Embedded program timing analysis based on path clustering and architecture classification

R. Ernst, W. Ye
Technische Universität Braunschweig, Institut für Datenverarbeitungsanlagen
Hans-Sommer-Str. 66
38106 Braunschweig
ernst@ida.ing.tu-bs.de, ye@ida.ing.tu-bs.de

Abstract

Formal Program running time verification is an important issue in system design required for performance optimization under "first-time-right" design constraints and for real-time system verification. Simulation based approaches or simple instruction counting are not appropriate and risky for more complex architectures in particular with data dependent execution paths. Formal analysis techniques have suffered from loose timing bounds leading to significant performance penalties when strictly adhered to. We present an approach which combines simulation and formal techniques in a safe way to improve analysis precision and tighten the timing bounds. Using a set of processor parameters, it is adaptable to arbitrary processor architectures. The results show an unprecedented analysis precision allowing to reduce performance overhead for provably correct system or interface timing.

1 Introduction

Program running time determination is an important issue in embedded system design required for performance optimization. In hard real time systems, program timing is part of the system functionality and incorrect timing assumptions can have disastrous consequences. Formal running time analysis is, therefore, highly desirable. Rather than a single value, formal analysis techniques provide upper and/or lower running time bounds. One reason is that program timing is data dependent, other reasons are imperfect program path analysis (in principle a known undecidable problem) and computer architecture impacts which are very computation expensive to analyze. While data dependent program timing is real and leads to unavoidable differences in lower and upper timing bounds, inaccuracies in path analysis and architectural modeling are artifacts leading to wider bounds than can occur in reality. To assure correct minimum and maximum timing of the design, bounds must always be conservative, i.e. maximum bounds must be higher and minimum bounds must be lower than the real timing bounds. Therefore, inaccurate bounds are expensive because they require to design systems with higher performance to meet the maximum time or rate constraints. As an example, if an analysis has 50% inaccuracy, the designer must provide 50% more performance than needed in reality to be able to verify correct timing using timing analysis. The problem is that 50% inaccuracy of the analysis is already pretty good in practice if all effects are taken into account and if more complex architectures are considered.

We developed a classification of orthogonal architecture and program properties which covers all practical processor designs. These properties are used to assign on individual analysis technique to each class which is highly accurate for the respective class. While these individual analysis techniques themselves are taken from practice or from literature, this paper shows for which classes they can be applied in a safe way.

In previous work on timing analysis a single general technique has been applied to the problem. Specific program or architecture properties have hardly been exploited, except for few instances such as cache properties. A detailed discussion of program path characteristics and target architectures reveals that bounds can be much closer if the analysis approach uses architectural knowledge. We also show that program path analysis must consider certain characteristics, such as pipelining, to be reliable.

In this paper, we consider program path characteristics as well as architecture properties to maximize the accuracy of timing bounds thereby minimizing analysis cost. The paper is organized as follows. Section 2 focuses on program path analysis. It starts with a problem definition and a summary of previous work. Then, a path classification is introduced which isolates input data dependent from independent path segments. In section 3 the hardware architectural impacts on timing analysis are described and an architecture classification that fits the program path classification is introduced. In section 4 and 5 we present our SYMTA approach which is based on these classifications and its implementation. Finally, in section 6 we summarize the experimental results for the SYMTA approach.

2 Program Path analysis

2.1 Problems and previous work

For path analysis techniques [10] [7], a program is divided in **basic blocks**.

Definition 1 *A* **basic block** *is a program segment which is only entered at the first statement and only left at the last statement [1].*

Function calls are considered as single statements. Any program can be partitioned into disjoint basic blocks. The program structure is represented on a directed program flow graph with basic blocks as nodes. Fig. 1 shows an example.

For each basic block the worst or best case execution time for each basic block is determined. Then, a longest or shortest path analysis on the program flow graph is used to identify lower and upper running time bounds.

This procedure does not yet provide sufficient accuracy. For acceptable program timing analysis one must identify **feasible paths** through a program.

Definition 2 *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, i.e. leading from the first to the last basic block of a program.*

Definition 3 *A* **program path segment** *is a segment of a program flow graph.*

This definition implies a *hierarchy* of program path segments. However, not all paths in the program flow graph represent feasible program paths.

Definition 4 *A* **false program path** *is a path in the program flow graph which cannot be executed under any input condition.*

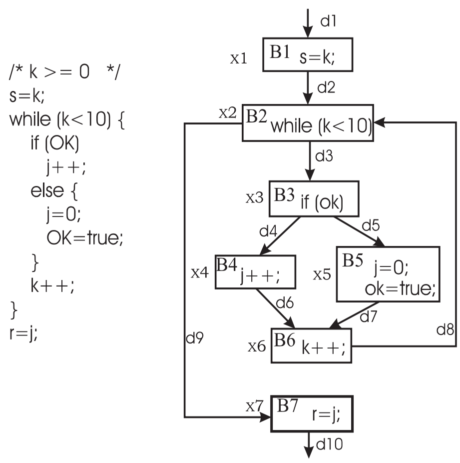

Figure 1: An example program flow graph

False path identification is mandatory for programs with loops since loops correspond to cycles in the flow graph which lead to an infinite number of potential paths. The approaches by Mok [9], Puschner and Koza [12], Park and Shaw [11] require iteration bounds for all loops in the program which the user must provide by loop annotation. The approach by Gong and Gajski [4] can identify more *false paths* by specifying the branching probabilities.

While making formal analysis feasible, loop bounding alone is not sufficient for accurate timing analysis. Nested loops are often interdependent, function timing is invocation dependent and conditions depend on each other. These dependencies can be rather complex as shown in the example in fig. 1, taken from [7]. Here, block $x5$ is executed at most once which would not be detected by program flow graph analysis alone. Therefore, as a second step, 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 [11]. 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 [5], [7]. Here, the user provides linear (in)equations to define false paths. For fig. 1, a simple equation would be "$x5 \leq x1$" meaning that node $x5$ is at most executed as often as node $x1$. To evaluate these (in)equations, Malik et al. map the upper and lower bound timing identification to two **ILP (integer linear programming)** optimization problems, the one optimizing for the shortest program running time, the other one for the longest running time.

2.2 Execution time model

The standard execution time model is the **sum–of–basic–blocks** model. Let a program consist of N basic blocks with c_i execution count of basic block bb_i and t_i execution time of basic block bb_i, $i = 1, 2, ..., N$. Then, the **sum–of–basic–blocks** model assumes for the total program execution time T [7]:

$$T = \sum_{i=0}^{N} c_i \times t_i$$

This model assumes that all executions of a basic block take identical time. However, data dependent instruction execution times and superscalar or superpipelined architectures with overlapped basic block execution have widely varying basic block execution times, with a substantial effect the overall execution time, as demonstrated in [13]. For these common architectures, the **sum–of–basic–blocks** model cannot provide close bounds, but must be pessimistic to be correct.

For higher accuracy, basic block sequences and data flow must be considered. This shall be called the **sequence–of–basic–blocks** model.

2.3 Path classification and analysis

Complete **sequence–of–basic–blocks** analysis requires exhaustive path analysis in the worst case and therefore, must be considered infeasible. It is, however possible to exploit program properties to simplify path analysis. The first observation is that many embedded system programs or at least parts of such programs have a single feasible program path. An FIR filter is a simple example and an FFT is a more complex one. In other words, 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. This shall be called **single feasible path (SFP)** property. The current analysis approaches give different lower and upper timing bounds 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 user interaction for SFP programs. In contrast, simulation would choose the one correct path for any input pattern without further user interaction. Fig. 2, a part of an FFT algorithm, shows that the *if/else* and the inner *while* loop are very difficult to annotate, although the program is SFP. Obviously, most practical systems contain non-SFP parts. These parts shall have the **MFP property (multiple feasible paths)**. Interestingly, a look at

```
n = nn << 1;
j = 1;
for (i=1; i<n; i+=2) {
  if (j > i) {
     swap(data[i],data[j]);
     swap(data[i+1],data[j+1]);
  }
  m = n >> 1;
  while ((m >= 2)&&(j > m)) {
        j  -= m;
        m >>= 1;
  }
  j += m;
}
```

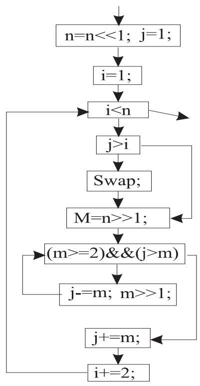

Figure 2: An example of the SFP property

embedded system algorithms reveals that at least for data dominated applications or applications with a signal processing component or complex computation, we find long program path segments with SFP property. So, it seems worth while to have a closer look at SFP exploitation. If we would be able to isolate SFP and MFP parts then we could try to exploit the SFP property in all programs.

To apply different techniques to MFP and SFP parts, we need some disjoint program partitioning. This is the first difficulty since the set of program paths is not disjoint. The solution is a program partitioning which will be explained later. For the moment, let us assume that SFP and MFP program segments are not overlapping.

So, the first parameter for classification refers to the application program and distinguishes MFP and SFP program segments.

2.4 Data flow analysis

Data flow analysis is relevant to program timing when the data access timing is address dependent. One example are architectures with data caches, the other important case are interleaved memory banks. The SFP property does not cover this case. However, if for a given (non false) program path segment,

- all instructions or statements always access the same data variables and array indices under whatever input conditions
- and the program path segment is SFP

then the data access sequence is unique for this part of the program. This shall be the **single data access sequence (SDS)** property. Examples for programs with SDS are filter or FFT algorithms. For all other cases we use the term **multiple data access sequences (MDS)**[1]. So, data access is the second classification parameter.

2.5 Multitasking and context switch impact

Static scheduling and context switch as in many smaller or high-performance signal processing systems can be treated with the same path analysis techniques.

Static priority scheduling (e.g. RMS) or dynamic scheduling (e.g. EDF) are hard to analyze if the process executions are interdependent. This is the case for cache architectures. There are, however, efficient approaches to improve cache performance and predictability, e.g. [8] and [6], which can directly be applied to our case. Then, the program path segments must be analyzed to identify potential remaining cache misses due to context switch. This is not a major limitation but has not been investigated in this project.

[1]Note that SDS could be defined such that it would not require SFP as a necessary condition, but an exploitation of SDS without SFP would require intricate architecture analysis which shall be omitted here.

3 Architecture classification and analysis

Architecture properties are important for basic block timing as well as path timing. We are interested in architecture properties from the analysis perspective. Two techniques to determine the timing of basic block and path segments are used.

1. **Instruction timing addition (ITA)**
   The instruction or statement execution times in a basic block or along a path segment are added. These execution times are taken from a table. This is a very computation time efficient approach. It is somewhat similar to circuit timing analysis. As in circuit timing analysis, minimum and maximum instruction execution times can be considered.
2. **Path segment simulation (PSS)**
   The basic block or path segment is simulated using a cycle true processor model which can be exactly modeled hardware timing and architectures [2].

We distinguish several orthogonal architecture properties relevant this context:

- *data dependent instruction execution times*
  This is typical for some microcoded CISC architectures. Examples are multiplication instructions implemented with $shift - and - add$ or $blockmove$ instructions. It can also occur in processor families where some of the processors have not implemented all instructions but trigger an exception on not implemented instructions to emulate them by software functions with possibly data dependent execution times. Data dependent instruction execution times lead to data dependent basic block execution times. PSS cannot generally guarantee accurate timing results, here, even for segments with SFP property. ITA would be appropriate.
- *pipelined architectures*
  Pipelining makes ITA imprecise. Pipelining is used in RISC processors. If pipelining is deep enough to allow overlapping execution of several basic blocks, which is the case for most of the larger RISC processors, then even basic block simulation would not help. More precisely, ITA must neglect most of the performance gain due to pipelining because it cannot anticipate pipeline hazards. In other words, if a user would design a system based on ITA data, RISC processor performance gain through pipelining would be of no or very little use. PSS precision depends on the length of the simulated path since, to be conservative, worst and best case behavior must be assumed at the beginning and at the end of a path segment. This causes imprecision

which depends on the pipeline length. So, the longer the paths, the more precise is PSS.

- *superscalar architectures*
  Superscalar architectures can execute several basic blocks at a time. They apply dynamic instruction scheduling leading to out-of-order execution and out-of-order completion and even speculative computation [3] . Here, ITA is completely inappropriate. PSS is the correct choice and it's precision, again, depends on the path segment length.
- *program caches*
  Program cache behavior depends on the sequence of instruction fetches. Therefore, PSS is exact for SFP program path segments (supposed the cache is modeled in simulation). ITA alone does not cover caches, unless one would count a potential cache miss for every access to another memory page. There are, however, efficient extensions to ITA to regard at least direct mapped caches [7].
- *data caches*
  Data cache behavior depends on the sequence of data accesses. Therefore, PSS is precise for path segments with SDS. ITA has the same problems as for program caches.

A typical processor can combine several of the properties.

The comparison shows that ITA alone is only suitable for very simple architectures. On the other hand, PSS is a problem for data dependent architecture behavior. The comparison shows that there is not a single technique which is best for all cases.

4 The SYMTA approach

SYMTA (Symbolic hybrid timing analysis) is a hybrid approach combining simulation and formal analysis adaptable to different architecture and program properties.

4.1 Hierarchical flow graph clustering

As a first step, the input program is mapped to a hierarchical control flow graph. In this graph, every control construct, such as $if, case, loop$, corresponds to a hierarchical node and the leaf cells are the basic blocks of the program. Each of the control constructs has an associated condition that decides which of the paths of the construct is executed. Functions are mapped to extra graphs, but can be copied to the calling statement for higher analysis accuracy. This way, an MFP function with an invocation dependent control flow can become an SFP function. Like in [7], we assume structured programs without goto's between hierarchical nodes, because this simplifies flow analysis.

4.2 SFP identification

The second step in the approach is the identification of SFP program parts. More precisely, we want to partition the flow graph nodes into SFP and MFP nodes.

(1) Since MFP requires that at least one control construct depends on input data, we can immediately conclude that every path segment which does not contain an input data dependent control construct must be SFP.

(2) Now we can apply a simple induction over the levels of hierarchy:

- Leaf nodes (basic blocks) are SFP by definition
- (1) $\Rightarrow$ A hierarchical node is SFP, if
  - it only contains SFP nodes
  - and its associated condition is independent of input data.

This defines a simple recursive clustering approach to flow graph partitioning. It automatically cuts the program into SFP and MFP path segments. The FFT in fig. 2 would completely be clustered in one SFP.

SFP clustering is not sufficient when MFP path segments are embedded. Fig. 3(a) shows an example. The example implements a bubble sort algorithm. In this program, the two loops are input data independent such that they always have the same iteration count while the if condition is dependent on the input data a[ ].

We extend the clustering algorithm to merge adjacent SFP blocks:

(3) If the associated condition of a hierarchical node depends on input data, this node is MFP. Set cut points at the beginning and the end of the MFP nodes.

(4) Repeat clustering according to (2) ignoring the MFP nodes but regarding the cut points.

The new clusters found in (4) shall also be defined as SFP, since, except for the path fork and join in the embedded MFP blocks, there is only one path outside these blocks. To be conservative for correct timing analysis, it is, therefore, sufficient to analyze the MFP node separately and assume worst case behavior at the remaining cut points[2]. This is guaranteed by leaving the cut points in (4) inside the SFP clusters. Then no false SFP paths leading to incorrect bounds can be introduced in the next steps. So, the result is still correct, but (4) maximizes SFP path length. Since for most higher performance architectures as well as for architectures with caches, analysis precision increases with path length, this appears to be a good compromise for high precision without path explosion. The results will show the high efficiency.

Fig. 3(b) shows the result for the example in fig. 3(a). Only blocks b5, the condition basic block, and b6, are in MFP path segments. Fig. 3(c) shows how the remaining program path segments are clustered to SFP nodes.

What we need for this clustering approach is an algorithm to determine condition (see above), i.e. input data dependency of conditions. This requires a global data flow analysis [1] with a transitive closure over all data dependencies of variables in control statements. A global data flow analysis, however, does typically not cover dependencies over array elements and operation on data. Therefore, the global data flow analysis is complemented with symbolic simulation of basic blocks [14][3].

In the same step using the same technique, basic block symbolic simulation also determines the SDS property for basic blocks in SFP paths. This is a very useful side effect.

---

[2] Worst case behavior here means the maximally extended upper and lower timing bounds for that architecture

[3] We also tried complete symbolic program analysis but this was only feasible for very small examples and with unacceptable memory resources and computation time. Also, in the examples which we tried it did not provide much additional information useful to SFP determination.

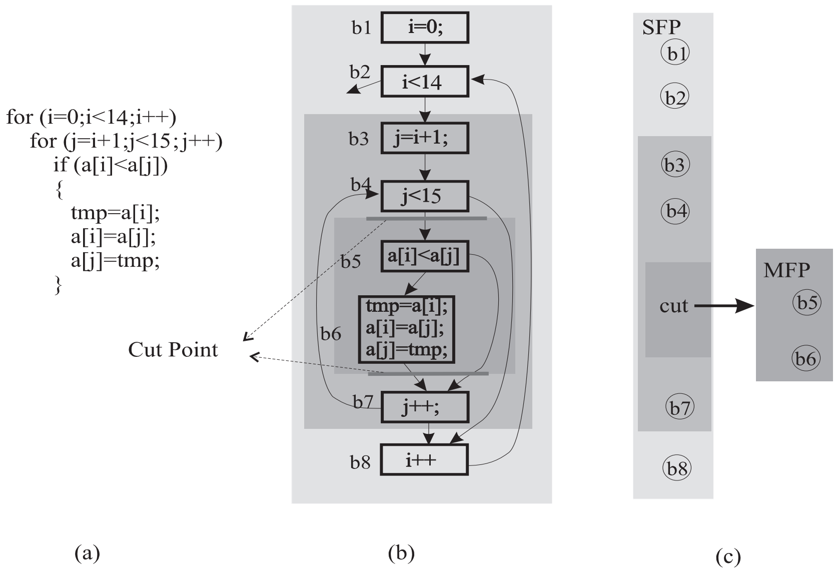

Figure 3: An example program for the SFP identification

4.3 Path segment timing - adaptation to processor architectures

Next problem is the running time determination of the remaining complex SFP nodes. As seen before this is strongly architecture dependent. So far, we have presented PSS and ITA for basic block timing analysis. While PSS can be directly applied to any SFP segment if a cycle true simulator is available, this is not the case for ITA which is only applicable to basic blocks. On the other hand, ITA can be useful for simple architectures without pipelining in particular with data dependent instruction timing.

Again, the solution resorts to the SFP property. The ITA approach obtains the same timing result on whatever path a basic block is executed. So, all we need to know is the number of times it_i a basic block b_i is executed. Then we can multiply this iteration count by the basic block running time $t(b_i)$. Note that it_i is unique because of the SFP property and can therefore be obtained by source code profiling which does not need a simulator but can run on a workstation[4].

Given the hierarchical SFP node p which appears in the reduced flow graph, we can simply use the **sum–of–basic–blocks** model.

$$t(p) = \sum it_i \times t(b_i)$$

So, whenever ITA is accurate it is a good choice, since it is faster than PSS which must use a cycle true simulator. A famous example for such an architecture is the 8051.

In case of data dependent instruction timing, ITA provides a lower and an upper bound for each basic block, $t_{min}(b_i)$ and $t_{max}(b_i)$ and, therefore a lower and an upper bound $t_{min}(p)$ and $t_{max}(p)$ which is then used for MFP analysis.

Pipelined and superscalar architectures can accurately be treated with PSS, as already shown. At the transition point between SFP nodes, worst case behavior must be assumed, i.e. no stalling for the lower bound and maximum time stalling for the upper bound. This overhead time can be added to SFP blocks or can be collected in extra nodes as e.g. proposed in [13].

Program cache miss timing can be treated with PSS. At the transition point between SFP nodes, one can either assume a default cache miss when accessing another memory page or use a more precise cache model and analyze potential misses with higher precision. The difference in precision between PSS and ITA depends on the number of page transitions and real cache misses in the SFP nodes, but generally PSS seems to be the much better choice when applicable (see above).

Similar arguments hold for data caches if one replaces the role of SFP nodes by SFP nodes with SDS property.

A major problem are high performance architectures with data dependent instruction execution times since neither ITA nor PSS are directly applicable with acceptable precision. As an example, many RISC processors, such as the PowerPC (used in embedded systems, e.g., as Motorola RCPU) implement integer divisions with data dependent timing. One approach is to analyze basic blocks with data dependent instructions using ITA hoping that there are few such instructions (which is usually true), another one is to extend the timing bounds by the difference between minimum and maximum instruction execution timing. Both approaches are conservative. One could choose whatever gives the narrower bounds. This extension would be sufficient, but we currently cannot present results for this specific feature since it is not yet implemented in the timing

[4] In reality the problem is a bit more complicated since we have to take care of source code and assembly code basic block correspondence.

analysis tool.

All other combinations of properties lead to subproblems of the architecture in the previous paragraph. So, the approach is complete over processor architecture and program properties, now.

4.4 Global timing analysis using nodes with bounded timing property

For global timing analysis, each hierarchical SFP node is merged to a single SFP node with a single running time. All remaining hierarchical nodes are MFP nodes and all leaf nodes are SFP with accurate running time. When leaf node timing has been determined, we apply the ILP approach presented in [7] which seems to be the most powerful MFP analysis approach known today.

In fig. 3 we let the program simulate on a SPARC simulator. Thereby we obtain the execution time of the SFP cluster $T_{sim}(SFP) = 2113$ cycles. By ILP solution, for one iteration of the MFP cluster (block 5 and 6) we get $T_{ilp}(5,6) = 43$ cycles in the worst case and 22 cycles in the best case. The MFP part is embedded in the SFP cluster and will be still executed 105 times for both worst and best case. Therefore, $T_{ilp}^{worst}(MFP) = 105 \cdot T_{ilp}^{worst}(5,6) = 105 \times 43 = 6825$ cycles in the worst case and $T_{ilp}^{best}(MFP) = 105 \cdot T_{ilp}^{best}(5,6) = 105 \times 22 = 2310$ cycles in the best case. The total number of cycles of the program is given as:

worst case: $T = T_{sim}(SFP) + T_{ilp}^{worst}(MFP) = 2113 + 6825 = 8938$ cycles.

best case: $T = T_{sim}(SFP) + T_{ilp}^{best}(MFP) = 2113 + 2310 = 4423$ cycles.

The execution time for the program in fig. 2 will be simulated entirely without ILP solution because it is completely SFP.

5 A SYMTA timing analysis tool

We have developed a tool using the SYMTA approach. Figure 4 shows the flow graph. First step is symbolic pro-

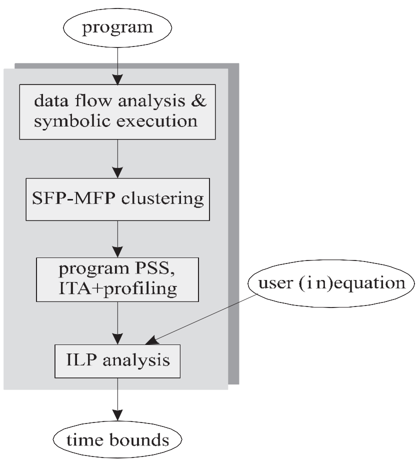

Figure 4: The flow graph of SYMTA timing analysis tool

gram execution and data flow analysis. The analysis results are used for clustering. Next step is either a PSS of the whole program or program profiling with subsequent ITA analysis.

PSS takes user provided input data for simulation which must be complete to cover all paths in the program. If not, ITA must be executed for the remaining basic blocks. The tool is able to identify basic blocks which are not covered and, then, switches to ITA for these blocks. This can only occur for basic blocks embedded in hierarchical MFP nodes.

We have selected two processors for our experiments: a complex 32-bit superscalar SPARC RISC processor with 4-stage pipeline and floating point operations; and a simple 8-bit processor Intel 8051 with no pipelining and no data dependent operations which is widely used in microcontrol systems. PSS is used as primary analysis technique for the SPARC while the 8051 is analyzed using ITA with profiling. For the SPARC processor, we use the GNU C compiler and for the 8051 we use a commercial compiler. Debug information is used to identify the source level basic blocks in the assembly code.

Now, the following timing information is produced by the PSS:

- the execution time of the SFP nodes including the SFP clusters.
- the worst case and best case execution times for data dependent operations and the execution counts of basic blocks of the program. They are later used for the ILP solver.

Final step is the ILP solution on the reduced program flow graph. The system accepts user provided (in)equations to improve accuracy for MFP parts.

In the next section we will give some experimental results using SYMTA.

6 Experimental results

The first table demonstrates the cluster results for a variety of algorithms taken from different sources. We have evaluated the SFP analysis which is shown in table 1. The first column provides the total number of nodes in the program flow graph. The second column contains the nodes which are located in SFP parts. The third column contains the number of nodes in MFP parts. The experimental results have revealed that many parts of the programs have SFP property which can be precisely analyzed using simulation.

Programs	Total nodes	Nodes in SFP		Nodes in MFP		Source lines
3D-image	94	85	90%	9	10%	164
diesel	65	65	100%	0	0%	160
fft	78	78	100%	0	0%	145
bsort	14	8	57%	6	43%	25
smooth	48	39	81%	9	19%	86
blue	80	53	66%	27	34%	127
check-data	18	0	0%	18	100%	44
whetstone	122	122	100%	0	0%	251
line	101	19	19%	82	81%	250
key3	100	100	100%	0	0%	151

Table 1: Experimental results for the Clustering

Programs	Measured bounds(cycles)		Analyzed bounds(cycles)		Analysis time*
	BCET**	WCET**	BCET	WCET	(sec)
SPARC					
3D-image	34908	37848	33874	38037	0.79
diesel	62944	62994	61445	63333	0.84
fft	1498817	1499176	1494650	1499290	135.69
bsort	4423	8938	4423	8938	0.34
smooth	3635651	4846511	3570227	4881135	304.90
blue	3564938	316865761	3345041	346541760	4325.23
check-data	80	431	65	435	0.23
whetstone	2928459	3369459	2880230	3378098	298.19
line	514	1619	381	2035	0.39
8051					
fft	26421460	26421460	26419338	26488288	0.23
bsort	9347	15045	7804	18167	0.12
smooth	9737378	9737516	9737469	9737522	0.23
key3	1218229	1223314	1164883	1265227	0.39
check-data	68	559	63	588	0.17

** The example programs have been analyzed on the SPARC 10 workstation.*
*** WCET: The worst case execution time; BCET: The best case execution time.*

Table 2: Experimental results of the example programs in SYMTA

Table 2 shows the timing results and compare them with the results of an extensive simulation. In order to evaluate the results of our approach, we have to know the real bounds of the programs. Therefore, we selected such programs, where worst case input data can be clearly identified with some effort. The measured bounds in the first column are obtained by simulating the program using these data. The second contains the analyzed bounds of the example programs running on both processors using our approach.

7 Conclusion

We have presented an approach to formal timing analysis and verification with tight timing bounds. The approach combines several analysis and simulation techniques to best exploit different program and architecture properties. Data flow analysis and symbolic execution are used to safely select the analysis technique with the tightest bounds for each program segment. The approach was applied to two very different processor architectures and a variety of programs showing a very high precision. Other potential applications of the approach are software test generation and timing analysis in the high-level synthesis.

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, *Compiler principles, Techniques and Tools,* Bell Telephone Laboratories, Inc. 1987.

[2] T.M. Conte, Ch.E. Gimarc, *Fast simulation of computer architectures,* Kluwer Academic Publishers, 1995.

[3] M. Johnson, *Superscalar Microprocessor Design,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[4] J. Gong, D. D. Gajski, S. Narayan *Software execution from executable specification,* The Journal of Computer & Software Engineering, 2(3), 239–258(1994).

[5] Y-T.S Li, S. Malik, A. Wolfe, *Performance Analysis of Embedded Software with Instruction Cache Modelling,* Proc. of ICCAD 1995. IEEE Society Press, pp. 380-387, 1993.

[6] D.B. Kirk, *Predictable Cache Design for Real-Time Systems.* PhD Thesis Department of Electrical and Computer Engineering Carnegie-Mellon Uni versity, Nov. 1990.

[7] S. Malik, W. Wolf, A. Wolfe, Y. S. Li, T. Yen, *Performance Analysis of Embedded Systems,* NATO ASI, Workshop on Hardware–Software Co-Design, Tremezzo, Italy, 1995.

[8] F. Mueller, R. Arnold, D. Whalley, *Bounding Worst-Case Instruction Cache Performance,* in Real-Time Systems Symposium. 1994 . Krithi, A., Editor. Puerto-Rico: IEEE P ress. p. 172-181.

[9] A. Mok et al. *Evaluating Tight Execution Time Bounds of Programs by Annotations,* Proc. IEEE WS Real-Time Operating Systems and Software, May 89, pp. 74-80.

[10] C. Y. Park, *Predicting Deterministic Execution Times of Real–Time Programs,* PhD Thesis, University of Washington, Seattle 98195, Aug. 1992.

[11] C.Y. Park, A.C. Shaw, *Experiments with a program timing tool based on source-level timing schema.* Proc 11th IEEE Real-Time system Symp., pp. 72-81,1990

[12] P. Puschner Ch. Koza, *Calculating the maximum execution time of real–time programs,* The Journal of Real–Time Systems, 1(2), 160–176, Sept. 1989.

[13] W. Ye, R. Ernst, Th. Benner, J. Henkel, *Fast Timing Analysis for Harware–Software Cosynthesis,* Proc. of ICCD 1993. IEEE Society Press, pp. 452-457, 1993

[14] W. Ye, R. Ernst, *Worst Case Timing Estimation based on Symbolic Execution,* COBRA report, Institute of Computer Engineering, Technical University Braunschweig, Oct. 1995.