# Improving Worst-Case Cache Performance through Selective Bypassing and Register-Indexed Cache

Mohamed Ismail, Daniel Lo, and G. Edward Suh
Cornell University
Ithaca, NY, USA
{mii5, dl575, gs272}@cornell.edu

## ABSTRACT

Worst-case execution time (WCET) analysis is a critical part of designing real-time systems that require strict timing guarantees. Data caches have traditionally been challenging to analyze in the context of WCET due to the unpredictability of memory access patterns. In this paper, we present a novel register-indexed cache structure that is designed to be amenable to static analysis. This is based on the idea that absolute addresses may not be known, but by using relative addresses, analysis may be able to guarantee a number of hits in the cache. In addition, we observe that keeping unpredictable memory accesses in caches can increase or decrease WCET depending on the application. Thus, we explore selectively bypassing caches in order to provide lower WCET. Our experimental results show reductions in WCET of up to 35% over the state-of-the-art static analysis.

## 1. INTRODUCTION

Real-time systems are becoming increasingly prevalent as our physical world continues to become more computerized. One essential component to the design of real-time systems is the analysis of the worst-case execution time (WCET) of tasks. WCET analysis is used in many real-time scheduling algorithms and is essential in providing timing guarantees. As a result, a large amount of work has looked into reducing the WCET and providing accurate WCET estimates [1]. Low WCET allows for better utilization of the processor and improved schedulability.

Data caches, which are widely used in modern computing systems, introduce a particular challenge in providing a tight bound for the WCET. While caches often improve the average-case performance by exploiting spatial and temporal locality, they cannot improve the WCET estimate unless a cache access can be guaranteed to be a hit in all possible executions. As a result, the WCET analysis of caches needs to consider all possible memory access patterns, and its effectiveness is heavily dependent on the ability to predict memory access patterns. While accesses to the instruction cache are relatively easy to predict, accesses to the data cache often depend on memory addresses that can only be resolved at run-time. This makes it difficult to predict the worst-case performance of the data cache.


*DAC '15* June 07 - 11 2015, San Francisco, CA, USA

ACM 978-1-4503-3520-1/15/06 ... $15.00
http://dx.doi.org/10.1145/2744769.2744855

Although data cache analysis for WCET has been studied [2, 3, 4, 5, 6, 7, 8, 9], existing techniques still result in overly conservative worst-case performance, in particular because of their inefficiency in handling memory accesses whose addresses cannot be predicted statically. Static cache analysis methods need to be conservative and assume that an unpredictable access can either evict any cache block or update the recency (i.e., LRU) history of any set, which makes it rather difficult to guarantee a hit on following accesses. Alternatively, researchers have proposed to have unpredictable memory accesses bypass caches. Yet, this approach can lead to an excessive WCET estimate when there exists a large number of unpredictable accesses because all unpredictable accesses must pay the latency of accessing main memory.

In this paper, we present a new approach to improve the worst-case performance guarantee for data caches by designing cache hardware and its worst-case static analysis together. Given that today's WCET analysis cannot effectively handle memory accesses with unpredictable addresses, we first propose a new auxiliary cache structure that is specifically designed to enable worst-case analysis for unpredictable addresses. We call this new cache structure a *register-indexed* cache. Intuitively, the cache and its analysis method exploit the observation that the spatial and temporal locality between nearby memory accesses can often be seen without knowing exact memory addresses. For example, a program may access multiple elements in an array using the same base address with different offsets. The new cache is indexed by a register number instead of a memory address, and allows static analysis of cache hits based on temporal and spatial locality between memory accesses that use the same register as a base address, even when the base address is unknown.

In addition to the *register-indexed* cache, we also note that using a fixed policy for unpredictable memory accesses to either bypass or use data caches is suboptimal; the effectiveness of bypassing a cache heavily depends on application memory access patterns and the cache configurations of a target system. For example, static analysis may be able to guarantee cache hits for memory accesses to an array for a large cache, but not for a small cache. For the lowest possible WCET guarantee, memory accesses need to be *selectively bypassed* based on whether the accesses lead to an improved static estimate of the worst-case performance for a given access pattern and cache configuration combination.

We evaluated our selective bypassing technique on various benchmark programs and found that we can improve the WCET by up to 24.3% over the state-of-the-art static analysis. Additionally, experimental results show that we can reduce the WCET by up to 35.3% by including both

selective bypassing and the register-indexed cache for unpredictable memory accesses.

This paper has the following main contributions:

- The paper presents a novel cache structure along with an analysis algorithm that can allow guaranteed cache hits in static analysis even for memory accesses whose addresses are considered statically unpredictable.
- The paper notes that, in order to reduce the WCET, memory accesses should selectively bypass data caches based on how they affect the static analysis results.
- The paper evaluates the proposed schemes and shows that they can significantly reduce the WCET guarantee on certain applications.

This paper is organized as follows. Section 2 presents related work. Section 3 presents details of the proposed framework. Section 4 evaluates the proposed methodology and Section 5 concludes the paper.

## 2. RELATED WORK

To the best of our knowledge, there is no previous work that presents a cache structure that can be statically analyzed to ensure hits on accesses with unpredictable memory addresses. Previous work has also only explored a fixed policy for cache bypassing whereas we show that selective bypassing depending on static analysis is important to achieve low WCET guarantee. This section briefly summarizes previous work on data caches in the context of WCET.

Hard real-time systems often use scratchpads [10, 1, 11, 12] instead of data caches in order to have predictable memory access time while utilizing low-latency on-chip memory. However, scratchpads require software to explicitly manage data movement between on-chip and off-chip memory. These data movement instructions lead to additional overhead that does not exist for caches, and also imply that programs need to be re-compiled for each system depending on their scratchpad size. As a result, data caches that can be transparently used for many programs still represent an attractive option if their worst-case performance can be tightly bounded.

For the purposes of using caches in hard real-time systems, there has been extensive work on static analysis techniques that aim to predict memory addresses and model the worst-case cache performance. These techniques include abstract interpretation [8], persistent analysis [6], scope-aware analysis [3], and unified multi-level cache modeling [2]. While the static analysis techniques can often accurately predict cache misses and hits for memory accesses with known addresses, they remain overly conservative for accesses with unknown addresses. Typically, the analysis treats an unknown access as a cache flush in order to account for a potential cache eviction and a change to the LRU history. In this paper, we augment the static cache analysis to selectively bypass unpredictable accesses to obtain a better worst-case guarantee.

In addition to simply analyzing existing caches, cache operations can be slightly changed in order to provide better worst-case bounds. For example, Lundqvist and Stenström [13] proposed to bypass memory accesses to unpredictable data structures and showed more than a factor of two improvement on worst-case performance for data caches. Additionally, they also empirically observed that in the SPEC95 benchmark suite 84% of data accesses are predictable and quantified the effect of bypassing the cache on all unpredictable memory accesses [14]. However, these bypassed accesses always suffer the full latency to main memory and can negatively impact the worst-case execution time in some cases. This paper proposes to bypass selectively and introduces a new register-indexed cache that can reduce the number of worst-case cache misses for unpredictable accesses.

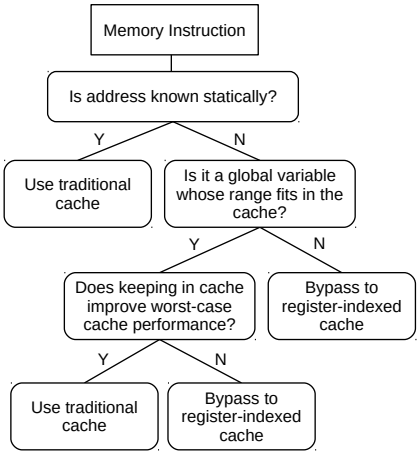


Figure 1: Overview of selective cache usage and bypassing based on memory access types.

Cache locking [15, 11] has also been proposed as a way to mitigate the effects of unpredictable memory accesses on worst-case cache performance. By locking certain cache blocks, accesses to those blocks can easily be analyzed without considering unpredictable memory accesses. However, cache locking does not help memory accesses such as unpredictable ones that are not explicitly locked, and requires explicit locking instructions that need to be inserted in a program, which reduces the transparency of caches.

## 3. WCET-AWARE CACHE DESIGN

### 3.1 Overview

In this section, we present our cache design that is targeted to provide a low WCET bound. While reducing the WCET by itself is important, our main goal is to optimize the worst-case performance that can be statically analyzed and guaranteed. The static analysis predicts cache operations based on the static knowledge of memory access patterns that are obtained through symbolic execution. As a result, the effectiveness of static analysis often depends on how predictable memory accesses and their addresses are. In this context, our scheme categorizes memory instructions based on their predictability and analyzability, and applies different techniques to obtain the worst-case performance for each type. The overall scheme is shown in Figure 1.

We can first separate memory instructions into ones whose data address can be determined statically and ones whose address is unknown without run-time information. For the instructions with known memory addresses, the only source of non-determinism in cache analysis across runs is through convergence of control flows with different cache states. Today's static analysis algorithms can model such accesses rather accurately and provide a reasonable bound on the worst-case performance. Therefore, our scheme places these accesses in a normal data cache and uses static analysis to

obtain the worst-case bound. We note that this class of instructions can also include instructions where the address is based on induction variables.

On the other hand, addresses for some memory instructions cannot be determined statically. For example, memory accesses whose address depends on input values cannot be precisely known until run-time. We categorize these memory instructions with unknown addresses into two types: ones whose address range is known and small enough to fit into the target cache, and the rest.

Although the specific address of a memory access may not be known, for some accesses, it is possible to bound the access to a range of addresses. For example, an instruction may access a global array, whose length is known and whose address can be obtained from the symbol table of the binary. In this case, although the index of an array access may not be known statically, the address range of this instruction can be bounded to the memory range of the array. If this range of addresses can fit in the cache, then static analysis may be able to use this information and lower the WCET. In fact, today's state-of-the-art static analysis [3] can indeed guarantee cache hits for certain accesses to a global variable within a predictable range.

However, not all memory accesses with a known range lead to better worst-case cache performance when included in the static analysis of a traditional cache. For example, accesses to global variables with large ranges can introduce extra conservatism in the static analysis. Because these accesses may map to multiple cache sets, the static analysis needs to assume that they can pollute many cache sets, effectively evicting many cache blocks that a program may access later. As a result, depending on the memory access pattern of a program and the cache size, using the data cache for memory accesses with a known range may or may not lead to a better WCET bound.

Therefore, our scheme selectively bypasses these memory accesses by comparing the static analysis result for cases with and without them. If the static analysis results in less cache misses without these accesses, the memory instructions are marked to bypass the normal cache and instead use the register-indexed cache, described in the next subsection. The bypass information is encoded as a single-bit tag per memory instruction and is used by the processor at runtime. In our current implementation, the bypass decision is made at a coarse grain; we either bypass all instructions with an unknown address but a known range or include all of them in the normal cache analysis. We note that this decision can be made at a finer granularity (i.e., instruction level). However, we found that even a coarse-grained decision can result in noticeable improvements to the WCET.

Finally, there exists memory instructions whose addresses cannot be determined or restricted at all statically. For these instructions, we always bypass the data cache and instead use the special register-indexed cache. The cache behavior of these instructions is particularly difficult to analyze statically because the data could potentially be placed in any cache set. The static analysis needs to assume that these accesses may evict a block in any set or change the LRU history, which leads to unpredictable replacement in the future. To be safe, today's static analysis techniques effectively flush the cache on such an unpredictable memory access, leading to an overly conservative estimate of the worst-case performance. Therefore, bypassing memory instructions with unknown memory addresses almost always improves the worst-case bound that the static analysis can provide.

```
ld r1, 0(r2) ; addr in r2 unknown
…
; Computation that does not modify r2
…
ld r3, 0(r2) ; mem[r2] can be predicted as hit
addi r2, r2, 4
ld r4, 0(r2) ; mem[r2 + 4] can be predicted as hit
```

Figure 2: Accesses which can be predictably cached despite unknown addresses.

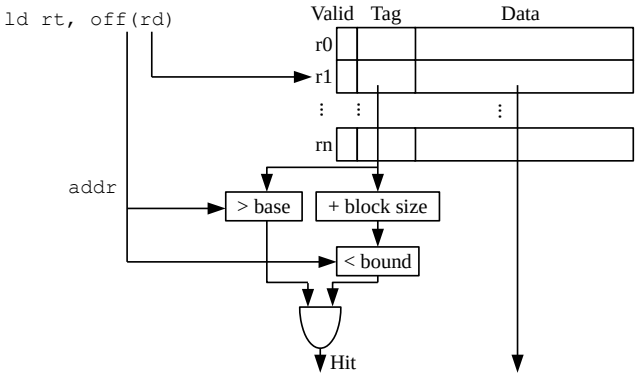


Figure 3: Block diagram of the register-indexed cache.

## 3.2 Register-Indexed Cache

Here, we present our new cache structure, named *register-indexed cache*, which is designed to lower the WCET for memory accesses with unpredictable addresses. We first describe the intuition, then show the cache structure and its operations, and finally discuss the static analysis for the proposed cache.

For memory accesses to unpredictable addresses, it may still be possible to cache some of these accesses to have predictable hits with a static guarantee. Specifically, we found that although the memory address of an initial access may be unknown, subsequent accesses using the same register as the base address may be statically analyzed. For example, consider the code shown in Figure 2. In this example, the address value in `r2` cannot be determined statically. However, if `r2` is not modified, then subsequent loads from address `r2` can be predicted to hit in a cache structure. Similarly, spatial locality can also be exploited if the offset from the initial unknown address is within a cache line. In the example, if the cache line for address `r2` includes `r2 + 4`, then the subsequent access to `r2 + 4` can be predicted as a hit. In order to prevent pollution of the traditional data cache and to be able to analyze these register-based predictable accesses, we designed a novel *register-indexed cache* architecture.

### 3.2.1 Cache Architecture

Figure 3 shows a block diagram of our register-indexed cache design. The basic idea is to index into the cache using the base address register number as opposed to using a portion of the memory address. For example, consider a memory instruction of the form: `ld r1, off(r2)`. Since `r2` is the base address register, the value `2` is used as the index into the cache.

Cache lines are typically larger than one word in order to exploit spatial locality. In traditional caches, a cache block stores a chunk of data with a fixed memory alignment no

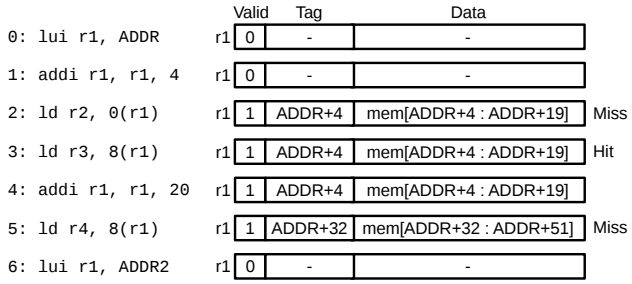


Figure 4: Example of register-indexed cache operation.


Figure 5: Example of WCET analysis for register-indexed cache.

matter which part of the block is accessed on a cache miss. For example, a cache with 32-byte blocks will always be aligned to 32-byte blocks in memory. However, given unpredictable base addresses, this construction does not allow static analysis to detect cache hits based on spatial locality; the static analysis cannot guarantee whether other accesses with a different offset will hit or not. For example, for the code example in Figure 2, if `r2` points to a location at the end of a memory-aligned block, then `r2 + 4` will be a cache miss. On the other hand, if `r2` points to the start of a memory-aligned block, then `r2 + 4` will be a cache hit. In order to enable static analysis of spatial locality in the worst-case, a cache block in the register-indexed cache is aligned to the memory address that initially loads the cache block. In the code example, a 32-byte cache block will store a 32-byte chunk starting from the location pointed by `r2`. This implies that a miss on the register-index cache may require two reads from memory in order to fill the cache line.

Since cache lines are not memory-aligned, the stored tag cannot simply be the most significant bits of the memory address. Instead, the tag is the full address of the first byte stored in the cache line. On a cache access, the memory address of the access is checked against the address range between the tag as the base and the tag plus block size as the bound to determine whether the access is a hit or miss.

One issue that arises with the register-indexed cache is that a memory location could be found in multiple cache lines. In order to handle this aliasing, on a write to the register-indexed cache, all cache lines are checked to see if they contain the address being written to. Any lines that include the address are updated with the new value. Since all cache lines are kept up-to-date, no modification is needed when reading to account for aliased locations. Note that the register-index cache only has a small number of cache blocks, one for each register that can be used as a memory base address. By assuming a write-through policy in our design, cache lines can be updated in parallel with the write to memory. For example, updating two lines per cycle allows us to update 32 cache lines in the register-indexed cache within 16 cycles which is faster than the worst-case access time of main memory. This allows us to simply use the worst-case latency of a write to memory when analyzing store instructions.

### 3.2.2 Run-Time Operation

Figure 4 shows an example of how the register-indexed cache operates. The left side shows a series of instructions and the right side shows the state of the cache line corresponding to architectural register `r1` after each instruction. For this example, we assume a cache with a block size of four words (16 bytes) and one cache block per register. On instruction `0`, an unknown address is loaded into `r1`. This causes the `r1` cache line to be invalidated. Instruction `1` increments `r1` so the address for instruction `2` is `ADDR + 4`. Instruction `2` uses `r1` as the base address which indexes into the `r1` line of the register-indexed cache. Currently, this line is invalid so the access is a miss. The block of memory from `ADDR + 4` to `ADDR + 19` is loaded in the cache line and the tag is set as `ADDR + 4`.

Instruction `3` performs a load from `ADDR + 12` using register `r1`. This address falls in the range that is stored in the cache line and so the access is a hit. Instruction `4` increments `r1` to `ADDR + 24`. Thus, the load for instruction `5` accesses memory location `ADDR + 32`. This does not fall within the range stored in the cache line so the access is a miss. The cache line is evicted and replaced with the data from `ADDR + 32` to `ADDR + 51`. Finally, instruction `6` loads a new unknown address into `r1`. Since this will be unpredictable in analysis, we also invalidate the cache line.

### 3.2.3 WCET Analysis

The register-indexed cache was designed to capture memory accesses to statically unpredictable addresses. Thus, a major challenge in analyzing its worst-case performance is determining its hit/miss pattern without having information about absolute memory addresses. However, it is possible to analyze the cache access pattern by tracking relative addresses from an initial unknown register value. In order to perform this analysis, we introduce the idea of *delta values*.

The static analysis maintains a delta value for each architectural register. Whenever an unknown value is loaded into a register, the delta value of that register is reset to 0 and the corresponding cache line is invalidated. On increments and decrements to the register, the delta value is updated to represent the relative address from the initial unknown value. When this register is used as the base address on a miss to the register-indexed cache, the cache line that is brought in is associated with the current delta value of the register plus the memory instruction offset. That is, rather than storing an absolute tag address, which is unknown at analysis time, the relative address (delta + offset) is stored instead. On subsequent accesses to the register-indexed cache, the analysis checks whether the current delta value plus memory instruction offset resides within the range of the cache line.

Figure 5 revisits the code example from Figure 4 in order to show the WCET analysis. Note that this example essentially shows a symbolic execution in the static analysis. The left side shows the sequence of instructions and the right side shows the delta value and analyzed cache state corresponding to register `r1`. The analysis does not actually load data values into the cache state but the data field is

| Instruction Type | WCET Analysis |
|---|---|
| Load | Reset delta; Invalidate cache line |
| Inc./Dec. by known value | Update delta |
| Inc./Dec. by unknown value | Reset delta; Invalidate cache line |
| Other ALU Ops | Reset delta; Invalidate cache line |

Table 1: WCET analysis of register-indexed cache.

shown here to represent the range of relative memory addresses that analysis knows to be in the cache. When the initial unknown address is loaded into `r1`, delta for `r1` is reset to 0 and the corresponding cache line is invalidated. On instruction `1`, when `r1` is incremented, delta is updated to 4. Thus, on the load at instruction `2`, the "tag" stored is the relative address of 4. With this, the analysis is able to recognize that instruction `3`, which accesses a relative address of 8, will result in a hit in the cache. Instruction `4` again increments `r1`, so its delta value is updated to 24. This causes the load for instruction `5` to miss in the cache since it now accesses the relative address 32 (delta + offset). On this miss, it evicts the old cache block and updates its tag to represent that the cache block starts at relative address 32. Finally, instruction `6` shows another unknown address being loaded into `r1` which forces the analysis to reset the delta value for `r1` and invalidate the cache line.

Table 1 outlines the operations that the analysis performs on each instruction type. For increments and decrements by statically known values (e.g., `addi`, `subi`), the analysis updates the delta value of the appropriate register. However, if the increment/decrement is by an amount that cannot be statically determined, the WCET analysis must conservatively reset the delta register and invalidate the cache line. Similarly, for other ALU operations such as shifts and logical operations where the delta value would depend on the register value, the analysis conservatively resets delta and invalidates the cache line.

In static analysis, we must also handle the case of merging control flows for different execution paths. We use the following criteria to ensure correct worst-case behavior:

1. Do any of the incoming basic blocks perform an invalidation?
2. Is there a mismatch between the delta values?

If the answer to any of the above questions is a *yes*, then we reset the delta and assume that the cache line is invalid. Otherwise we merge the cache state in a similar way to traditional analysis methods (i.e., taking the intersection of the states accounting for proper LRU behavior).

Intuitively, the first question captures the case where only a subset of control paths perform an invalidation. To flatten the state, we propagate all invalidations through multiple paths in the control flow graph. For loops, we only consider the pre-head basic block in the first iteration. For other iterations, only the basic blocks in the loop are considered. The second question captures the case where one register can have multiple possible values depending on the control flow. By performing an invalidation on a mismatch, we can flatten the state for the cache line corresponding to the register.

# 4. EVALUATION

## 4.1 Experimental Setup

In order to evaluate our design, we compared the WCET estimate from our framework with the WCET of two state-of-the-art techniques: today's static analysis techniques with and without simple cache bypassing where all unpredictable accesses are bypassed to memory. We use Chronos [16], an open source WCET analysis tool, as our baseline static analysis tool. We modify Chronos to evaluate the simple bypassing as well as our selective bypassing scheme. We also implement the analysis for our register-indexed cache in Chronos by modifying the symbolic analysis to track deltas, tags, and invalidations corresponding to each register. We did not see significant increases in analysis time due to our modifications.

We evaluated our scheme using a subset of programs from the MiBench [17] and the Mälardalen [18] suites. By default, these benchmarks have a fixed input set and their memory access pattern is deterministic across runs. In order to evaluate unpredictable accesses, we modified the benchmarks to be analyzed with unknown input variables. This is more representative of real-world applications where input values will not be known until run-time. Some benchmarks did not show any unpredictable memory accesses, even with modifying the input variables, and so we do not show their results. Other benchmarks proved too complex to port and successfully execute within a reasonable analysis time even without our modifications to Chronos and thus are omitted.

For the baseline hardware, we assume a 5-stage in-order processor with single-level L1 instruction and data caches with 30 cycle miss penalties. For the cache configuration, we used specifications similar to the ARM R5 processor. The instruction cache is a 2-KB 2-way set-associative cache with 32-byte cache lines. The data cache is an 8-KB 4-way set-associative cache with 32-byte cache lines. For our register-indexed cache design, we also use 32-byte blocks and assume a 30 cycle miss penalty. Because the blocks are not memory-aligned, register-indexed cache misses may require two memory requests. To limit the miss penalty, we assume that the register-indexed cache returns data after the first memory request completes so that the execution can resume. With 32 architectural registers, the capacity of the register-indexed cache is 1 KB. We assume a write-through cache policy to simplify the analysis, which is common for studies on the worst-case cache analysis.

Based on a first-order estimate from CACTI [19] assuming a 40nm technology node, the area overhead of the register-indexed cache compared to the data cache is 15.3%. Similarly, the power overhead of the register-indexed cache is 12.0% compared to the data cache.

## 4.2 Selective Bypassing

We first run the analysis with just selective bypassing. All accesses which are bypassed go directly to memory. Since our selective bypassing balances between the overly conservative cache analysis and the long latency of bypassing, we achieve the same or better WCET for all benchmarks compared to the static analysis. Figure 6 shows the WCET results for different techniques normalized to the baseline Chronos. Even with a coarse-grain decision of whether to bypass or not, we find that selective bypassing can improve the WCET by as much as 24.9%. We also note that the simple bypassing policy can lead to a significant overhead for certain benchmarks (up to 84% overhead).

## 4.3 Register-Indexed Cache

Next, we evaluate using our register-indexed cache in the bypass path. The results show that the register-indexed

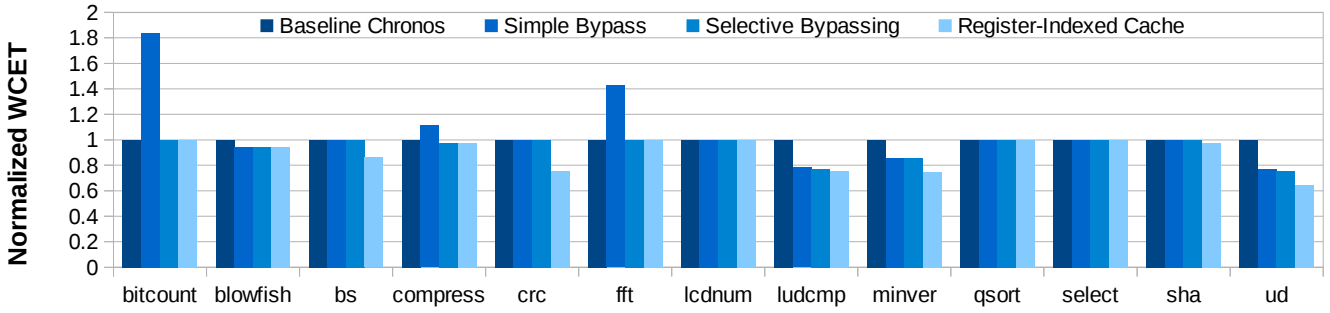


Figure 6: WCET with selective bypassing and register-indexed cache.

cache combined with the selective bypass can improve the WCET by as much as 35.3% compared to today's static analysis method. We inspected benchmark binaries to understand the sources of the improvements, and found that the benchmarks with a large number of accesses with unknown addresses that incremented in a loop showed the most improvements for the register-indexed cache.

Finally, we note that different cache configurations and memory latencies may yield different results. For example, increasing the block size of the register-indexed cache can further improve the results. We found that the best WCET improvement increased to 39.9% and 42.2% for 64-byte and 128-byte blocks. In addition, increased memory latencies highlight the need for predictable caching. For example, we found that with a doubled memory latency, the WCET improvement increased to up to 38.5%.

## 5. CONCLUSION

In this paper, we presented a framework to improve the worst-case performance guarantee for caches in real-time systems. Our approach uses selective bypassing of certain memory instructions along with a new register-indexed caching structure that can improve the memory access time of instructions whose memory addresses are unknown statically. Applying these two techniques, our results show up to 35.3% improvements in WCET.

## 6. ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation grant CNS-0746913, 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] 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.

[2] S. Chattopadhyay and A. Roychoudhury, "Unified Cache Modeling for WCET Analysis and Layout Optimizations," in *Proceedings of the 30th IEEE Real-Time Systems Symposium*, 2009.

[3] B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-Aware Data Cache Analysis for WCET Estimation," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.

[4] J. Blieberger, T. Fahringer, and B. Scholz, "Symbolic Cache Analysis for Real-Time Systems," *Real-Time Systems*, 2000.

[5] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon, "Timing Analysis for Data Caches and Set-Associative Caches," in *Proceedings of the 3rd Real-Time Technology and Applications Symposium*, 1997.

[6] C. Ferdinand and R. Wilhelm, "Efficient and Precise Cache Behavior Prediction for Real-Time Systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, Dec. 1999.

[7] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and Precise WCET Prediction by Separated Cache and Path Analyses," *Real-Time Systems*, vol. 18, no. 2/3, pp. 157–179, May 2000.

[8] M. Alt, C. Ferdin, F. Martin, and R. Wilhelm, "Cache Behavior Prediction by Abstract Interpretation," in *Science of Computer Programming*. Springer, 1996, pp. 52–66.

[9] C. Ferdinand, R. Heckmann, and R. Wilhelm, "Analyzing the worst-case execution time by abstract interpretation of executable code," in *Automotive Software–Connected Services in Mobile Networks*. Springer, 2006, pp. 1–14.

[10] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET Centric Data Allocation to Scratchpad Memory," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, 2005.

[11] I. Puaut and C. Pais, "Scratchpad Memories vs Locked Caches in Hard Real-Time Systems: A Quantitative Comparison," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2007.

[12] L. Wehmeyer and P. Marwedel, "Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2005.

[13] T. Lundqvist and P. Stenström, "A Method to Improve the Estimated Worst-Case Performance of Data Caching," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[14] T. Lundqvist and P. Stenström, "Empirical Bounds on Data Caching in High-Performance Real-Time Systems," Chalmers University of Technology, Tech. Rep., 1999.

[15] X. Vera, B. Lisper, and J. Xue, "Data Cache Locking for Higher Program Predictability," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003.

[16] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Annual IEEE International Workshop on Workload Characterization*, 2001.

[18] 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.

[19] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "Cacti 5.3," *HP Laboratories, Palo Alto, CA*, 2008.