Worst-Case Execution Time Analysis for Parallel Run-Time Monitoring

Daniel Lo and G. Edward Suh
Cornell University
Ithaca, New York, USA
{dl575,gs272}@cornell.edu

ABSTRACT

The increasing safety-critical role of real-time systems requires increased attention to their security and reliability. Several recent studies have shown that parallel run-time monitoring of programs can significantly improve the security and reliability of computing systems. However, these techniques cannot be applied to real-time systems without first estimating their impact on worst-case execution time (WCET). In this paper, we present a method for determining the impact of parallel monitoring on WCET using a mixed integer linear programming (MILP) formulation. We use our method to estimate the WCET for seven benchmark programs and two possible monitoring techniques. This estimate is compared against observed execution times from simulation and an upper bound based on sequential monitoring. The results show that our method estimates a WCET within 71% of worst-case observed execution times and up to 74% lower than the sequential bound.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Measurement, Performance, Reliability, Security

Keywords

WCET analysis, run-time monitoring, real-time systems

1. INTRODUCTION

Embedded real-time systems are becoming increasingly prevalent as we deeply integrate computing devices into the physical world. For example, many mechanical systems including automobiles and planes are now electronically controlled by computers. Because electronic systems can provide more intelligent control and coordination through networks, such cyber-physical integration is expanding into even more systems including buildings, medical systems, and power grids. Secure and reliable computation is critical in these systems because a malfunction may cause physical damage or loss of life.

In this context, recent studies have shown that parallel monitoring of run-time program behavior can significantly improve the security and reliability of a computing system with minimal overheads. As an example, Dynamic Information Flow Tracking (DIFT) is a recently proposed security technique that tracks and restricts the use of untrusted I/O inputs, and has been shown to be able to effectively detect a large class of common software attacks [17]. While software DIFT on a single core can incur a significant slowdown even with optimizations (3.6x on average) [14], parallel DIFT on multiple processing modules can reduce overheads to tens of percents on average [4]. Similarly, run-time monitoring can enable many new capabilities such as fine-grained memory protection [19], array bound checks [5], hardware error detection [11], etc.

However, today's parallel monitoring techniques cannot be easily applied to critical real-time systems due to their lack of timing guarantees. The development of a safety-critical real-time system requires an estimate of the worst-case execution time (WCET) of each task in order to ensure that tasks meet the system's real-time deadlines. Yet, previous studies on parallel monitoring have only focused on average slowdowns through simulations with no worst-case guarantee. Unfortunately, estimation of the worst-case performance overhead of parallel monitoring is not straightforward because of its loosely coupled nature. In the best case, the monitoring happens in parallel to the main task and does not cause any slowdown. However, parts of the main task with heavy monitoring may be required to slow down in order to allow the parallel monitor to keep up.

In this paper, we present a method for estimating the increase in WCET of programs running on a system with parallel monitoring. We first investigate how to mathematically model the loosely-coupled relationship between the main processing core and parallel monitoring hardware, which are often connected through a FIFO buffer with a fixed number of entries. The resulting model is non-linear but can be transformed into a mixed integer linear programming (MILP) formulation. The MILP formulation produces the maximum number of cycles for each basic block that the main core may be stalled due to monitoring. These monitoring stalls can be incorporated into popular WCET analysis methods based on implicit path enumeration techniques (IPET) [9] which use integer linear programming (ILP).

We evaluate the effectiveness of the proposed method by

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

comparing its WCET estimates with a conservative estimate for sequential monitoring, simulation results, and the WCET without monitoring. The experiments use the Mälarden WCET benchmark suite [7] and two monitoring techniques: uninitialized memory check (UMC) and control flow protection (CFP). The results indicate that our WCET formulation can provide a bound that is up to 74% lower than a straightforward estimate from a sequential formulation. These bounds are within 71% of observed worst-case run times from simulations for the selected benchmarks. This is similar to the results when no monitoring is present which show up to a 52% difference between simulations and WCET estimates. As a result, the proposed WCET estimation method enables parallel monitoring techniques to be applied to hard real-time systems with worst-case timing guarantees without being excessively conservative.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the parallel monitoring architecture that is modeled in this paper. Section 4 develops the MILP formulation for WCET analysis and Section 5 presents experimental evaluation results. Finally, Section 6 concludes the paper.

2. RELATED WORK

This paper aims to enable parallel monitoring techniques on real-time systems by providing a general WCET analysis framework that can be applied to a broad range of monitoring techniques. While there exist many parallel monitoring schemes where our WCET analysis can be applied to, we briefly discuss some recent parallel platforms here as examples. For example, INDRA [15] uses a checker core to monitor coarse-grained events on a computation core such as function call/return, code origin inspection, and control flow inspection. Nagarajan et al. studied implementing DIFT on multi-cores [12]. Chen et al. proposed hardware acceleration techniques for multi-core systems and showed that a set of parallel monitoring techniques for security and software debugging can be realized with low performance overheads (tens of percents) [4]. FlexCore [6] shows that parallel monitoring can be made even more efficient by utilizing heterogeneous accelerators implemented on FPGA fabric. These previous studies demonstrate that parallel monitoring can significantly improve system security and reliability with minimal overheads.

Estimating the worst-case execution time of a sequential program on a single-core system is a well studied problem. A survey paper by Wilhelm et al. [18] provides an overview of existing methods and tools in this context. However, to the best of our knowledge, this paper represents the first study on the WCET of parallel monitoring. Researchers have recently started studying the WCET problem for multi-core systems. For example, Paolieri et al. proposed a multi-core hardware architecture for hard real-time systems and analyzed its WCET behavior [13]. McAiT is a tool that has been developed for WCET analysis of multi-core real-time software [10]. These studies focused on the contention between parallel programs for shared resources such as memory. However, the loosely coupled link between the main core and parallel monitoring hardware represents a producer-consumer relationship rather than shared resources. Thus, we found that previously developed techniques were not directly applicable or easily adaptable to provide a tight WCET bound for a system with parallel monitoring.

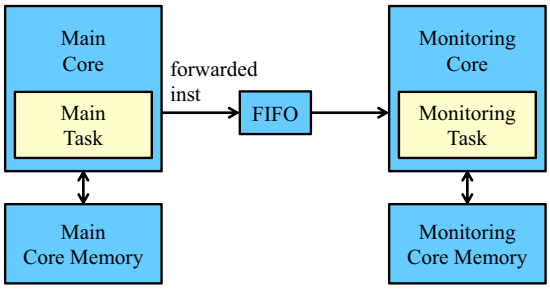

Figure 1: Parallel monitoring architecture model.

3. ARCHITECTURE MODEL

Figure 1 shows the model of run-time parallel monitoring architecture that is assumed in this paper. The architecture consists of two parallel processing elements, main and monitoring cores, which are loosely coupled with a FIFO buffer. The main core runs a computation task, called the *main task*, which performs the original function of the real-time system. The monitoring core receives a trace of certain main task instructions through a FIFO, and performs a *monitoring task* in parallel. We refer to the main task instructions that need to be sent to the monitoring core as *forwarded instructions* or *monitored instructions*. The forwarded instructions are determined based on a particular monitoring technique, and are often sent by the main core transparently without explicit instructions added to the main task. The FIFO allows the monitoring core to operate in a decoupled manner by buffering forwarded instructions. However, if the FIFO is full, the main core needs to wait on a forwarded instruction until a FIFO entry becomes available. We refer to these stalls of the main core due to monitoring as *monitoring stalls* and the number of cycles stalled as *monitoring stall cycles*. The forwarded instruction triggers the monitoring core to execute a series of monitoring instructions.

There are many possible monitoring techniques that can be implemented on this monitoring architecture. One example is to use the monitoring core to detect a software bug that reads a memory location before writing a valid value. We call this monitor an uninitialized memory check (UMC). In UMC, the main core forwards load and store instructions to the monitoring core once they happen in the main task. On a store, the monitoring task sets a tag bit corresponding to the store's memory location, indicating that the location has been initialized. On a load, the monitoring task checks the tag bit, and raises an exception if the bit is not set.

The analysis in this paper focuses on the interaction between the main core and the monitoring core through a FIFO with an assumption that each core has its own memory, as shown in Figure 1. Therefore, there is no interference between the two cores on memory accesses. This configuration applies for typical multi-core embedded microprocessors or a small monitor with a dedicated memory that is attached to a large core. The main core is assumed to not exhibit timing anomalies. This is required so that the worst-case monitoring stall cycles can be assumed to produce the WCET on the main core. This paper does not make any other assumptions on the microarchitecture of each processing core. However, we assume that the WCET of a main task and a monitoring task on the given processing cores can be estimated individually using traditional WCET techniques.

4. WCET ANALYSIS

This section presents our method for estimating the im-

pact of parallel run-time monitoring on WCET. We first review the traditional ILP-based analysis for a sequential program execution, and show that this analysis can be extended to incorporate the overhead of monitoring if the worst-case increase in execution time can be estimated for each basic block. Then, we discuss how to estimate the worst-case monitoring stalls, which happen when the FIFO is full and cannot take a forwarded instruction. We start this discussion with a simple yet rather conservative bound based on the case when a monitoring task always stalls the main task (Section 4.2). Then, we show how the FIFO decoupling can be modeled analytically (Section 4.3), and formulated using MILP (Section 4.4) to create a tighter WCET bound.

4.1 Implicit Path Enumeration

Most of the WCET analysis techniques today rely on an ILP formulation that is obtained from implicit path enumeration techniques [9]. In this method, a program is converted to a control flow graph (CFG). From the control flow graph, an ILP problem is formulated that seeks to maximize

$$t = \sum_{B \in \mathcal{B}_{CFG}} N_B \cdot c_{B,max}$$

where $\mathcal{B}_{CFG}$ is the set of basic blocks in the control flow graph. N_B is the number of times block B is executed and $c_{B,max}$ is the maximum number of cycles to execute block B. The maximum value of t is the WCET of the task. To account for the fact that only certain paths in the graph will be executed, a set of constraints are placed on N_B. For example, on a branch, only one of the branches will be taken on each execution of the block. A variable can be assigned to each edge corresponding to the number of times that edge is taken. The number of times edges out of the block are taken must equal the number of times the block is executed. Similarly, the number of times edges into the block are taken must equal the number of times the block is executed. Various methods have been developed to create additional constraints to convey other program behavior [9, 18].

Integer linear programming is an attractive optimization technique for this problem because the solution found is a global optimum. In addition, many aspects of program and architecture behavior can be described by adding constraints to the ILP problem. Several open source and commercial ILP solvers exist which can solve the formulated ILP problem. Thus, in developing a method for estimating the WCET of parallel run-time monitoring, we look to build upon this ILP framework.

The IPET-based ILP formulation can be extended in a straightforward fashion to incorporate run-time monitoring overheads if we have the maximum (worst-case) monitoring stall cycles for each basic block by maximizing

$$t = \sum_{B \in \mathcal{B}_{CFG}} N_B \cdot (c_{B,max} + s_{B,max})$$

Here, $s_{B,max}$ represents the maximum number of cycles that block B is stalled due to monitoring. In this sense, the challenge in WCET analysis with monitoring lies in determining $s_{B,max}$. The rest of this section addresses this problem.

4.2 Sequential Monitoring Bound

One way to determine a conservative bound on the worst-case monitoring stall cycles is to consider sequential monitoring. In sequential monitoring, the monitoring task is run in-line with the main task on the same core rather than in parallel. That is, after each instruction that would be forwarded, the monitoring task is run on the main core before the main task resumes execution. In this case, the WCET estimate can be obtained from a traditional method by analyzing one program that contains both main and monitoring tasks. The resulting WCET can be considered as a simple bound for parallel monitoring because it models the case where every forwarded instruction causes the main core to stall. However, this bound is extremely conservative as it does not account for the FIFO buffering or the parallel execution of the monitoring core. These features are critical to utilizing run-time monitoring techniques while maintaining low performance overheads.

4.3 FIFO Model

To obtain tighter WCET bounds, we need to model the FIFO. The main task can continue its execution as long as a FIFO entry is available, but needs to stall on a forwarded instruction if the FIFO is full. The WCET model needs to capture the worst-case (maximum) number of entries in the FIFO at each forwarded instruction and determine how many cycles the main task may be stalled due to the FIFO being full. Here, we propose a mathematical model to express the load in the FIFO and estimate the worst-case stalls.

In this approach, the original control flow graph must be transformed so that each node contains at most one forwarded instruction which is located at the end of the code sequence represented by the node. This transformed graph is called a *monitoring flow graph (MFG)*. Intuitively, the analysis needs to consider one forwarded instruction at a time in order to model the FIFO state on each forwarded instruction and capture all potential stalls from monitoring.

To model how full the FIFO is, we define the concept of *monitoring load*. The monitoring load is the number of cycles required for the monitoring core to process all outstanding entries in the FIFO at a given point in time. The monitoring load increases when a new instruction is forwarded by the main task, and decreases as the monitoring core processes forwarded instructions. For simplicity, the increase in monitoring load for any forwarded instruction is conservatively assumed to be the worst-case (maximum) monitoring task execution time among all possible forwarded instructions. This maximum, $t_{M,max}$, can be obtained from the WCET analysis of the monitoring tasks. We make this simplification because it is difficult to model the FIFO mathematically at an entry-by-entry level. With this simplification, each FIFO entry is identical and so the monitoring load fully represents the state of the FIFO. The monitoring load cannot be negative and is upper-bounded by the maximum monitoring load the FIFO can handle, l_{max}. The maximum monitoring load is the number of FIFO entries, n_F, multiplied by the increase in monitoring load for one forwarded instruction, $t_{M,max}$.

In our context, we need to determine the worst-case (maximum) monitoring load at the node boundaries in the MFG. For a given node, M, in the MFG, we define li_M as the monitoring load coming into the node and lo_M as the monitoring load exiting the node. The change in monitoring load for the node is denoted by Δl_M. The maximum Δl_M can be calculated as the difference between the WCET of a monitoring task that corresponds to M and the minimum

execution cycles of the node, $c_{M,min}$:

$$\Delta l_M = \begin{cases} t_{M,max} - c_{M,min}, & \text{forwarded inst. } \in M \\ -c_{M,min}, & \text{no forwarded inst. } \in M \end{cases}$$

In order to ensure that the analysis is conservative in estimating the worst-case (maximum) stalls, we use the best-case (minimum) execution time for the main task here.

Because the monitoring load is bounded by zero and the maximum load that the FIFO can handle, l_{max}, the monitoring load coming out of a node is

$$lo_M = \begin{cases} 0, & li_M + \Delta l_M < 0 \\ li_M + \Delta l_M, & 0 \le li_M + \Delta l_M \le l_{max} \\ l_{max}, & li_M + \Delta l_M > l_{max} \end{cases}$$
$$l_{max} = n_F \cdot t_{M,max}$$

The worst-case monitoring load entering node M, li_M, is the largest of the output monitoring loads among nodes with edges pointing to node M. Let $\mathcal{M}_{\text{prev}}$ represent the set of nodes with edges pointing to node M. Then,

$$li_M = \max_{M_{prev} \in \mathcal{M}_{prev}} lo_{M_{prev}}$$

The above equations describe the worst-case monitoring load at each node boundary. A monitoring stall occurs when a forwarded instruction is executed but there is no empty entry in the FIFO buffer. In terms of monitoring load, if a node would add monitoring load that would cause the resulting total load to exceed l_{max}, then a monitoring stall occurs. The number of cycles stalled, s_M, is the number of cycles that this total exceeds l_{max}. That is,

$$s_M = \begin{cases} 0, & li_M + \Delta l_M < l_{max} \\ (li_M + \Delta l_M) - l_{max}, & li_M + \Delta l_M \ge l_{max} \end{cases}$$

Then, the worst-case monitoring stall cycles for each MFG node can be obtained by maximizing the sum of the s_M across all possible execution paths:

$$\max \sum_{M \in \mathcal{M}_{MFG}} s_M$$

where $\mathcal{M}_{MFG}$ is the set of nodes in the MFG. Once the worst-case stalls for each MFG node is found, the worst-case stalls for a CFG node, $s_{B,max}$, can be computed by simply summing the stalls from the corresponding MFG nodes. We note that since the monitoring load is always conservative in representing the FIFO state, no timing anomalies are exhibited by this analysis. That is, determining the individual worst-case stalls results in the global worst-case stalls.

4.4 MILP Formulation

The proposed FIFO model requires solving an optimization problem to obtain the worst-case stalls, where the input and output monitoring loads, li_M and lo_M, and the monitoring stalls, s_M, need to be determined for each node. Here, we show how the problem can be formulated using MILP. Although the equations for lo_M and s_M are non-linear, they are piecewise linear. Previous work has shown that linear constraints for piecewise linear functions can be formulated using MILP [16]. In the following constraints, all variables are assumed to be lower bounded by zero unless otherwise specified, as is typically assumed for MILP.

First, a set of variables, lo' and s', are created to represent the unbounded versions of lo and s. For readability, the per block subscript M has been omitted.

$$s' = li + \Delta l - l_{max},\ s' \in (-\infty, \infty)$$
$$lo' = li + \Delta l,\ lo' \in (-\infty, \infty)$$

The following piecewise linear function calculates s from s'.

$$s = f(s') = \begin{cases} 0, & s' < 0 \\ s', & s' \ge 0 \end{cases}$$

This function can be described in MILP using the following set of constraints.

$$\begin{aligned} a_s\lambda_0 + b_s\lambda_2 &= s' \\ \lambda_0 + \lambda_1 + \lambda_2 &= 1 \\ \delta_1 + \lambda_2 &\le 1 \\ \delta_2 + \lambda_0 &\le 1 \\ \delta_1 + \delta_2 &= 1 \\ b_s\lambda_2 &= s \end{aligned}$$

where a_s is chosen to be less than the minimum possible value of s' and b_s is chosen to be greater than the maximum possible value of s'. The choice of a_s and b_s is arbitrary as long as it meets these requirements. λ_i are continuous variables and δ_i are binary variables. In this set of constraints, s' is expressed as a sum of the endpoints of a segment of the piecewise function. The δ_i variables ensure that only the segment corresponding to s' is considered. $\delta_1 = 1$ corresponds to the $s' < 0$ segment of $f(s')$ and $\delta_2 = 1$ corresponds to the $s' \ge 0$ segment of $f(s')$. The λ_i variables represent exactly where s' falls on the domain of that segment. s can be calculated using this information and the values of the function at the segment endpoints.

Similarly, lo can be bound between 0 and l_{max} by using the following set of constraints.

$$\begin{aligned} a_l\lambda_3 + l_{max}\lambda_5 + b_l\lambda_6 &= lo' \\ \lambda_3 + \lambda_4 + \lambda_5 + \lambda_6 &= 1 \\ 2\delta_3 + \lambda_5 + \lambda_6 &\le 2 \\ 2\delta_4 + \lambda_3 + \lambda_6 &\le 2 \\ 2\delta_5 + \lambda_3 + \lambda_4 &\le 2 \\ \delta_3 + \delta_4 + \delta_5 &= 1 \\ l_{max}\lambda_5 + l_{max}\lambda_6 &= lo \end{aligned}$$

As before, a_l and b_l are chosen such that $lo' \in (a_l, b_l)$. Again, λ_i are continuous variables and δ_i are binary variables.

Finally, for each node, the input monitoring load li_M must be determined. li_M depends on the previous nodes, $\mathcal{M}_{\text{prev}}$. If there is only one edge into the node, then li_M is simply

$$li_M = lo_{M_{prev}}$$

When there is more than one edge into node M, one set of constraints is used to lower bound li_M by all $lo_{M_{prev}}$.

$$li_M \ge lo_{M_{prev}},\ \forall M_{prev} \in \mathcal{M}_{\text{prev}}$$

Then, another set of constraints upper bounds li_M by the maximum $lo_{M_{prev}}$,

$$li_M - b \cdot \delta_{M_{prev}} \le lo_{M_{prev}},\ \forall M_{prev} \in \mathcal{M}_{\text{prev}}$$
$$\sum_{M_{prev} \in \mathcal{M}_{\text{prev}}} \delta_{M_{prev}} = |\mathcal{M}_{\text{prev}}| - 1$$

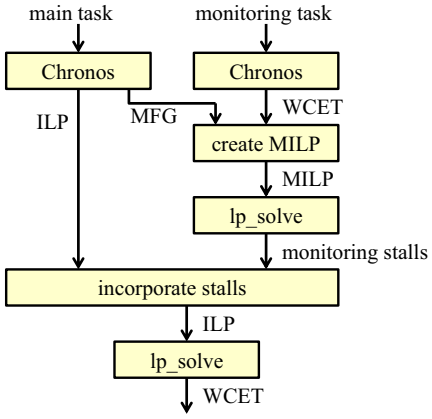

Figure 2: Toolflow for WCET estimation of parallel monitoring.

where b is chosen to be greater than $[\max(lo_{M_{prev}}) - \min(lo_{M_{prev}})]$ and $|\mathcal{M}_{\text{prev}}|$ is the number of nodes with edges pointing to M. δ_i are binary variables. The use of the binary variables δ_i and the second constraint ensure that li_M is only upper bound by one of the $lo_{M_{prev}}$. In order for all constraints to hold, this must be the maximum $lo_{M_{prev}}$. Together with the lower bound constraints, these constraints result in $li_M = \max(lo_{M_{prev}})$. For an example of this complete analysis with numbers, see Appendix A.

5. EVALUATION

5.1 Experimental Setup

Our toolflow for the proposed WCET method is shown in Figure 2. We first use Chronos [8], an open source WCET tool, to estimate the WCET for the main task and the monitoring tasks. We also modified Chronos to produce a MFG of the main task. This MFG and the monitoring task WCET are used to produce an MILP formulation as in Section 4. This MILP problem is solved using lp_solve [3], which produces the worst-case monitoring stall cycles for each forwarded instruction. These monitoring stalls are combined into the ILP formulation that is originally generated for the main task to estimate the overall WCET with parallel run-time monitoring. Although we use Chronos and lp_solve for our implementation, these components can be replaced with any WCET estimation tool and LP solver respectively.

To evaluate the effectiveness of our WCET scheme, we compared its estimate with a simple WCET bound from sequential monitoring (Section 4.2) as well as simulation results using the SimpleScalar tool [2]. In addition to the WCET estimates with monitoring, we also compared the results with the WCET of the main task without monitoring, using both Chronos and simulations.

For the experiments, we configured Chronos and SimpleScalar to model simple processing cores that execute one instruction per cycle for both main and monitoring cores and used an 8-entry FIFO. This configuration represents typical embedded microcontrollers, and is designed to focus on the impact of parallel run-time monitoring by removing complex features such as branch prediction and caches. In the evaluation, we used seven benchmarks from the Mälarden WCET benchmark suite [7] and two monitoring techniques: uninitialized memory checks (UMC) and control flow protection (CFP). UMC detects a software bug that reads memory without a write as briefly explained in Section 3. CFP protects a program's control flow by checking a target address on each control transfer [1]. In this technique, a compiler determines a set of valid targets for each branch and jump in the main task. This information is stored on the monitoring core. On a branch or jump, the monitoring core ensures that the target is contained in the list of valid targets.

5.2 Results

Table 1 shows the experimental results for each benchmark under different configurations. The first set of rows show the WCET estimate from Chronos (`wcet-none`) and actual run-times from simulations (`sim-none`) without monitoring. The remaining rows show the WCET for the UMC and CFP monitoring extensions. The results are shown for three different approaches: a bound from sequential monitoring (`sequential`), our approach (`wcet`), and simulations (`sim`). The numbers indicate the number of clock cycles. Appendix B includes running times for these experiments.

Table 2 shows relative comparisons between different configurations or WCET methods. The first set of rows compare the WCET estimates from ILP or MILP formulations with the worst-case simulation cycles for each monitoring setup. The results show that the analytical WCET estimates from our proposed scheme are larger than the observed WCET by 0% to 52% for UMC and 0% to 71% for CFP, depending on the main task. This difference is comparable to the case without parallel run-time monitoring, where the analytical WCET from Chronos is larger than simulation results by 0% to 52%. In fact, for `expint`, the majority of the difference is from the WCET estimate of the main task rather than the effects of monitoring. This result suggests that our WCET approach is not significantly more conservative than the baseline WCET tool for the main task.

The second set of rows compare the bound from sequential monitoring and the WCET from our proposed method. For UMC, our approach shows up to a 74% reduction in WCET estimates over the simple bound. Similarly, for CFP, our method shows up to a 73% improvement. These results demonstrate that modeling the FIFO decoupling between the main and monitoring tasks is important for obtaining tight WCET estimates of parallel monitoring.

Finally, the last two rows in Table 2 compare the WCET estimates with and without run-time monitoring. The results show that the increase in WCET varies significantly depending on benchmark and monitoring technique. Benchmarks with infrequent monitoring events (forwarded instructions) show minimal overheads while ones with frequent monitoring can see significant impacts. Also, the benchmarks with large WCET increases differ between UMC and CFP. Therefore, when applying parallel run-time monitoring techniques to real-time systems, a careful WCET analysis for the given tasks and monitoring techniques needs to be performed.

The impact of run-time monitoring on the execution time in our experiments (up to 3.48x in UMC and 2.58x in CFP) is roughly in line with previous studies on multi-cores without any hardware support [4, 12]. The performance overheads will be much lower for multi-cores with optimizations [4] or heterogeneous monitors [6]. Our analysis technique does not depend on any specific monitoring core microarchitecture and is applicable to more optimized architectures.

6. CONCLUSION

Parallel run-time monitoring techniques are an attractive

Monitoring	Experiment	Benchmark						
		cnt	expint	fdct	fibcall	insertsort	matmult	ns
None	wcet-none	64531	3483	1805	245	598	133668	5951
	sim-none	62931	2293	1805	245	598	133668	5951
UMC	sequential-umc	103052	3591	4382	257	2489	357453	10338
	wcet-umc	64550	3498	3035	245	2083	256120	5953
	sim-umc	62931	2297	2564	245	1864	235120	5951
CFP	sequential-cfp	151732	11669	1976	794	1174	231507	18623
	wcet-cfp	93544	8984	1805	547	677	133668	13614
	sim-cfp	72540	5247	1805	382	598	133668	9824

Table 1: Estimated and observed WCET (clock cycles) with and without monitoring.

Ratio			Benchmark							min	max	geomean
			cnt	expint	fdct	fibcall	insertsort	matmult	ns			
wcet-none	:	sim-none	1.03	1.52	1.00	1.00	1.00	1.00	1.00	1.00	1.52	1.07
wcet-umc	:	sim-umc	1.03	1.52	1.18	1.00	1.12	1.09	1.00	1.00	1.52	1.12
wcet-cfp	:	sim-cfp	1.29	1.71	1.00	1.43	1.13	1.00	1.39	1.00	1.71	1.26
sequential-umc	:	wcet-umc	1.60	1.03	1.44	1.05	1.19	1.40	1.74	1.03	1.74	1.33
sequential-cfp	:	wcet-cfp	1.62	1.30	1.09	1.45	1.73	1.73	1.37	1.09	1.73	1.45
wcet-umc	:	wcet-none	1.00	1.00	1.68	1.00	3.48	1.92	1.00	1.00	3.48	1.41
wcet-cfp	:	wcet-none	1.45	2.58	1.00	2.23	1.13	1.00	2.29	1.00	2.58	1.55

Table 2: Ratios comparing results from different experiments.

solution for improving the safety and reliability of future real-time systems. Before these solutions can be applied, the WCET impact of these techniques must be analyzed. In this paper we have presented a method for estimating the WCET for tasks running on a parallel monitoring system. We have shown how the non-linear FIFO behavior can be modeled as an MILP problem to produce the worst-case monitoring stall cycles. These can then be incorporated into traditional IPET methods for WCET estimation. Our evaluation of the method shows significant improvements over an estimate assuming sequential execution of the monitoring. In addition, the amount of overestimation is comparable to the overestimation for a system without parallel monitoring. Appendix C discusses some future directions for this work.

Acknowledgments

This work was partially supported by the National Science Foundation grants CNS-0746913 and CNS-0708788, the Air Force grant FA8750-11-2-0025, the Office of Naval Research grant N00014-11-1-0110, the Army Research Office grant W911NF-11-1-0082, and an equipment donation from Intel.

7. REFERENCES

[1] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Secure Embedded Processing Through Hardware-Assisted Run-Time Monitoring. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2005.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 2002.

[3] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve Version 5.5. `http://lpsolve.sourceforge.net/5.5/`.

[4] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. Gibbons, T. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proceedings of the 35th International Symposium on Computer Architecture*, 2008.

[5] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective Memory Protection Using Dynamic Tainting. In *Proceedings of the* 22^{nd} *International Conference on Automated Software Engineering*, 2007.

[6] D. Deng, D. Lo, G. Malysa, S. Schneider, and G. Suh. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proceedings of the 43rd International Symposium on Microarchitecture*, 2010.

[7] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[8] X. Li, Y. Liang, T. Mitra, and A. Roychoudury. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming*, 2007.

[9] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd Conference on Design Automation*, 1995.

[10] M. Lv, W. Yi, N. Guan, and G. Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *Proceedings of the 31st Real-Time Systems Symposium*, 2010.

[11] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the 40th International Symposium on Microarchitecture*, 2007.

[12] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta. Dynamic Information Flow Tracking on Multicores. In *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures*, 2008.

[13] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[14] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th International Symposium on Microarchitecture*, 2006.

[15] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. INDRA: An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[16] G. Sierksma. *Linear and Integer Programming*, pages 237–239. Marcel Dekker, Inc., 2002.

[17] G. E. Suh, J. Lee, D. X. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[18] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 2008.

[19] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

APPENDIX

A. EXAMPLE OF MILP-BASED METHOD

In this section we show a detailed example of applying our MILP-based method for estimating the WCET of a task running on a system with parallel run-time monitoring.

A.1 Example Setup

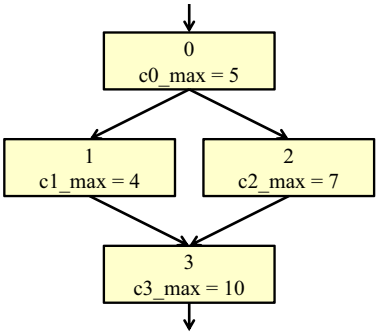

Figure 3: Control flow graph of a main task.

The control flow graph for an example main task is shown in Figure 3. We assume that the execution time for each node has already been calculated using previous methods. These execution times are labeled as `cB_max` in the figure.

In this example, let us assume that the monitoring technique requires loads and stores to be forwarded, as in the case of UMC. The monitoring task requires 5 cycles to handle a load and 7 cycles to handle a store. Thus, the maximum execution time of the monitoring task, $t_{M,max}$, is 7 cycles.

Because of the simplicity of the example, we assume that the FIFO only holds one entry ($n_F = 1$). Thus, $l_{max} = n_F \cdot t_{M,max} = 7$.

A.2 Creating the MFG

The first step is to create the monitoring flow graph. For each node in the CFG, the code represented by that node is analyzed. After any forwarded instruction, in this case any load or store instructions, an edge is created, dividing a node into 2 new ones. For example, the assembly-level code for node 1 in the CFG is shown below.

node 1

```
1    add $t0, $t1, $t2
2    add $t3, $t4, $t5
3    lw  $t4, 0($t3)
4    add $t0, $t0, $t4
```

Since the third instruction is a load instruction, node 1 must be split into two nodes in the MFG. The first node represents the first three instructions and the second node represents the last instruction.

The full MFG is shown in Figure 4. Nodes that are blue (dark) include a forwarded instruction, which is located at the end of the node. Nodes that are yellow (light) do not include a forwarded instruction. The nodes in the graph are labeled with minimum (`cB_min`) rather than maximum execution times. It can be seen that node 1 from the CFG corresponds to nodes 1.0 and 1.1 in the MFG. In this example, nodes in the CFG were only transformed into at most 2 nodes in the MFG. However, in general, a CFG node will

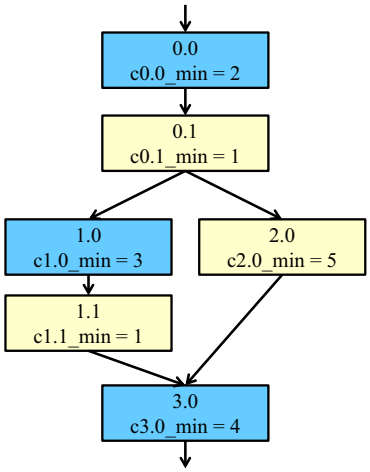

Figure 4: Monitoring flow graph of the main task. Blue (dark) nodes indicate ones with a forwarded instruction at the end. Yellow (light) nodes indicate ones without a forwarded instruction.

be transformed into a number of nodes in the MFG equal to the number of forwarded instructions plus one.

A.3 Calculating the Monitoring Load

Once the MFG is constructed, a set of MILP constraints is generated for each node. This process can be automated, but for this example we will construct the constraints for one node by hand. Specifically we will consider node 3.0 in the MFG. We will also calculate, by hand, the MILP solution for the node using some assumed values for variables associated with other nodes. Note that all variables are assumed to be non-negative unless otherwise specified.

Calculating input monitoring load: First, we will determine the worst-case input monitoring load for node 3.0, $li_{3.0}$. One set of constraints lower bounds the monitoring load by all possible incoming monitoring loads.

$$li_{3.0} \geq lo_{1.1}$$
$$li_{3.0} \geq lo_{2.0}$$

Then, a set of constraints upper bounds this input monitoring load.

$$li_{3.0} - 1000\delta_{1.1} \leq lo_{1.1}$$
$$li_{3.0} - 1000\delta_{2.0} \leq lo_{2.0}$$
$$\delta_{1.1} + \delta_{2.0} = 1$$

Here, the value 1000 is chosen arbitrarily but is known to be greater than $|lo_{2.0} - lo_{1.1}|$. A different value could have been chosen as long as this condition was true. $\delta_{1.1}$ and $\delta_{2.0}$ are binary variables which can only assume values of 0 or 1. To see how these constraints work, suppose that $li_{2.0} = 7$ and $li_{1.1} = 4$. The constraints are then evaluated as

$$li_{3.0} \geq 4$$
$$li_{3.0} \geq 7$$
$$li_{3.0} - 1000\delta_{1.1} \leq 4$$
$$li_{3.0} - 1000\delta_{2.0} \leq 7$$
$$\delta_{1.1} + \delta_{2.0} = 1$$

The first pair of constraints ensures that $li_{3.0} \geq 7$. This

means that for the third constraint to hold, $\delta_{1.1} = 1$. If $\delta_{1.1} = 1$, then by the last constraint, $\delta_{2.0} = 0$. Plugging this value into the fourth constraint gives $li_{3.0} \leq 7$. Thus the only possible solution is $li_{3.0} = 7$.

Calculating output monitoring load: In order to determine the output monitoring load for node 3.0, we must first calculate the change in monitoring node, $\Delta l_{3.0}$. Since there is a forwarded instruction in node 3.0,

$$\begin{aligned}\Delta l_{3.0} =& t_{M,max} - c_{3.0,min}\\ =& 7 - 4 = 3\end{aligned}$$

We first create a variable, $lo'_{3.0}$ to represent the unbounded output monitoring load.

$$lo'_{3.0} = li_{3.0} + \Delta l_{3.0},\ lo'_{3.0} \in (-\infty, \infty)$$

Using the example input monitoring load previously calculated of $li_{3.0} = 7$, this unbounded output monitoring load is $lo'_{3.0} = 7 + 3 = 10$. Then, the following set of constraints determines the bounded output monitoring load, $lo_{3.0}$.

$$-1000\lambda_3 + 7\lambda_5 + 1000\lambda_6 = 10 \quad (1a)$$
$$\lambda_3 + \lambda_4 + \lambda_5 + \lambda_6 = 1 \quad (1b)$$
$$2\delta_3 + \lambda_5 + \lambda_6 \leq 2 \quad (1c)$$
$$2\delta_4 + \lambda_3 + \lambda_6 \leq 2 \quad (1d)$$
$$2\delta_5 + \lambda_3 + \lambda_4 \leq 2 \quad (1e)$$
$$\delta_3 + \delta_4 + \delta_5 = 1 \quad (1f)$$
$$7\lambda_5 + 7\lambda_6 = lo_{3.0} \quad (1g)$$

The -1000 and 1000 values were chosen arbitrarily and only require that $lo'_{3.0}$ to fall between them. δ_3, δ_4, and δ_5 are binary variables. By Constraint 1b, it can be seen that all λ_i are less than or equal to 1. Thus, in order for Constraint 1a to hold, $\lambda_6 > 0$. Since $\lambda_6 > 0$, Constraints 1c and 1d force δ_3 and δ_4 to both be zero. From this, by Constraint 1f, $\delta_5 = 1$. Then, by Constraint 1e, λ_3 and λ_4 are both forced to be zero. If we now go back to the first two constraints, they are reduced to

$$\begin{aligned}7\lambda_5 + 1000\lambda_6 =& 10\\ \lambda_5 + \lambda_6 =& 1\end{aligned}$$

Solving this system of equations gives the solution $(\lambda_5, \lambda_6) = (0.997, 0.003)$. Plugging these values into Constraint 1g,

$$\begin{aligned}lo_{3.0} =& 7\lambda_5 + 7\lambda_6\\ =& 7 \cdot 0.997 + 7 \cdot 0.003\\ =& 7\end{aligned}$$

Thus, the output monitoring load is indeed bound by the maximum monitoring load of 7. Although this may seem to be a complicated series of calculations to determine this obvious result, this set of constraints is required in order for the piecewise linear, and thus non-linear, bounding function to be expressed in an MILP problem.

Calculating the monitoring stall cycles: The one remaining value that needs to be determined for node 3.0 is the monitoring stall cycles. Based on our previous calculations, the worst-case input monitoring load ($li_{3.0}$) is 7, the change in monitoring load ($\Delta l_{3.0}$) is 3, and the maximum monitoring load (l_{max}) is 7. Thus, we expect the worst-case monitoring stall cycles to be $(7 + 3) - 7 = 3$. To handle this as an MILP problem, first the unbounded monitoring stall cycles, s', is calculated.

$$\begin{aligned}s'_{3.0} =& li_{3.0} + \Delta l_{3.0} - l_{max},\ s'_{3.0} \in (-\infty, \infty)\\ =& 7 + 3 - 7 = 3\end{aligned}$$

In this case, since $s'_{3.0}$ is positive, we expect $s_{3.0} = s'_{3.0}$. The MILP problem determines $s_{3.0}$ using the following set of constraints.

$$-1000\lambda_0 + 1000\lambda_2 = 3 \quad (2a)$$
$$\lambda_0 + \lambda_1 + \lambda_2 = 1 \quad (2b)$$
$$\delta_1 + \lambda_2 \leq 1 \quad (2c)$$
$$\delta_2 + \lambda_0 \leq 1 \quad (2d)$$
$$\delta_1 + \delta_2 = 1 \quad (2e)$$
$$1000\lambda_2 = s_{3.0} \quad (2f)$$

The -1000 and 1000 values are chosen arbitrarily, only requiring that $s'_{3.0}$ is between them. From Constraint 2a, λ_2 must be positive. Since δ_i are binary variables, Constraint 2c then implies that $\delta_1 = 0$. Constraints 2d and 2e then force $\delta_2 = 1$ and $\lambda_0 = 0$. The first two constraints then reduce to

$$\begin{aligned}1000\lambda_2 =& 3\\ \lambda_1 + \lambda_2 =& 1\end{aligned}$$

Solving this system of equations leads to $(\lambda_1, \lambda_2) = (0.997, 0.003)$ and thus calculating $s_{3.0}$ using Constraint 2f:

$$\begin{aligned}s_{3.0} =& 1000\lambda_2\\ =& 1000 \cdot 0.003 = 3\end{aligned}$$

This is the value for s that we expected. If s' had instead been negative, then δ_1 would be forced to 1 and λ_2 would be forced to 0. From the last constraint, it can be seen that if λ_2 is 0, then s is also 0.

A.4 MILP Optimization

In the previous subsection, the monitoring loads for one node were calculated in detail. However, note that the output monitoring load for each node with an edge pointing to node 3.0 was assumed to be a certain value. In an actual MILP problem, these would be variables that are also being solved for. Solving for these inter-related variables and determining the global maximum number of cycles stalled due to monitoring is impractical to do by hand. While the amount of calculations may seem excessive for these simple examples, the ability to formulate the problem in MILP is essential in order to solve large problems.

B. TIME TO SOLVE LINEAR PROGRAMMING PROBLEM

The most time intensive portion of the WCET analysis is the actual solving of the linear programming (LP) problem. For our experiments, we used lp_solve 5.5.2.0 [3] as our LP solver. These experiments were run on a 2.67 GHz Xeon E5430 quad-core processor with 4 GB of RAM. The running times for lp_solve are shown in Table 3. The first set of rows show the running time for determining the worst-case stalls from the monitoring flow graph (`stall`). The second set of rows show the lp_solve running time for finding the sequential bounds. The final set of rows show the running

Solver Target	Benchmark							min	max	geomean
	cnt	expint	fdct	fibcall	insertsort	matmult	ns			
stall-umc	17.789	6.256	21.733	0.043	0.39	161.796	3.655	0.043	161.796	4.224
stall-cfp	3.691	97.93	0.038	0.024	0.025	14.209	1.474	0.024	97.930	0.778
sequential-umc	0.006	0.004	0.004	0.005	0.002	0.004	0.006	0.002	0.006	0.004
sequential-cfp	0.007	0.001	0.003	0.002	0.003	0.006	0.003	0.001	0.007	0.003
wcet-none	0.003	0.003	0.004	0.002	0.002	0.002	0.001	0.001	0.004	0.002
wcet-umc	0.004	0.004	0.003	0.001	0.004	0.005	0.002	0.001	0.005	0.003
wcet-cfp	0.002	0.007	0.005	0.004	0.003	0.005	0.004	0.002	0.007	0.004

Table 3: Running time of lp_solve in seconds to determine worst-case stalls (stall), sequential bound (sequential), and worst-case execution times (wcet).

time for determining the overall WCET (`wcet`). For `wcet-umc` and `wcet-cfp`, this is for the ILP problem given the worst-case stalls .

The running times for the `sequential` cases and the `wcet` cases are very similar. This is because these cases are all solving essentially the same problem with different numbers. That is, for a given benchmark, these different cases are all solving a linear programming problem for the same control flow graph (CFG). As a result, the number of variables and the set of constraints is the same, though the WCET for each basic block changes depending on the extension and the estimation method. The `stall` cases have a longer running time. This is due to the fact that a MFG has more nodes than its corresponding CFG. The increased number of nodes also implies more variables and more constraints.

C. FUTURE WORK

There are two main directions for future work. One direction for future work is to tighten the WCET bound and the other is to improve the time needed to solve the linear programming (LP) problem. The WCET bound could be improved by incorporating more detailed information about the main task. Program behavior such as infeasible paths and loop bounds have previously been studied in the IPET context [18]. Incorporating this information into the WCET analysis for parallel monitoring can decrease the worst-case stall cycles found. For example, the current formulation does not include any notion of loop bounds. As a result, for a loop that increases the monitoring load, the worst-case conclusion is that there are enough loop iterations for the FIFO to become full. With information about loop bounds, it may be the case that certain loops do not cause the FIFO to fill completely. We believe that since our formulation uses a linear programming approach similar to IPET, additional program behavior can be added in a similar manner using additional constraints. Along these lines, another possible direction for future work is to extend this work for additional architectural features. For example, one assumption in this work was that the main and monitoring cores had separate memory spaces. It would be interesting to extend this WCET analysis to a system where the main and monitoring cores shared memory.

Appendix B shows the running time for solving the LP problems created. Determining the worst-case stalls requires a longer run time than determining the WCET. This is due primarily to the larger graph size of the MFG compared to the CFG. The larger graph size means that there are more variables to optimize over. It may be possible to model the monitoring load for each basic block in such a way that the optimization problem can remain at the CFG graph size. Since the monitoring load calculations for a series of MFG nodes, without branch entries or exits, is relatively straightforward, it may be possible to "collapse" them into a set of equations for one node. However, care must be taken that these simplifications do not remove any worst-case possibilities.

Finally, we mention the possibility of combining the worst-case stall cycles MILP problem and WCET ILP problem into a single LP problem. Combining the two problems into a single optimization may provide improvements in tightening the WCET bound. It may also improve the LP running time by requiring only one LP problem to be solved. At first glance, this may seem possible by combining the constraints from both problems and maximizing the objective

$$t = \sum_{B \in \mathcal{B}_{CFG}} N_B \cdot (c_{B,max} + s_{B,max})$$

from Section 4.1. However, combining the problems means that this is optimizing over both N_B and $s_{B,max}$. These variables form a product term in the equation for t and so the optimization objective is no longer linear. There may exist a method to formulate a combined problem that has a linear objective. Alternatively, non-linear programming techniques may serve as a solution.