A Compiler Optimization to Increase the Efficiency of WCET Analysis

Mohamed Abdel Maksoud
Compiler Research Group
Saarland University
mohamed@cs.uni-saarland.de

Jan Reineke
Real-Time and Embedded Systems Lab
Saarland University
reineke@cs.uni-saarland.de

ABSTRACT

For complex microprocessors, micro-architectural analysis and precise path analysis constitute the most expensive steps in worst-case execution time (WCET) analysis. We introduce a parameterized compiler optimization to reduce analysis time and memory consumption during the two steps.

The optimization makes use of a synchronization instruction, which flushes queues in the memory subsystem. By injecting this instruction at selected program points, analysis uncertainty about the state of the pipeline and the memory subsystem can be drastically reduced, at the cost of an increase in execution time. A parameter allows the user to control the trade-off between increased analysis efficiency and decreased worst-case performance.

We have developed a prototype implementation of the optimization for the PowerPC instruction set architecture, and evaluate it using a version of AbsInt's WCET analyzer `aiT` for the PowerPC 7448, a high-performance microprocessor used in safety-critical real-time systems. On a set of Mälardalen benchmarks, we observe an analysis speedup of around 635% at the cost of an increase in the WCET bound of 6%. Moreover, under a traditional ILP-based path analysis, the WCET bound is *decreased* by 5% while the analysis is sped up by 350%.

Categories and Subject Descriptors

B.8.0 [**Performance and Reliability**]: General

General Terms

Embedded Systems, Program Optimization, Timing Validation

Keywords

WCET analysis efficiency, analyzability, predictability

1. INTRODUCTION

Modern microarchitectures feature numerous mechanisms to increase performance in the common case. Examples include multiple levels of caches, dynamic branch predictors, and pipelines with out-of-order execution facilitated by load-store units with multiple registers buffering pending loads and stores.

For soundness and precision, worst-case execution time (WCET) analysis tools need to determine the possible states of these features throughout the execution of the program being analyzed. However, often, the contents of registers or the cache cannot be precisely determined as they depend on the program's inputs or the particular loop iteration. Whenever the processor's next state depends upon such missing information, the analysis performs a so-called *split*, accounting for *all* possible successor states. For complex microarchitectures state-of-the-art analyses often track billions of possible microarchitectural states, resulting in long analysis times and high memory consumption.

The increasing complexity of microarchitectures and its effect on WCET analysis has been observed earlier [28]. This observation has lead to a body of work on the design of microarchitectures that aim to reconcile performance with predictability [21, 29, 19, 18]. So far, this research has had limited impact on commercially-available microarchitectures.

In this paper, we explore an alternative approach to new hardware solutions: we propose a compiler optimization that reduces the cost of WCET analysis for complex commercial microarchitectures. This is accomplished by inserting a *synchronization* instruction at selected program points. This instruction stalls the execution until all pending instructions execute to completion, effectively flushing queues in the load-store unit and emptying the pipeline.

This reduces the number of analysis states in two ways. The immediate effect is that many analysis states become *similar* after executing the synchronization instruction, and hence can be *merged*. In addition, eliminating uncertainty about pending memory accesses in the load-store unit, reduces the number of splits on subsequent load-store instructions. The reduced number of analysis states comes at the cost of an increase in execution time due to the stalling induced by the synchronization instruction on the one hand, and the increased program size, which may increase the number of cache misses, on the other hand.

To identify valuable locations to insert synchronization instructions, our optimization estimates, for each program point, the loss in terms of execution time and the gain in analysis efficiency. While the former estimate is based on

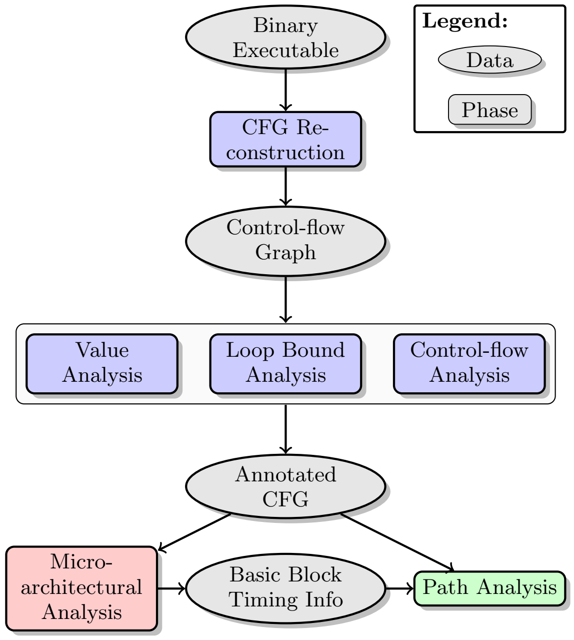

Figure 1: Main components of a timing-analysis framework and their interaction.

the loop-nesting level, the latter is computed using annotations obtained by performing a simple static analysis of the program. These annotations provide rough estimates of how long each instruction takes and how many splits it induces.

We have developed a prototype implementation of the optimization for the PowerPC instruction set architecture. We employ a version of AbsInt's WCET analyzer `aiT` for the PowerPC 7448, a high-performance microprocessor used in safety-critical real-time systems, on a set of Mälardalen benchmarks, to evaluate our prototype. Under an expensive prediction-file based path analysis, we observe an analysis speedup of around 635% at the cost of an increase in the WCET bound of 6%. Moreover, under a traditional ILP-based path analysis, the WCET bound is *decreased* by 5% while the analysis is sped-up by 350%.

The rest of the paper is organized as follows: we provide background on WCET analysis and the PowerPC 7448 architecture in Section 2. In Section 3 we describe the optimization pass, before describing the experimental evaluation in Section 4. After discussing related work in Section 5, we conclude the paper in Section 6.

2. BACKGROUND

2.1 WCET Analysis Flow

Over roughly the last decade, a more or less standard architecture for timing-analysis tools has emerged. Figure 1 gives a general view of this architecture. The following list presents the individual phases and describes their objectives.

1. *Control-flow reconstruction* [24] takes a binary executable to be analyzed, reconstructs the program's control flow and transforms the program into a suitable intermediate representation. Problems encountered in this phase are dynamically computed control-flow successors, e.g. those stemming from switch statements, function pointers, etc.

2. *Value analysis* [3] computes an over-approximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis. The computed information is used for a precise data-cache analysis and in the subsequent control-flow analysis. Value analysis is the only one to use an abstraction of the processor's arithmetic. A subsequent pipeline analysis can therefore work with a simplified pipeline where the arithmetic units are removed. There, one is not interested in what is computed, but only in how long it will take.

3. *Loop bound analysis* [7, 15] identifies loops in the program and tries to determine bounds on the number of loop iterations; information indispensable to bound the execution time. Problems are the analysis of arithmetic on loop counters and loop exit conditions, as well as dependencies in nested loops.

4. *Control-flow analysis* [7, 23] narrows down the set of possible paths through the program by eliminating infeasible paths or by determining correlations between the number of executions of different blocks using the results of value analysis. These constraints will tighten the obtained timing bounds.

5. *Micro-architectural analysis* [6, 26, 10, 4] determines bounds on the execution time of basic blocks by performing an abstract interpretation of the program, combining analyses of the processor's pipeline, caches, and speculation. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc.

6. *Path Analysis* [17, 25] finally determines bounds on the execution times for the whole program by implicit path enumeration using an integer linear program (ILP). Bounds of the execution times of basic blocks are combined to compute longest paths through the program. The control flow is modeled by Kirchhoff's law. Loop bounds and infeasible paths are modeled by additional constraints. The target function weights each basic block with its time bound. A solution of the ILP maximizes the sum of those weights and corresponds to an upper bound on the execution times. In the following, we refer to the kind of path analysis described above as *traditional* ILP-based analysis.

The commercially available tool `aiT` by AbsInt, cf. `http://www.absint.de/wcet.htm`, implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [10, 9, 27, 16].

The ILP-based path analysis in `aiT` comes in two variants depending on how micro-architectural state graphs are constructed [2]:

1. *Traditional ILP-based analysis*, where an ILP is solved to find the worst-case path through the program, given

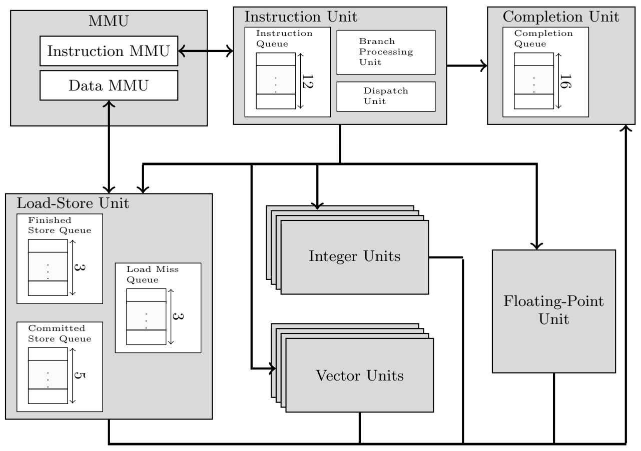

Figure 2: PowerPC 7448 Block Diagram.

worst-case timings of all basic blocks (possibly in various contexts). In this approach the size of the ILP formulation is independent of the size of the micro-architectural state space. The downside is that the computed WCET bound may be imprecise, because the worst-case timings of consecutive basic blocks may not occur simultaneously on a single architectural path through the program.

2. *Prediction-file-based ILP analysis* (PF-ILP), where a global state graph consisting of micro-architectural states is constructed, and an ILP is solved to find the worst-case path through this state graph. This results in a more precise WCET bound since architecturally-infeasible paths are excluded. However, it comes at the cost of a much larger ILP to be solved, whose size depends on the micro-architectural state space.

2.2 Freescale PowerPC 7448

The PowerPC 7448 is a reduced instruction set computer (RISC) superscalar processor that implements the 32-bit portion of the PowerPC architecture and the SIMD instruction set AltiVec architectural extension. It features a two-level memory hierarchy with separate L1 data and instruction caches (Harvard architecture), a unified L2 cache, four independent integer and four independent vector units for superscalar execution. It also features static and dynamic branch prediction, and a sophisticated load-store unit with long buffers.

"The PowerPC 7448 provides virtual memory support for up to 4 PB (2^{52}) of virtual memory and real memory support for up to 64 GB (2^{36}) of physical memory. It can dispatch and complete three instructions simultaneously" [12]. It consists of the following execution units, depicted in Figure 2:

- Instruction Unit (IU): the IU provides centralized control of instruction flow to the execution units. It contains an instruction queue (IQ), a dispatch unit (DU), and a branch processing unit (BPU). The IQ has 12 entries and loads up to 4 instructions from the instruction cache in one cycle. The DU checks register dependencies and the availability of a position in the *completion queue* (described below), and issues or inhibits subsequent instruction dispatching accordingly. The BPU receives branch instructions from the IQ and executes them early in the pipeline. If a branch has a dependency that has not yet been resolved, the branch path is predicted using either architecture-defined static branch prediction or PowerPC 7448-specific dynamic branch prediction.
- Completion Unit (CU): The CU retires an instruction from the 16-entry completion queue (CQ) when all instructions ahead of it have been completed. The CU coordinates with the IU to ensure that the instructions are retired in program order.
- Integer, Vector, and Floating-Point Units: the PowerPC 7448 provides nine execution units to support the execution of integer, fixed point, and AltiVec instructions.
- Cache/Memory Subsystem: The PowerPC 7448 microprocessor contains two separate 32 KB, eight-way set-associative level 1 (L1) instruction and data caches (Harvard architecture). The caches implement a pseudo least-recently-used (PLRU) replacement policy. In addition, the PowerPC 7448 features a unified 1 MB level 2 (L2) cache.
- Load-Store Unit (LSU): The LSU executes all load and store instructions and provides the data transfer interface between registers and the cache/memory subsystem. The LSU also calculates effective address and aligns data. This unit is described in detail in the following section.

Load-Store Unit. The LSU provides all the logic required to calculate effective addresses, handles data alignment to and from the data cache, and provides sequencing for load-store string and load-store multiple operations [12]. The LSU contains a 5-entry load miss queue (LMQ) which maintains the load instructions that missed the L1 cache until they can be serviced. This allows the LSU to process subsequent loads. Unlike loads, stores cannot be executed speculatively: a store instruction is held in the 3-entry finished store queue (FSQ) until the completion unit signals that the store is committed; only then it moves to the 5-entry committed store queue (CSQ). In order to reduce the latency of loads dependent on stores, the LSU implements *data forwarding* from any entry in the CSQ before the data is actually written to the cache. When a load misses the cache, its address is compared to all entries in the CSQ. On a hit, the data is forwarded from the newest matching entry. If the address is also found in the FSQ, however, the LSU stalls since the newest data at this address could be updated should the store instruction in the FSQ be committed.

Analysis Model of the PowerPC 7448. During static analysis, crucial information on program execution such as register and cache contents cannot be determined exactly. When the analysis flow depends on such information, the analysis has to proceed along all possible ways to ensure a sound WCET bound in the presence of timing anomalies [20]. When the analysis is to proceed in more than one path, the analysis state has to be *split*, increasing the size of the state space and hence reducing analysis efficiency.

Splits induced by unknown cache contents and conditional branch outcomes occur independently of the complexity of the pipeline. Other split types, however, are induced by the missing or partial information about the state of the pipeline.

The various queues in the PowerPC 7448 pipeline necessitate keeping track of a significant amount of information, proportional to the number of instructions the processor can execute concurrently. Due to analysis uncertainty about memory addresses, this translates to a significant amount of splits during microarchitectural analysis.

In the load-store unit, the addresses of different memory accesses are represented by enclosing intervals, rather than exact numbers. As described in the previous section, serving a load that misses the cache involves a number of comparisons to the entries in the store queues. Performing these comparisons on imprecise addresses results in splits whenever it cannot be decided whether two addresses alias or not. The number of comparisons is proportional to the queue occupancies at the time instants when loads that missed the cache arrive.

Another source of splits is branch prediction. This is because conditional branches (whose outcome is potentially unknown) are predicted and the instructions are executed speculatively on the predicted path, a split takes place when deciding whether or not the prediction was correct.

Our previous work in [19] suggests that simplifying the load-store unit results in a substantial reduction in the number of splits with little or no change in the WCET bound.

Transforming the program in a way that eliminates or limits the splits induced by such features promises to achieve similar results. This motivates the optimization pass described in the following section.

3. THE OPTIMIZATION PASS

3.1 High-Level Optimization Approach

The mechanism at our disposal are *synchronization instructions*, described in more detail below, which reduce analysis cost at and after the program point at which they are inserted. As inserting such instructions does not come for free—it increases program size and execution times—blindly inserting synchronization instructions everywhere in the program is not a viable option.

Instead, we follow a simple incremental approach that alternates between the following two steps, until a user-defined threshold is reached:

1. A cheap heuristic is used to estimate both the gain in terms of reduced analysis effort and the loss in terms of increased execution time for each program point.

2. A synchronization instruction is inserted at the program point that maximizes the gain/loss ratio.

Step 1 needs to be repeated in each iteration, because each insertion changes the gains and losses of other program points. Figure 3 illustrates this process, whose steps are described in more detail in the following.

3.2 Mechanism: Synchronization Instructions

To eliminate splits due to clashes in the load-store unit, we need to make sure its queues are cleared by the time a new load arrives. The semantics of the data-synchronization instruction (`sync`) is the closest fit for this purpose. Executing `sync` instruction ensures that all preceding instructions

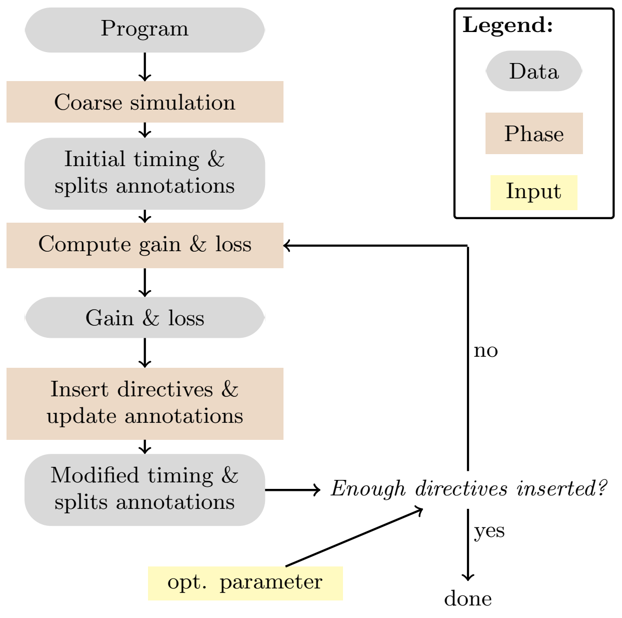
Program
Coarse simulation
Initial timing & splits annotations
Compute gain & loss
Gain & loss
Insert directives & update annotations
Modified timing & splits annotations
Enough directives inserted?
no
yes
done
opt. parameter
Legend:
Data
Phase
Input

Figure 3: The general operation of the optimization pass.

execute to completion before any subsequent instruction is initiated [13]. This implies that inserting a `sync` instruction ensures the emptiness of the LSU queues and consequently excludes the possibility of comparing imprecise addresses. Another benefit is that executing the `sync` instruction makes micro-architectural analysis states *similar*. This fosters merging states and hence reduces the number of subsequent states to be explored. The downside is that inserting `sync`s increases the program size. A longer program will most likely feature a longer execution time and a higher number of splits induced by querying the instruction cache. Moreover, executing the `sync` instruction causes a stalling in the pipeline until all pending operations are completed. This prohibits executing the instructions that follow concurrently and hence prolongs the execution time.

We use the term *normalization point* to refer to a program point before which inserting a `sync` could enhance the analysis efficiency. Normalization points are the points where one or more splits occur. With this choice, every normalization point corresponds to one or more split types. The normalization point is said to be *realized* if a `sync` is actually inserted before it.

3.3 Evaluating Normalization Points

Realizing a normalization point induces gain in terms of the analysis states spared and loss in terms of the increase in the execution time and hence the WCET bound. The additional execution time induced by adding one instruction to a basic block is proportional to how frequently the basic block is executed. This can be approximated as follows:

$$loss = B^n$$

where n is its loop-nesting level and B is a constant signifying the average number of loop iterations. A better approximation of the loss could be computed given information on the loop bounds.

```
int main(int argc, char **argv)
{
    int i, sum = 0;
    for (i=0; i<10; i++)
    {
        sum += 1;
    }
    return 0;
}
```

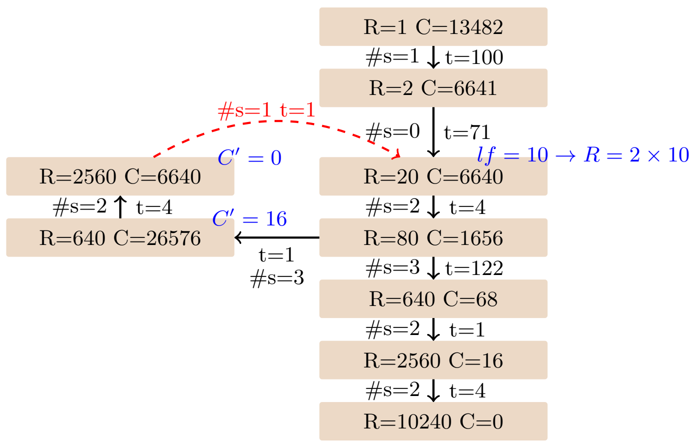

Figure 4: An example program and its control-flow graph annotated with $R(x)$ and $C(x)$ computation results.

We estimate the number of analysis states spared by realizing a normalization point x as follows:

$$gain(x) = (R(x) - 1) \times C(x)$$

where:

$R(x)$	:=	the number of analysis states reaching x,
$C(x)$	:=	the cumulative number of states explored per initial state from x to the end of the program.

To compute $R(x)$ and $C(x)$ for each normalization point, we construct a control-flow graph. Every edge in the graph is annotated by estimates of the time (in cycles) taken to execute the source node (t) and the number of splits encountered during the execution ($\#s$). These estimates are computed by performing a coarse simulation of the program. The simulation keeps track of an approximation of microprocessor state. The state and its evolution is modeled based on the instruction timings listed in the processor manual. To simplify and speed up the simulation, every type of non-determinism in the architecture is avoided by choosing the locally-worst option: memory accesses are assumed to always miss and stores are executed at the lowest speed. Features like branch prediction and speculative execution are not modeled. The only type of non-determinism considered is the one introduced by conditional branches. For such branches, the pass proceeds over all possible paths in depth-first manner. To handle loops, every basic block is processed exactly once per call location.

Furthermore, we make the control-flow graph acyclic to allow for quick estimation of the two metrics. This is accomplished by finding the feedback edges and removing them from the graph. Finally, program points where no splits take place are discarded unless their removal would affect program structure (e.g. return points).

Computing $R(x)$ proceeds from the program entry point (where $R = 1$) in forward breadth-first manner. Every single split doubles the number of reaching state. To account for loops, we multiply the number of states reaching the *loop head* by an arbitrary constant (we assume that every iteration introduces an additional state). A loop head is the program point to which a feedback edge returns. To formalize, $R(x)$ is computed as follows:

$$R(x) = lf(x) \times \sum \left\{ R(p) \times 2^{\#s(p,x)} : p \in predecessors(x) \right\}$$

where the loop factor lf is defined as:

$$lf(x) = B^{|\{e : e \in \text{feedback-edges} \wedge target(e) = x\}|}.$$

Similarly, $C(x)$ is computed from the program end point (where $C = 0$) in a backward breadth-first manner according to the following equation:

$$C(x) = \sum \left\{ 2^{\#s(x,s)} \times \big(t(x,s) + C(s)\big) : s \in successors(x) \right\}.$$

Removing feedback-edges leaves the last program point in a loop with no successors, we call such points *loop tails*. We initially set $C(x)$ at loop tails to zero and proceed with the computation according to the equation above. As a post-processing step, we propagate the value of $C(x)$ along feedback edges, then we update $C(x)$ from the loop tail backwards to the program point after the loop condition.

An Example Gain/Loss Computation. Figure 4 shows an example demonstrating the computation of $R(x)$ and $C(x)$ on a simple program with $B = 10$. The graph contains one feedback edge (dashed), the source point is the loop tail and the target point is the loop head.

$R(x)$ assumes the value 1 at the program entry point. At the third program point (which is the target of the feedback edge), $R(x)$ is computed as 20 rather than 2 since the loop factor of this point is 10 (i.e. $lf(x) = B^1$).

$C(x)$ assumes the value of 0 at the program end point and initially at the loop tail. For demonstration purposes, the initial values of $C(x)$ at the loop points are shown as $C'(x)$ in italic type. After computing $C(x)$ for all program points, its value is propagated along the feedback edge (i.e. $C(x)$ is updated to the value of 6640 at the loop tail), and $C(x)$ is re-computed for the loop nodes. In this example, only the loop tail and its predecessors have their $C(x)$ re-computed.

3.4 Putting It All Together

To let the user control the trade-off between execution time and analysis efficiency, we introduce the *aggressiveness* parameter. This parameter determines the proportion of the normalization points that should be realized.

Given a certain aggressiveness value, the optimization pass operates in the following phases:

Table 1: The Mälardalen benchmarks and their optimization statistics at 40% aggressiveness.

Benchmark	Description	# instructions	Size increase	Time (s)
bsort100	Bubble sort for an 100-integers array.	132	3.03%	0.078
cnt	Counts non-negative numbers in a matrix.	226	3.10%	0.078
crc	Cyclic redundancy check computation on 40 bytes of data.	314	2.23%	0.396
expint	Series expansion for computing an exponential integral function.	187	2.14%	0.184
fac	Calculates the factorial function.	61	4.92%	0.044
fdct	Fast Discrete Cosine Transform.	657	1.98%	1.393
fibcall	Simple iterative Fibonacci calculation, used to calculate fib(30).	54	1.85%	0.022
janne	Nested loop program.	72	4.17%	0.065
ludcmp	Read ten values, output half to LCD.	471	2.55%	0.936
prime	Calculates whether numbers are prime.	146	2.05%	0.086
qurt	Root computation of quadratic equations.	234	1.71%	0.164
ud	Calculation of matrixes.	415	2.17%	0.519

1. A coarse simulation is performed to compute timing and split information for each program point.

2. An annotated control-flow graph is constructed in the way described in Section 3.3.

3. The gain and loss are computed for each unrealized normalization point. The normalization points are then sorted by the ratio of their gain to their loss and the point with the maximum ratio is realized and has its $\#s$ updated accordingly.

   This phase is repeated until the number of covered points is equal to or exceeds

$$\mathit{aggressiveness} \times |\mathit{normalization\_points}| \, .$$

In our implementation of the prototype, we use the GCC compiler [11] (version 4.3.2) as a front-end to compile the C sources to PowerPC assembly. The assembly is then parsed to obtain a control-flow graph. Based on this CFG, the optimization pass described above is implemented to produce an optimized assembly file. The assembly parser, simulator and the optimization pass are implemented in Python [5]. Finally we use the GCC compiler to assemble the binary executable from the optimized assembly source.

4. EXPERIMENTAL EVALUATION

4.1 Experimental Setup

We used the compiler optimization pass on benchmarks from the Mälardalen suite [14] and performed WCET analysis on them using a version of the `aiT` analyzer [2] for the PowerPC 7448. We collected the following metrics to quantify the increase in program size, the gain in the analysis efficiency and the loss in the predicted WCET bound:

- the program size,

- the WCET bound,

- the time taken by the optimization pass, the micro-architectural analysis, and the path analysis combined,

- the maximum of the memory consumptions by the micro-architectural analysis and the path analysis.

We aggregate the metrics obtained for individual benchmarks as follows, where $\mathcal{B}$ is the set of all benchmarks, $\mathcal{M}$ is the set of all metrics, $m_{\text{opt.}}(b)$ is the metric value for the optimized benchmark b, and $m_{\neg\text{opt.}}(b)$ is the respective value for the non-optimized benchmark b:

$$\begin{aligned}
\text{ratio}_m(b) \quad &:= \quad \tfrac{m_{\text{opt.}}(b)}{m_{\neg\text{opt.}}(b)}, \\
\text{aggregate-ratio}_m \quad &:= \quad \underset{\forall b \in \mathcal{B}}{\text{geometric-mean}} \left\{ \text{ratio}_m(b) \right\}, \\
m\text{-increase} \quad &:= \quad \text{aggregate-ratio}_m - 1, \\
m\text{-reduction} \quad &:= \quad 1 - \text{aggregate-ratio}_m .
\end{aligned}$$

We also refer to the analysis-time-gain metric as the analysis speedup.

For the first set of results, we configured the `aiT` analyzer to use the more precise, yet more expensive prediction-file based ILP path analysis (with the CLP solver [1]). For the second set of results, we performed the same analyses using the computationally-cheaper traditional ILP path analysis. To account for the relatively small benchmarks, the instruction cache size was reduced to 1 KB.

The experiment was performed on a virtual machine with 14 64-bit cores, 54 GB RAM with QEMU/KVM on an AMD Opteron 8360 SE with 16 64-bit cores and 64 GB RAM. As the WCET analysis is not parallelized, we ran multiple analyses concurrently on this machine. To prevent paging, the concurrency level was adjusted such that the combined memory consumption by the running analyses never exceeded the physical memory size.

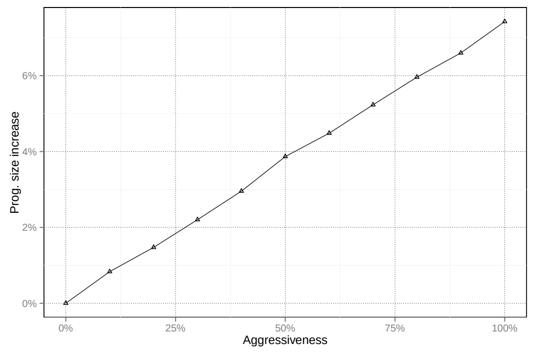

Figure 5: Aggregate program size increase for different values of aggressiveness.

Table 2: WCET bounds and performance metrics for benchmarks using PF-ILP path analysis (aggressiveness=40%).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Opt.	Opt.	Ratio	¬Opt.	Opt.	Speed-Up	¬Opt.	Opt.	Ratio
bsort100	567899	592295	1.0430	13404.45	69.36	193.2367	8186	701	0.0856
cnt	21897	25927	1.1840	88487.60	24.41	3623.1884	21594	95	0.0044
crc	353724	357144	1.0097	24.12	18.54	1.3012	207	179	0.8647
expint	12716	12824	1.0085	2.41	2.67	0.9043	59	66	1.1186
fac	3404	4021	1.1813	0.93	0.46	2.0022	58	0	0.0000
fdct	33637	35158	1.0452	146.50	17.11	8.5626	849	75	0.0883
fibcall	3387	3356	0.9908	0.48	0.39	1.2220	0	0	1.0000
janne	1793	2083	1.1617	0.86	1.1306	0.8845	58	57	0.9828
ludcmp	17750	18986	1.0696	16.99	2.88	5.9061	101	67	0.6634
prime	6801	6911	1.0162	6.25	1.92	3.2551	75	58	0.7733
qurt	18900	18496	0.9786	271.20	69.18	3.9202	822	294	0.3577
ud	14867	15835	1.0651	11.73	3.94	2.9787	83	67	0.8072
geometric mean			**1.0605**			**6.3600**			**0.1440**
geometric mean excl. `cnt`			**1.0500**			**3.5723**			**0.1978**

4.2 Experimental Results

Optimization Results. Table 1 describes the benchmarks used in the experiment along with the percentage increase in program size and time taken by the optimization pass with the aggressiveness parameter set to 40%. The time taken by the optimization is negligible in comparison to the time taken by the value analysis or the micro-architectural analysis as we shall see in the following section. The aggregate increase in program size for different values of aggressiveness is plotted in Figure 5. As expected, the aggregate increase is proportional to the aggressiveness value.

WCET Analysis Results. For each benchmark, the value of the metric is shown for the non-optimized and the optimized versions, along with the ratio between the two values.

First, we consider the metrics collected using the more precise path analysis PF-ILP. The metrics obtained at 40% aggressiveness are presented in Table 2. The WCET bound is slightly higher in most of the optimized versions, with an aggregate increase of 6.05%. An increase in the execution time is expected due to the additional `sync` statements that need to be fetched and the stalling in the pipeline it causes. On the other hand, the analysis has a substantial aggregate speedup of approximately 636% and its memory consumption was reduced by about 85%.

Surprisingly, some benchmarks, i.e., `fibcall` and `qurt` show a *decrease* in the WCET bound. This decrease could be attributed to the change in the memory access pattern. Alternating the execution of code and data accesses induces less overhead than executing the accesses of each type in chunks. The benchmark `fac` and `janne` were harmed most by the optimization, with a significant increase in the WCET bound without achieving a proportional speedup. This implies that the gain/loss ratio of one or more normalization points was over-estimated.

The benchmark `cnt` suffers a large increase in the WCET bound too, yet it displays the highest analysis speedup and memory-consumption reduction. Examining the detailed analysis statistics, we found that analyzing the non-optimized version of this benchmark encountered over 16 million splits due to clashes in the load-store unit. The optimization pass with 40% aggressiveness reduced this number to a 40 thousand. This explains the huge gains achieved for

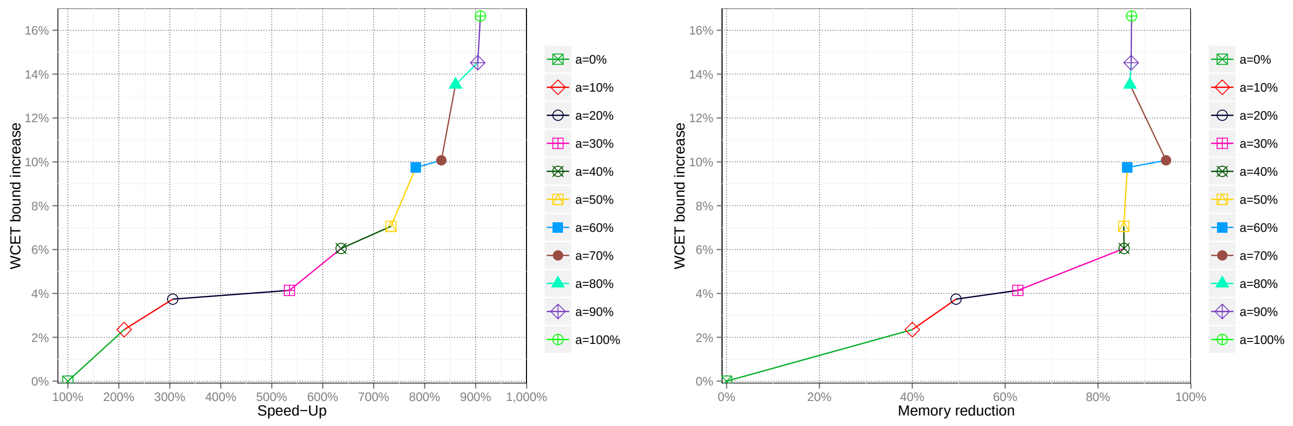

Figure 6: WCET increase vs. speedup for several aggressiveness values using PF-ILP path analysis.

Table 3: WCET bounds and performance metrics for benchmarks using ILP path analysis (aggressiveness=40%).

	WCET *in cycles*			**Analysis time** *in seconds*			**Memory consumption** *in MBytes*		
Benchmark	¬Opt.	Opt.	Ratio	¬Opt.	Opt.	Speed-Up	¬Opt.	Opt.	Ratio
bsort100	683942	767803	1.1226	832.81	40.87	20.3782	6873	702	0.1021
cnt	40422	28654	0.7089	9596.36	17.70	542.2993	3340	90	0.0269
crc	361219	358287	0.9919	15.65	10.34	1.5135	211	178	0.8436
expint	13543	13343	0.9852	1.99	1.51	1.3164	58	58	1.0000
fac	4357	4277	0.9816	0.72	0.42	1.7102	58	0	0.0000
fdct	36459	37420	1.0264	21.86	11.50	1.9010	98	74	0.7551
fibcall	3793	3648	0.9618	0.42	0.35	1.1780	0	0	1.0000
janne	2313	2339	1.0112	0.60	0.57	1.0615	57	57	1.0000
ludcmp	21945	20798	0.9477	6.16	1.84	3.3476	98	66	0.6735
prime	8354	7435	0.8900	3.58	1.30	2.7629	74	58	0.7838
qurt	27935	23604	0.8450	45.72	14.02	3.2614	266	130	0.4887
ud	18660	18162	0.9733	4.97	2.92	1.6986	82	58	0.7073
geometric mean			**0.9483**			**3.5902**			**0.2048**
geometric mean excl. `cnt`			**0.9737**			**2.2752**			**0.2462**

this benchmark. Excluding **`cnt`** from the aggregate values reduces the analysis speedup and memory-consumption reduction while it improves the increase in the WCET bound.

The second-best speedup and memory-consumption reduction is seen in the benchmark `bsort100`. We can see a pattern here: the benchmarks involving significant number of iterations over large arrays benefit most from the optimization.

To examine the effect of the aggressiveness value, we computed the aggregate WCET increase versus the analysis speedup and memory-consumption reduction for several values of the parameter, the results are shown in Figure 6. The WCET bound increases proportionally with the aggressiveness peaking around 17%. The speedup also increases proportionally with the aggressiveness up to the value of 90%, peaking close to 900%. The memory-consumption reduction increases proportionally with the aggressiveness up to 40%, spikes at 70% aggressiveness to around 95% and then declines to around 88% for greater aggressiveness values. A possible explanation of this observation is that there are two factors which affect the memory-consumption reduction: the number of analysis states spared by inserting **`sync`** instructions and the number of splits induced by the additional queries to the the instruction cache.

Next, we consider the metrics collected using the traditional ILP-based path analysis. The results are shown in Table 3 and Figure 7. While the speedup and memory-consumption reduction are not as significant as they are in the PF-ILP case (approximately 450% and 90% at maximum, respectively), the aggregate WCET increase is consistently *negative* for all aggressiveness values below 70%. The reduced speedup is attributed to the fact that the ILP path analysis does not depend on the complexity of the microarchitectural analysis. The path analysis therefore does not benefit from the reduced size of the state-space in terms of performance. It does benefit though in terms of precision. Realizing a normalization point in a basic block forces the processor to execute all pending operations within the same basic block. Localizing such operations within basic blocks reduces the infeasible combinations of events that the traditional ILP-based path analysis considers to compute the global bound. The noticable reduction in the WCET bound can therefore be explained by this precision enhancement.

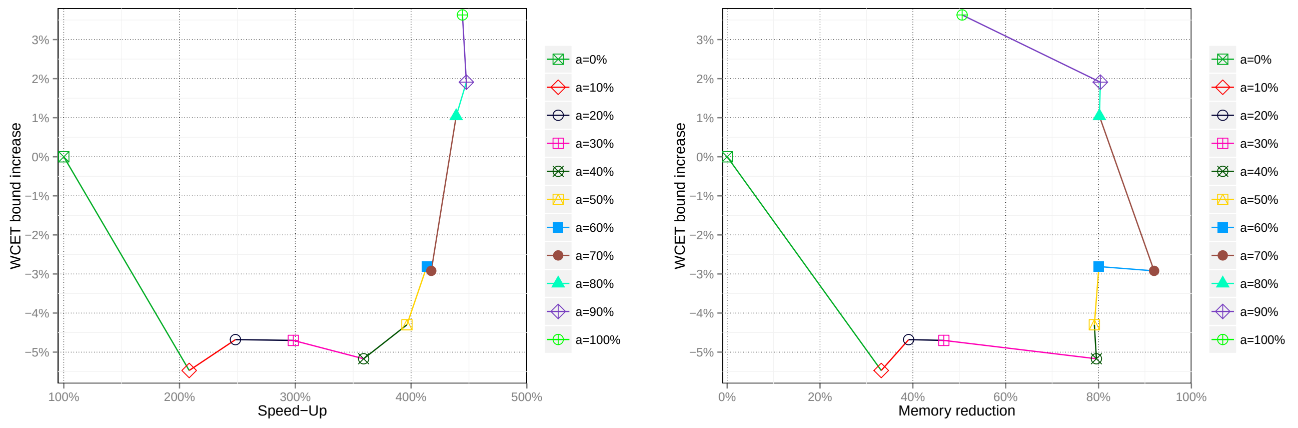

Figure 7: WCET increase vs. speedup for several aggressiveness values using ILP path analysis.

5. RELATED WORK

Recently, significant efforts have been undertaken to develop timing-predictable microarchitectures. The goal of such efforts is to develop microarchitectures that have good worst-case performance and permit sound, precise, and efficient timing analysis.

Wilhelm et al. [29] recommend using compositional pipelines and separate level-1 caches for code and data with the least-recently-used replacement policy. The recommendation of using the less-sophisticated compositional pipelines is motivated by the fact that the execution time is often dominated by memory-access times. The optimization we present here is inspired by this observation: we alter the program at specific points to render the stall-on-accident behavior characteristic to compositional architectures. The Java-Optimized Processor [22] by Schoeberl was designed to be WCET friendly. Beside featuring constant instruction execution times, the processor design prevents interleaving instruction fetches with data accesses by loading whole methods on invocation and return into the instruction cache. Rochange et al. [21] propose an execution mode of a superscalar microprocessor which excludes interferences between consecutive basic blocks. While this method achieves significant reduction in the analysis complexity, it causes a large slow-down of the system. This approach would roughly correspond to inserting synchronization instructions at the beginning of each basic block. Liu et al. [18] present the PTARM, a PRET (precision-timed) architecture implementing a subset of the ARM instruction set architecture. The architecture features a thread-interleaved pipeline which exploits thread-level parallelism to combine high throughput with the compositional way instructions are executed within each thread.

Our previous work [19] suggests that shortening queues in the load-store-unit of the PowerPC 7448 causes little or no increase in execution times while speeding up WCET analysis significantly. We observe a similar speedup in WCET analysis in the present compiler-based approach as was observed for the hardware modifications suggested in [19]. However, in contrast to the present paper, the hardware approach did not incur an increase in the WCET bound. The main reason for the difference is that synchronization instructions stall the pipeline.

The compiler optimizations in the WCET domain we are aware of aim at improving the WCET bound rather than WCET analysis efficiency. Falk et al. [8] propose a variety of techniques in this direction. Some of the mechanisms used to achieve the reduction are reducing the number of calling contexts for each procedure (and hence improving the precision of the value analysis), implementing a better loop-bound analysis, and reducing the frequency of jumps (and consequently their performance penalty). The first two mechanisms likely reduce analysis time as well. However, these particular optimizations are targeting software rather than hardware aspects of WCET analysis, and are thus orthogonal to the optimization we present in this work.

6. CONCLUSIONS

We have presented a parameterized compiler optimization pass to increase WCET analysis efficiency. Experimental results confirm that the pass achieves significant analysis speedup at the cost of a small increase in the WCET bound. The optimization also enables the use of traditional ILP-based path analysis with greater precision. Having a parameter to control the aggressiveness of the optimization enables the user to control the trade-off between system performance and analyzability. In contrast to approaches that rely on custom predictable hardware, our compiler-based approach is readily applicable to existing commercial microarchitectures.

In future work, the optimization pass could be improved by using the results of a loop-bound analysis to better estimate the increase in execution time due to synchronization instructions.

7. ACKNOWLEDGMENTS

We would like to thank Reinhard Wilhelm and Markus Pister for insightful input and constructive discussions, and AbsInt for supporting our experimental evaluation.

8. REFERENCES

[1] COIN-OR Linear Programming. `http://www.coin-or.org/Clp`

[2] AbsInt Angewandte Informatik GmbH. *AbsInt Advanced Analyzer for PowerPC MPC7448 (Simple Memory Model): User Documentation.* `http://www.absint.com/ait/mpc7448.htm`

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM Press. `http://dx.doi.org/10.1145/512950.512973`

[4] C. Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, 2013. `http://dx.doi.org/10.1145/2435227.2435236`

[5] F. L. Drake and G. Rossum. *The Python Language Reference Manual.* Network theory Ltd., 2011. `http://www.worldcat.org/isbn/9781906966140`

[6] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Uppsala University, Sweden, 2002. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5355`

[7] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par*, pages 1298–1307, 1997. `http://dx.doi.org/10.1007/BFb0002886`

[8] H. Falk and P. Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Journal on Real-Time Systems*, 46(2):251–300, 2010. `http://dx.doi.org/10.1007/s11241-010-9101-x`

[9] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *International Conference on Embedded Software*, volume 2211 of *LNCS*, pages 469–485, 2001. `http://dx.doi.org/10.1007/3-540-45449-7_32`

[10] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Sys.*, 17(2-3):131–181, 1999

[11] Free Software Foundation. *GNU GCC Manual*, 2005. http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/

[12] Freescale Semiconductor. *MPC7450 RISC Microprocessor Family Reference Manual*

[13] Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPCTM Architecture*, 2005. http://www.freescale.com/files/product/doc/MPCFPE32B.pdf

[14] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG. http://dx.doi.org/10.4230/OASIcs.WCET.2010.136

[15] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Sys.*, pages 129–156, 2000. http://dx.doi.org/10.1023/A:1008189014032

[16] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003. http://dx.doi.org/10.1109/JPROC.2003.814618

[17] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, 1995. http://dx.doi.org/10.1145/217474.217570

[18] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012)*, October 2012. http://dx.doi.org/10.1109/ICCD.2012.6378622

[19] M. A. Maksoud and J. Reineke. An empirical evaluation of the influence of the load-store unit on WCET analysis. In T. Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 13–24, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. http://dx.doi.org/10.4230/OASIcs.WCET.2012.13

[20] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006. http://dx.doi.org/10.1.1.165.9737

[21] C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 307–314, New York, NY, USA, 2005. ACM. http://dx.doi.org/10.1145/1062261.1062312

[22] M. Schoeberl. JOP: A Java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889, pages 346–359. Springer, 2003. http://dx.doi.org/10.1007/b94345

[23] I. Stein and F. Martin. Analysis of path exclusion at the machine code level. In C. Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. http://dx.doi.org/10.4230/OASIcs.WCET.2007.1196

[24] H. Theiling. *Control-Flow Graphs For Real-Time Systems Analysis*. PhD thesis, Saarland University, Saarbrücken, Germany, 2002. http://scidok.sulb.uni-saarland.de/volltexte/2004/297/

[25] H. Theiling. ILP-based interprocedural path analysis. In *International Conference on Embedded Software*, volume 2491 of *LNCS*, pages 349–363. Springer, 2002. http://dx.doi.org/10.1007/3-540-45828-X_26

[26] S. Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004. http://scidok.sulb.uni-saarland.de/volltexte/2005/466/

[27] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 625–632, June 2003. http://dx.doi.org/10.1109/DSN.2003.1209972

[28] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157–177, 2004. http://dx.doi.org/10.1023/B:TIME.0000045316.66276.6e

[29] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009. http://dx.doi.org/10.1109/TCAD.2009.2013287