoung Choi
kchoi@snu.ac.kr
Seoul National University,
Seoul, Republic of Korea

Eui-young Chung
euiyoung.chung@samsung.com
Samsung Electronics,
Yongin, Republic of Korea

Kyu-Myung Choi
kmchoi@samsung.com
Samsung Electronics,
Yongin, Republic of Korea

Abstract - We propose a hardware performance estimation flow for fast design space exploration, based on worst-case execution time analysis algorithms for software analysis. Test cases on some real-world applications show that our flow provides a tight upper bound of the execution time, and many useful hints to the designer.

I. Introduction

As the Moore's law continues to apply to the embedded system industry, systems become more complex every year. Unfortunately, designer productivity doesn't continue to grow as fast as system complexity, making design cost grow rapidly every year. Especially, failing to meet the given constraints (cost, performance, power consumption, etc) in late stages of design and repeating the whole design cycle can be catastrophic. Therefore, accurately estimating the final design in early design stages becomes more and more important.

Although many recent designs try to rely more on software, many modern embedded systems use hardwired logic for performance-critical portions of the given algorithm, which is mainly because hardwired logic provides performance higher than software. However, the performance is achieved at higher manufacturing cost. To find an optimal design in early design stages, it is crucial to estimate accurately the performance and cost of the hardware implementation of a given algorithm.

Since it is difficult and time-consuming to estimate the final design manually considering the exceedingly large design size of modern systems, there has to be some automated method. This problem has been a research issue for more than 10 years and there exist many hardware estimation and analysis tools, along with tools that generate hardware implementations from behavioral models. However, most of the previous approaches evaluate the hardware's performance based on simulation.

Simulation-based estimation flows have many limitations. As the design size grows, simulation speed gets slower. Moreover, the number of test cases needs to increase exponentially as the system size grows, since there can be many corner cases. Even with all those efforts, there's no way to guarantee that every corner case has been tested, so there always exists some possibility of missing tests for worst case performance.

In this paper, we present a hardware estimation flow based on static performance analysis techniques. The proposed estimation flow translates a given C function into CDFG, synthesizes a hardware structure from the CDFG, and statically analyzes the generated hardware to estimate the performance. Although the proposed estimation flow analyzes the hardware implementation statically to obtain worst case performance, it also does simulation-based estimation for average case performance.

II. Related Work

A. Behavioral Synthesis and Estimation

Behavioral synthesis (commonly known as high-level synthesis or architectural synthesis)[1] is a core part of our estimation flow. It has been a research topic for more than a decade and there are some real-world products that perform behavior-level synthesis on C-based input description. Catapult C-synthesis[2] from Mentor Graphics, an example of such product, generates synthesizable HDL code out of some restricted form of C/C++ code. The user can explore the hardware design space by interactively specifying how to implement some portions of codes, such as by setting a loop to be always unrolled, or by modifying the resource constraints. Additionally, the user can select what kind of external interface the hardware will have.

Forte's Cynthesizer[3] is another such synthesis flow which starts from SystemC behavioral description. By using Cynthesizer, users can generate many RTL descriptions of an algorithm with different constraints, and choose the appropriate one for the whole design.

However, these commercial tools focus on fast design implementation by automating RTL coding, neglecting tight worst-case execution time (WCET) analysis techniques. The user can figure out how many cycles it takes by simulation, or by figuring out how many times a loop might iterate in the worst case, and multiply it by the worst-case execution cycles of the loop body.

B. Static Estimation

There are many approaches to static software analysis. Our work is inspired from those software estimation flows.

The Cinderella system[4] is a static approach to estimating the performance of real-time software. The goal

0-7803-9451-8/06/$20.00 ©2006 IEEE.

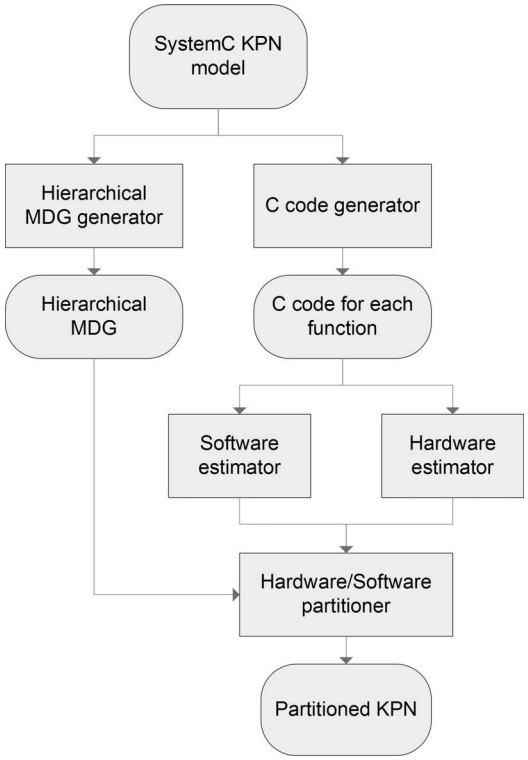

Figure 1. The KPN-based SoC design space exploration flow.

of this approach is to estimate the worst-case execution time (WCET) of a program. Based on basic block analysis, the authors generate a series of linear constraints about the execution counts of each basic block. Thus, the problem of finding the WCET of a program is reduced to an integer linear programming (ILP) problem. Together with the delay of each basic block, the WCET is obtained by solving an ILP with the objective of maximizing the total delay. Our static hardware estimation flow is based on the idea from this work.

SymTA/S[5] is another work based on static analysis. It uses many approaches from real-time analysis theory to the system level. The task execution times are obtained by using the approaches from Cinderella. Furthermore, it extends the Cinderella's work to system level WCET analysis. Users can explore the design space of the given system specification by trying different schedules and different implementations.

There have been many researches on WCET analysis for various microprocessor architectures, such as microprocessors with cache, or microprocessors with branch prediction[6][7]. However, the authors are not aware of any prior research on applying WCET analysis on synthesized hardware.

III. Application of the Estimation Flow

The hardware estimation flow described in this paper is based on the hardware estimator of our prior work[8]. Figure 1 illustrates our SoC design space exploration flow. Our prior work is an interactive SoC design space exploration tool, which uses a KPN-modeled SystemC description as its starting point. The steps of our SoC design space exploration tool is as follows:

1) The input KPN model is translated to an HMDG (hierarchical module dependency graph) consisting of many MDGs. (module dependency graphs) and the behavior of each node of the MDGs is extracted into C code.
2) The C code is sent to the hardware estimator and software estimator, which estimates the execution time and implementation cost of each function.
3) Based on the estimation information, the hardware/software partitioner decides whether an MDG node should be implemented in hardware or software. It may report mixed implementation of an MDG, if some of its sub-MDGs are implemented in hardware and the rest is implemented in software.
4) Since there can be many different implementations of an algorithm, the optimal system implementation may not be obtained unless we examine various designs with various constraints. In our case, the partitioner runs the hardware estimator multiple times on the same input C function, each with different cost constraints. However, since just estimating the hardware design for all possible constraints for each functional block of the system takes infeasible amount of time, our partitioner uses a heuristic algorithm to decide which subset of constraints will be tried for a given function. However, it still takes a huge amount of time with a naive approach such as simulation, and thus requires a fast hardware estimation method.

IV. Hardware Estimation Flow

A. Hardware Model

We assume that the generated hardware will be connected using a bus, and communicate using a DMA controller. We also assume that a hardware block will have its own private memory space, which can be accessed by other processors via the bus while the hardware block is idle.

We assume the system operates as follows:

1) The caller (microprocessor or another hardware block) checks to see if the hardware block is available, and acquires the control over the generated hardware. This control can be implemented using mutex.
2) The caller uses the DMA controller to transmit all data required for the hardware to complete its execution to the hardware's private memory space.
3) The caller writes a 'start' command on the hardware's 'command' register, which starts the execution of the hardware.
4) After the execution completes, the caller uses a DMA controller to fetch the computed result from the hardware's private memory.

Figure 2 shows the hardware model we assumed. We used this model because of the following reasons:

1) The hardware estimation flow is expected to estimate hardware blocks that will run as a microprocessor accelerator, thus, assuming a DMA using general buses for communication will be appropriate.

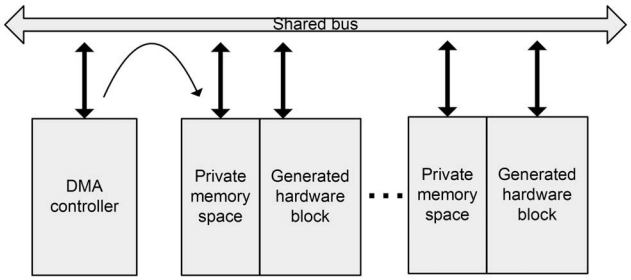

Figure 2. Hardware model.

2) We assumed a private memory block because it's difficult to predict how long a bus access will take. If we assumed shared memory space, many kinds of hard-to-predict latencies such as arbitration has to be added, and it will be difficult to predict the worst-case execution time correctly.

Although the model limits the generality of the synthesized hardware, we believe that the proposed approach can be applied to a more general model with proper adjustment.

B. *Overview of the analysis flow*

Figure 3 illustrates our hardware estimation flow. The input of the current estimation flow is a C function with some restrictions on the constructs, including pointer access and some control flow statements such as *goto* or *break*. First, the input code is translated into an in-house control-flow data graph (CDFG) implementation[9]. Next, many target-independent optimizations, such as common subexpression elimination and constant propagation, are applied to the CDFG. The optimized CDFG is then scheduled, and appropriate hardware resources are allocated to meet the hardware constraints given by the user or some other tool. This generates a CDFG that's ready to be synthesized into hardware. These steps are a typical behavioral synthesis flow.

However, no hardware is generated after this step. The 'synthesized' CDFG, which contains cycle-accurate scheduling information, can be either analyzed statically or simulated. The analysis (or simulation) result provides a feedback such that the user can apply different constraints to the CDFG and obtain different results.

Although this paper focuses on static analysis of worst-case execution, simulation-based analysis is still useful for analyzing the average-case execution cycles. The CDFG simulator does cycle-accurate simulation for obtaining the performance for some test case. Test cases are generated by extracting the input data applied to the function. The testbench generator library gets linked to the original C code, to log the C code's input data to be used later as the input test vector for simulation.

The following two sections describe the two blocks - the constraint extractor and the CDFG extractor - which are tightly related to static analysis.

C. *Constraint Extractor*

The constraint extractor analyzes the C code's structure,

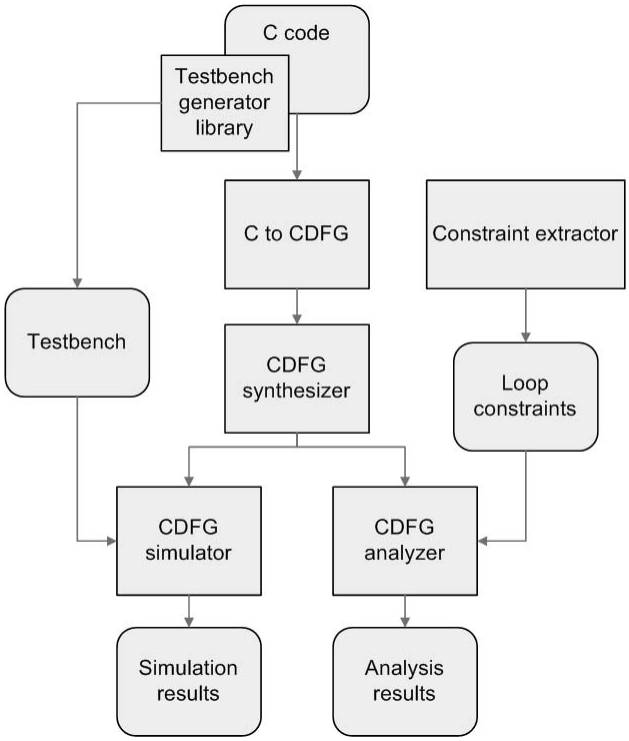

Figure 3. Hardware estimation flow.

```
int some_function(int a, int data[1200]) {
    /*##constraints
    loop1 < 1200;
    b1(true) < b2(true);
    */
    int i;
    for(i=0; i<a; i++){ //##label:loop1
        if(data[i] == TYPE_A) { //##label:b1
            /* .... some code .... */
        }
        //##label:b2
        else if(data[i] == TYPE_B) {
            /* .... some code .... */
        }
    }
    /* .... some code .... */
}
```

Figure 4. An example code with user constraints inserted.

and generates constraints that are always fulfilled regardless of the input condition. Currently, our constraint extractor looks for trivial loops with fixed number of iterations by using the SUIF compiler infrastructure's code analyzer.[10] Although our constraint extractor looks for trivial constraints only, this can be further improved in our later versions of the flow. Since constraint analysis has been an active research field, we believe that we can integrate most of these approaches to our flow.[11]

Even with the most sophisticated analysis algorithms, there can be many constraints that are difficult to determine automatically, or that are input-dependent. In this case, the user may add constraints by annotating C code. The constraint extractor extracts all constraints including the user-specified constraints, and sends them to the CDFG analyzer.

Constraints can be specified as a number of equations, using variables as the execution count of the corresponding

control flow. Figure 4 shows an example function with the user-given constraints added in. The user can add simple constraints such as the maximum number of iterations a loop will run (line 3). Moreover, the user can add complicated dependencies between many execution counts, such as the relationship between two branches' execution paths (line 4).

D. Static CDFG Analyzer

The static CDFG analyzer generates a large list of integer linear programming (ILP) constraints from the synthesized CDFG. The constraints are then solved using an ILP solver. For our flow, we have used GLPK[12], an open-source ILP solver. Since we are interested in the worst case performance, the ILP solver is invoked to find the worst case.

The method of generating the ILP is adopted from software estimation flows[13]. Although the approach has been originally developed for static software analysis, it is also possible to apply this technique to CDFG analysis, since CDFGs also represent an algorithm's operations and control flow.

The total execution time can be modeled as:

$$\sum_{i}^{N} c_i x_i$$

where N is the total number of basic blocks, c_i is the execution time of the basic block, and x_i is the number of times that the basic block is executed. This is the object value that must be maximized when solving the ILP. Since c_i is the number of cycles a basic block takes to execute, it is determined when the hardware is synthesized.

Constraints by the control path is generated by the fact that for each basic block, the number of times the control enters the basic block equals to the number of times the control exits the basic block. However, unlike the typical 'flat' basic block graph structure, our synthesis flow represents the control flow by using hierarchical nodes which contains many basic blocks. This approach can be found from some other high-level synthesis flows[14].

Our control flow representation contains two kinds of hierarchical nodes - loop nodes which represents *do-while* loops, and condition nodes which represents *if-then-else* statements. A loop node contains one basic block which represents the body and the condition expression of the loop, and a condition node contains three basic blocks where each of them represents the condition block, true block and the false block. Each basic block can contain other hierarchical nodes, and the whole function is represented as a single basic block.

We modified the control path modeling as the following: for all condition nodes C_i, if we let x_i be the execution count of that node, and $x(true)_i$ and $x(false)_i$ be the execution count of the 'true' basic block and the 'false' basic block of that condition node, a condition node can be modeled as:

$$x_i = x(true)_i + x(false)_i$$

For loops, since we are modeling *do-while* loops, we generate the constraint that the loop body should execute more or same times than the loop node itself. This can be modeled as:

$$x_i \leq x(body)_i$$

Additionally, all nodes within a basic block has the same execution count. Therefore, the constraint for that should also be added.

For example, the constraints by the control path of the example on figure 4 is generated as:

$$body = 1$$
$$loop1 = body$$
$$loop1 \leq loop1(body)$$
$$b1 = loop1(body)$$
$$b1 = b1(true) + b1(false)$$
$$b2 = b1(false)$$
$$b2 = b2(true) + b2(false)$$

where $body$ is the execution count of the function itself, and the other terms equal to the execution count of the corresponding control block annotated on the code.

The final ILP formulation is generated by adding user-specified constraints and automatically extracted constraints. For the example on figure 4, the following user-specified constraints can be added:

$$loop1(body) < 1200$$
$$b1(true) < b2(true)$$

In order to make the static estimator give accurate results, the execution counts of all loops in the CDFG have to be known prior to performing static estimation. The loop information generated by the constraint analyzer and constraint extractor is used in this stage. However, in some cases, the user may fail to specify all required loop information, and that will result in failing to find tight upper bound of execution time. In that case, users can incrementally improve the analysis results by adding more constraints on performance-critical code first. Additionally, the user can compare the static analysis results with the simulation results to see if there are unacceptably loose estimation results.

Currently, the static analyzer can only analyze the execution time. Other performance metrics such as energy consumption per execution are obtained via simulation.

V. Test Case and Experimental Results

We have done experiments using two real-world multimedia applications: the h.263 video encoder, and the Karplus-Strong algorithm,

A. H.263 Encoder

We have experimented with the h.263 encoder, a widely adopted video compression application. We analyzed the two most heavily used functions: SAD_Macroblock and Quantize.

Figure 5 and 6 show the area-performance tradeoff of SAD_Macroblock function and Quantize function, respectively. We have used different number of ALUs as the area constraint of the implementation.

Figure 7 and 8 show the simulation results of the hardware implementation with the maximum number of

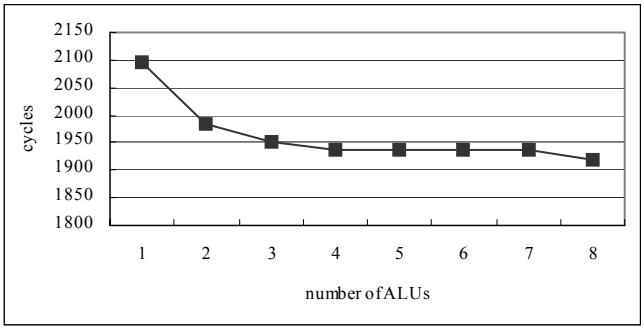

Figure 5. Area-performance tradeoff of SAD Macroblock.

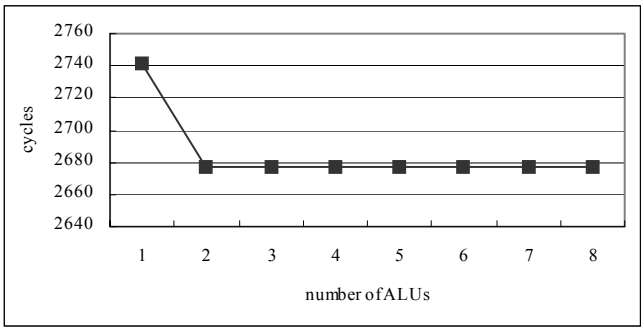

Figure 6. Area-performance tradeoff of Quantize.

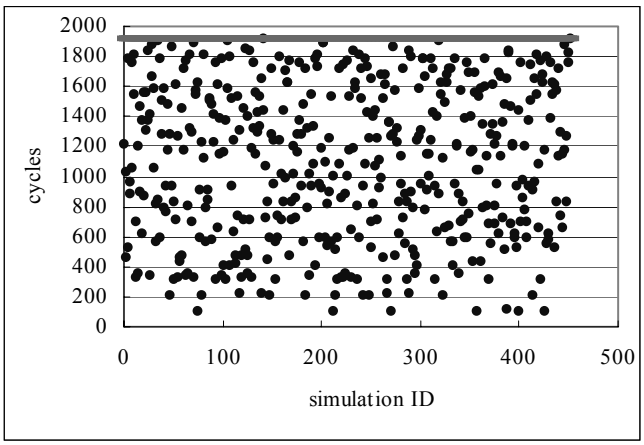

Figure 7. Simulation result of SAD Macroblock.

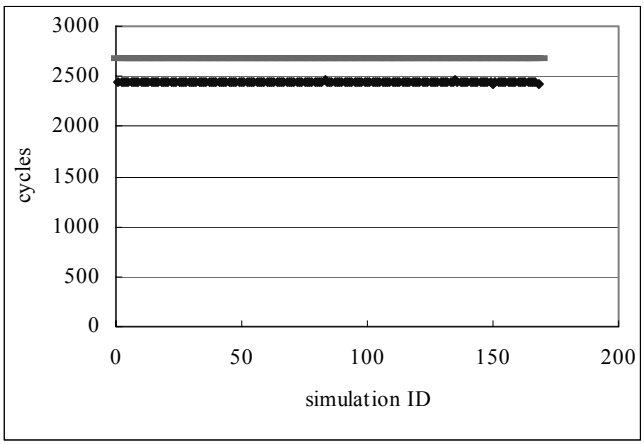

Figure 8. Simulation result of Quantize.

```
#define NUM_SAMPLES 1024
#define COMB_FILTER(cn,cn1,v0,vn,vn1) ¥
    ((((v0)-MID)*NSF + ((vn)-MID)*(cn)
    +((vn1)-MID)*(cn1)
    /256) + MID)

void karplus_strong(int cn, int cn1,
              unsigned int n, short block[NUM_SAMPLES],
              short blockprev[NUM_SAMPLES]){
    int i;
    for (i = 0; i < n; i++){
        block[i] =
            COMB_FILTER(cn, cn1, MID,
                blockprev[NUM_SAMPLES + i - n],
                blockprev[NUM_SAMPLES + i - n - 1] );
    }
    block[n] =
        COMB_FILTER(cn, cn1, MID, block[0],
            blockprev[(NUM_SAMPLES - 1)] );
    for (i = n + 1; i < NUM_SAMPLES; i++)    {
        block[i] =
            COMB_FILTER(cn, cn1, MID, block[i - n],
                block[i - n - 1]);
    }
}
```

Figure 9. The Karplus-Strong code for this experiment.

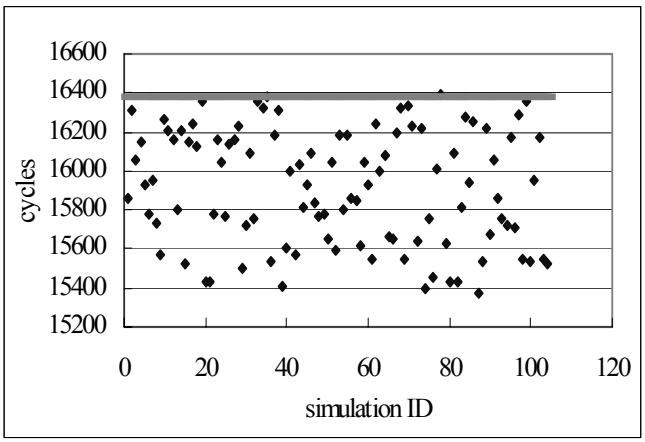

Figure 10. Simulation result of Karplus-Strong.

functional units. The X axis represents the simulation run ID number, and the Y axis represents the number of cycles. We have done 451 simulations for SAD_Macroblock and 168 for Quantize, using different real-world testbenches. From the figures, we can make sure that the execution time obtained by simulation is always within the upper bound obtained by static analysis.

For SAD_Macroblock, the worst case execution time obtained by the simulation exactly matches the upper bound. However, for Quantize, the worst case execution time by the simulation is 2,458 cycles, whereas the static estimation result gives 2,677 cycles. Through a careful analysis, we have found that our simulation-based estimation has missed running the h.263 encoder with different modes, which takes exactly the same cycles as the static analysis reports.

Even though ILP is known to be an NP-complete problem, the execution of the ILP solver is done almost instantly on an average workstation. (Intel Xeon 2.4GHz). This is because the ILP problem is reasonably small - the set of equations from Quantize has 49 variables and 55

equations and that from SAD_Macroblock has 54 variables and 39 equations.

B. Karplus-Strong

Karplus-strong[15] is a method of generating synthesized sound waveforms, which is based on physical modeling of a hammered or plucked string, or some type of percussion. We have used our implementation of Karplus-strong, which does double-buffering on the output buffer. Figure 9 shows the C code of our implementation.

By observing the C code, it's easy to find the constraint that the two *for* loops have dependencies on loop iteration numbers - the sum of the number of iterations on the two loops are 1,023. These kind of constraints can be added by formulating it as an ILP. In this case, the estimated worst case was 16,385 cycles.

However, when using the simple method - multiplying the number of worst-case iterations to the cycles that takes to execute the loop body, we have to use the worst case for both loops, and the two loops are assumed to iterate 1,023 times. In this case, the estimated worst case is 31,731 cycles.

Figure 10 shows the execution cycles of the simulation. The X axis represents the simulation run ID number, and the Y axis represents the number of cycles. The worst case execution cycle of the simulation equals to our analysis result. This shows that the analysis method of our flow gives a tight upper bound.

VI. Conclusion and Future Work

In this paper, we present a hardware estimation flow that can be used for design space exploration. The test cases and experiments show that our flow can help the designer to understand many possible issues that can happen in the hardware implementation.

We summarize our contribution as:

1) Presenting a hardware estimation flow based on static analysis of the execution pattern, and
2) presenting a method of adding complex execution path constraints to a C function, and using them for worst-case execution analysis.

However, our estimation flow has some more points to improve. Our estimation flow made many on the restrictions made in the input C code. These limitations have added much labor on modifying the reference code to work with the estimation flow. However, we expect to remove most of these limitations in our future work.

Some additional features that might help the users of this flow can be static energy consumption analysis. By predicting a reasonable upper bound of energy consumption, it would be possible to make a reasonable power budget.

Additionally, the hardware model that we used assumes a local buffer memory, so that there would be no unpredictable memory accesses delays. We believe this can be improved by adding information about the external bus into the static analyzer, and generate the appropriate ILP formula based on those information.

References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994

[2] Catapult C synthesis, Mentor Graphics, http://www.mentor.com/products/c-based_design/

[3] Forte Cynthesizer, http://www.forteds.com/products/cynthesizer.asp

[4] Cinderella 2.0 http://www.princeton.edu/~yauli/cinderella-2.0/

[5] SymTA/s, http://www.symta.org/

[6] S. Kim, S. Min, R. Ha, "Efficient worst case timing analysis of data caching", *In Proceedings of the IEEE Real-Time Technology and Applications Symposium,* Pages 230 - 240, June 1996

[7] A. Colin, I. Puaut, "Worst case execution time analysis for a processor with branch prediction", *Real-Time Systems.* Vol. 18, no. 2, Pages 249-274. Kluwer Academic Publishers, 2000,

[8] Y. Ahn et al, "An Interactive Environment for SoC Design Starting from KPN in SystemC", Global Signal Processing Expo., Oct. 2004

[9] Control Data Flow Graph Toolset, http://poppy.snu.ac.kr/CDFG/

[10] The SUIF 1.x compiler system, http://suif.stanford.edu/suif/suif1/index.html

[11] C. Healy and D. Whalley, "Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints", *In Proceedings of Fifth IEEE Real-Time Technology and Applications Symposium*, pages 79-88, 1999

[12] GNU Linear Programming Kit, http://www.gnu.org/software/glpk/glpk.html

[13] Y. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," *In Proceedings of the 16th IEEE Real-Time Systems Symposium,* pages 298-307, 1995

[14] S. Gupta, N. Dutt, R. Gupta, A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing computer transformations", *In Proceedings of the 16th International Conference on VLSI Design,* pages 461-466, 2003

[15] Karplus-strong string synthesis, from Wikipedia http://en.wikipedia.org/wiki/Karplus-Strong_string_synthesis