THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRA5/09

Program Transformations for Predictable Cache Behavior

Bach Khoa Huynh, Lei Ju, Sudipta Chattopadhyay and Abhik Roychoudhury

May 2009

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Program Transformations for Predictable Cache Behavior

Bach Khoa Huynh Lei Ju Sudipta Chattopadhyay Abhik Roychoudhury
Department of Computer Science, National University of Singapore
{huynhbac, julei, sudiptac, abhik}@comp.nus.edu.sg

ABSTRACT

Real-time embedded software developers need to balance the dual (and seemingly conflicting) concerns of efficiency and predictability. Efficiency concerns are typically addressed by tuning the application and its underlying processing platform through a variety of techniques such as generating custom instructions in the instruction set, or configuring the processing platform. However, timing predictability remains a difficult goal to achieve, specifically in the presence of performance-enhancing micro-architectural features such as data caches. Presence of data caches can cause vast variation in the execution time for even programs with a single path.

In this paper, we study a new approach to achieve predictable cache behavior (without large performance degradation) in data-intensive embedded applications. Our approach is to rewrite a given application into a "cache-efficient" style, where the data memory accesses are tracked and transformed to systematically reduce data cache conflicts. Our program transformation leads to lesser execution time variation in the transformed program (across program inputs as well as across cache configurations). We also develop a new Worst-case Execution Time (WCET) analysis method for data caches, and show that it leads to tighter WCET estimates for cache-efficient programs. Our experiments indicate that adopting the cache-efficient style of programming for data-intensive embedded software can help balance the dual concerns of efficiency and predictability.

Keywords: Timing predictability, Timing Analysis, Cache-efficient algorithms.

1. INTRODUCTION

Over the past decades, a very substantial portion of computing research has been directed towards building techniques and platforms for high-performance software. Indeed, many of the advances in computer architecture such as caches, pipelines and branch prediction — optimize the average-case in a program, while introducing large variations in the execution time of a program across its inputs. As a result, these performance enhancing techniques also result in a loss of "predictability" — meaning there are large variations in the possible execution time of a program. This loss of predictability is evident when we try to develop timing analysis methods to bound a program's Worst-Case Execution Time (WCET) — the analysis methods become complicated and the WCET estimates are sometimes gross over-approximations of the actual WCET.

Memory access behavior is one of the significant reasons for lack of timing predictability in programs. Most modern processors are endowed with a cache — an on-chip memory which stores recently accessed memory blocks. Typically separate caches are maintained for instruction and data. Unfortunately caches also reduce predictability in a program's execution time — access to the same memory memory block can sometimes be a hit in the cache, and be a miss at other times. For instruction memory accesses, this is less of an issue — due to heavy regularity in instruction memory accesses and also due to the fact that the execution of a particular program instruction I involves exactly one block in the instruction memory (the memory block which contains I). Indeed, analysis techniques to bound instruction cache access timings have been developed (*e.g.*, see [21]). However, ensuring timing predictability of data cache accesses remains a difficult problem.

Ensuring timing predictability of data cache accesses is hard for (at least) two reasons. First of all, many programs have highly irregular data accesses — whose cache access time is affected by the (irregular) order in which they are executed and the data memory layout. Secondly, "address analysis" to find the different data memory addresses accessed by repeated executions of the same instruction is also a difficult problem. For data intensive programs this causes a large variation in the program's execution time.

In this paper we take a fresh look at the issue of cache access predictability. Our proposal does not involve building heavyweight analysis methods to analyze and bound data cache behavior. Nor does it assume any hardware/compiler support to replace caches with compiler-controlled memories. Instead, we propose certain program transformation strategies which can make cache access more predictable. We show that adopting such program transformations significantly lessens variations in program execution time, while providing reasonable program performance. Further, we develop a timing analysis method which can be used to accurately bound the execution time of the transformed program while taking into account its data cache behavior.

Technical Contributions. Technically, our program transformations are inspired by the so-called "cache-efficient" algorithms which have been studied in the algorithms research community [19]. These algorithms systematically track data memory accesses in a program and try to ensure that they will not lead to conflict misses in the data cache. However, adopting such algorithms for efficient *and* time-predictable embedded software construction involves several technical challenges. We tackle these challenges as follows.

- First of all, cache-efficient algorithms are based on a "theoretical execution model", merely concentrating on the cache misses. We execute them on realistic processor models (such as the one supported by Simplescalar platform [3]) to study the efficiency issues – specifically those involving instruction as well as data cache.
- Secondly, developing a cache-efficient algorithm is a highly "creative" activity. Indeed, algorithms researchers have studied at great depth how to make individual algorithms to be cache-efficient [5]. We distill the intuition behind cache-efficient algorithms to develop some *guidelines* to transform an arbitrary data-intensive program into a cache-efficient one.
- We show through detailed experiments that the variation in execution time (say due to differing program inputs) is significantly reduced by our program transformations. This constitutes a conceptual novelty, since cache-efficient algorithms were primarily designed for program efficiency, whereas we advocate a balance of efficiency and predictability via our program transformation.
- For programs with a single path and pre-determined access patterns, there is no execution time variation for a given cache configuration. However, execution time may vary substantially due to change in cache configuration. As shown in our experiments (Sec. 7), our transformed programs show less variation due to change in cache configuration. In other words, we can execute the transformed programs on a processor with smaller cache without substantially sacrificing performance.
- Last but not the least, we develop a static Worst-case Execution Time (WCET) analysis method for data caches. By exploiting the cache-efficient property of the transformed programs we can give tighter WCET estimates than the state-of-the-art data cache analysis methods proposed in literature.

In summary, this paper presents a program transformation approach to produce programs which show less execution time variation across inputs/cache-configurations, while maintaining acceptable performance. Experimental evidence is presented to demonstrate the efficiency, predictability and accurate timing analysis of the transformed programs.

2. RELATED WORK

Timing predictability of embedded systems has been an important topic of research in recent years. Researchers in timing analysis have stressed the importance of bringing a balance between analysis methods and predictable system design (*e.g.*, see [22]). Lee has articulated several roadmaps for integrating timing predictability issues into system design (*e.g.*, see [13]).

Among solutions proposed for constructing predictable embedded software, writing programs with fewer paths has been studied. In particular, [7, 11] advocate the "single-path" approach where the authors recommend developing programs with a single program path. Clearly, writing programs with a single path may not always be possible. Moreover, even for programs with a single path, the cache behavior can be very unpredictable. To see this issue, let us consider a simple example (the arrays `b`, `d` are inputs).

```
for(i = 0; i < 256; i++)
    for(j = 0; j < 16; j++)
       c[d[i]+j] = a[b[i]+j] + 10;
```

The program has only one path. If inputs `b`, `d` are such that `c[d[i]+j]` and `a[b[i]+j]` map to the same cache line there can be lot of cache misses. Furthermore, depending on the data layout, if the starting address of arrays `b` and `d` map to the same cache line, all the `b[i]` and `d[i]` accesses also conflict with each other. This example illustrates that ensuring predictable cache behavior goes beyond developing programs with a single control flow path.

Among solutions proposed for predictable data cache behavior in programs, works on data cache locking deserve mention [23]. In these approaches, certain memory blocks are pre-loaded into the cache for enhancing timing predictability. Works on scratchpad memory allocation [20, 24] propose predictable memory access via compiler managed memories. However, data cache locking approaches requires platform extensions at least in the form of a cache-locking instruction in the instruction set, whereas scratchpad memories require explicit compiler support. Our approach does not involve any changes to the compiler or processor.

Finally, we note that our work differs substantially from works on worst-case execution time analysis of data cache behavior (*e.g.*, see [18]). Instead of developing heavyweight WCET analysis methods, we propose a different approach towards program predictability. Moreover, worst-case data cache behavior is not so amenable to static WCET analysis and leads to gross over-approximations as discussed in the following section.

3. BACKGROUND

In this section, we give the background for our program transformation oriented approach. We first briefly discuss why static timing analysis of data cache behavior is difficult (Sec. 3.1). This motivates our program transformation based approach. We then give an overview of the intuition behind

cache-efficient algorithms — a recent development in algorithms research which inspires our approach (Sec. 3.2).

3.1 Difficulties in Data Cache Analysis

Data cache is an important component for WCET analysis. For data intensive programs, analyzing data cache behavior can give much tighter WCET estimates compared to analysis results without considering data cache (*i.e.*, all misses). However, unpredictability of data accesses in a program makes it difficult to analyze data cache behavior. A significant amount of research has already been done in data cache analysis. Cache miss equations [10] and Presburger arithmetic formulations [4] are two such important techniques. Both of these works can be used only for access patterns which are affine in terms of loop induction variables. Moreover, the analysis time can be exponential in the length of the formulas generated in worst case. Cache miss equation approach has been further extended to handle more general loop nests and data dependent conditionals in [17]. Static cache simulation [25] has also been proposed to categorize data access patterns for analyzing the worst case data cache behavior.

Abstract interpretation is a theoretical framework which inherently guarantees safety and used is for statically analyzing programs. Thus there has been a lot of interest in using abstract interpretation for WCET analysis and more specifically cache analysis (as safety is crucial for WCET estimation). Abstract interpretation for data cache analysis is used in [8] and [18]. [8] describes a persistence analysis for data cache, whereas [18] extends the classical $must$ cache analysis for instruction cache to data cache. Both of these analyses rely on an address computation technique which returns an over-approximation on the set of addresses accessed by a particular load or store instruction in the program. Since the address range is over-approximated for safety, both of the above works suffer from heavy over estimation as it is not clearly known which of the memory blocks are actually accessed in concrete execution of an instruction. Although no experimental results are presented in [8], experiments from [18] show that most of the cases generate a heavy overestimate in WCET computation in presence of data cache. The problem is tackled in [18] by partial unrolling of a loop and compromising the analysis efficiency a bit. The following simple example shows the imprecision that creeps into generic data cache analysis.

```
int c[512];
for(i = 0; i < 1024; i++){
  x = b[i];
  for(j = 0; j < 8; j++)
    c[x+j] = c[x+j] + 10;
}
```

First of all, in the above example both cache miss equation and presburger arithmetic formulation will fail as access patterns of array c is not an affine combination of loop induction variables. This leaves us with using abstract interpretation for data cache analysis as described in [8] and [18].

```
bounded_copy ( int *A, int *B) {
   int C[2 * BLOCK_SIZE];  /*copy buffer*/
    if  (set(A)==set(B))   /*two memory blocks conflict*/
        if  (set(A)==set(C[0])) /*also conflict with first memory block in copy buffer */
          *C = C[BLOCK_SIZE];
        else *C = C[0];
        for (i = 1 to BLOCK_SIZE)    C[i] = B[i];
        for (i = 1 to BLOCK_SIZE)    A[i] = C[i];
   else
        for (i = 1 to BLOCK_SIZE)    A[i] = B[i];
}
```

Figure 1: Bounded copy procedure

For the sake of illustration assume that we are dealing with a 1 KB and 2-way set associative data cache and cache line size 32 bytes. Let each of the array elements be 4 bytes (they are integers). Also assume that elements of array b has value such that all of the memory blocks of array c has been touched in full computation of the loop. The size of array c is $512 * 4 = 2^{11}$ bytes, spanning across at least $2^{11}/32 = 2^6$ memory blocks. A global address computation thus computes (at least) 2^6 memory blocks to be accessed. Since the cache has only 2^5 cache lines, the c array accesses cannot be inferred to be globally persistent and the analysis presented in [8] thus computes a total of $8 \times 2 = 16$ data cache misses for the whole computation of the inner loop.

Extended $must$ analysis presented in [18] will also fail to give a precise result as the number of memory blocks accessed by array c is more than one, although the store to array c is an all-hit situation. However in concrete execution we can see that at most 2 memory blocks of c (in fact, $8 * 4 = 32$ bytes spanning across at most two memory blocks) are accessed in each outer loop iteration leading to only 2 data cache miss per outer loop iteration. Further, the store to c array is always a hit.

Thus, in the above example program, both [8] and [18] will compute a $8\times$ overestimation in data cache miss count which leads to a very imprecise worst-case execution time (WCET) estimate.

3.2 Cache Efficient Algorithms

We now give a general overview of I/O and cache-efficient algorithms. Our proposed program transformation is inspired by these past developments in the algorithms research community.

I/O-efficient algorithms have been studied by the algorithms community to minimize the memory/secondary storage I/O accesses for data-intensive algorithms [1, 16]. Some general techniques have been developed for a program to be I/O-efficient. For example, *scanning* is one of the basic paradigms applied in I/O-efficient algorithms. *Scanning* technique is motivated by the observation that if N data are read in sequential order, they can be accessed in $O(N/B)$ I/O operations; otherwise if the data are randomly accessed, the I/O cost is $\Omega(N)$ in the worst case. Thus, if the original algorithm is modified so that it accesses the input in sequential order, overall I/O cost is reduced. Designing the I/O-

efficient version of an algorithm often requires good understanding of the algorithm's behavior. I/O-efficient versions of many classic algorithms have been proposed, including sorting, matrix tranposition/multiplication, FFT and graph algorithms. Furthermore, given the input size and memory configuration, asymptotic bounds on average/worst case I/O operations can be calculated.

Caches are commonly used in modern processor architectures to bridge the increasing gap between processor speed and memory access latency. Due to the limited cache size and associativity, cache conflict miss is one of the major sources for cache performance degradation and unpredictability. For example, consider the following code fragment.

```
for(i = 0; i < 100; i++) a[i] = b[i];
```

Assuming cache associativity to be 1 and cache block size to be equal to 4 array elements, memory accesses to $a[i]$ result in 25 cache misses (1 miss for every 4 consecutive data accesses) in the best case (where $a[i]$ and $b[i]$ never conflicts in the cache), and 100 cache misses in the worst case (where $a[i]$ and $b[i]$ are always mapped to the same cache set, cache thrashing). Cache efficient algorithms have been proposed to reduce worst-case cache conflict misses [19, 6]. A cache-efficient algorithm can be designed by introducing *cache emulation theorem* to the I/O-efficient one. A special cache-resident data structure Buf is used to hold memory blocks in each stage of the computation to prevent cache conflicts. Buf is allocated as contiguous memory locations having a size smaller than the total cache size. By restricting all memory block references in the computation phase of an I/O efficient algorithm though Buf, the cache emulation theorem guarantees these memory block references do not conflict with each other in the cache. To achieve this, data in the memory blocks referenced in each stage of the computation must be copied to Buf before its use. Similarly, output data in each stage of the computation are also stored in Buf and will be copied back to the output array after completion of the stage.

Note that the worst case behavior of cache thrashing (as discussed in the above example) could also be incurred during copying data between input/output arrays and Buf for direct mapped cache. [19] introduced the concept of *bounded copy* to copy one memory block to another block that guarantees no cache conflicts between the two. Figure 1 presents the bounded copy proposed in [19]. If the two memory blocks with starting addresses of A and B — are mapped to the same cache set, an auxiliary buffer (C in Figure 1) that is mapped to a difference cache set is used. The source memory block is first moved into the auxiliary buffer, and then the memory block is copied from the auxiliary buffer to the destination memory block. In this case, the auxiliary buffer C should contain 2 contiguous memory blocks so that one of them does not conflict with both source and destination. Otherwise, if source and destination memory blocks are mapped to distinct cache set, direct copy can be performed without cache conflicts.

4. PERFORMANCE ISSUES

Cache efficient algorithms are designed to improve performance of data-intensive computations, by minimizing I/O operations (use of underlying I/O efficient algorithms) and cache conflict misses (cache emulation techniques). The asymptotic bounds for worst and average case behavior of the cache efficient algorithms are presented in [19]. For example, it shows that the optimal cache-aware M/B-way mergesort algorithm can sort N numbers in

$$O(N \log N + L \cdot \frac{N}{B} \cdot \frac{\log(1 + N/B)}{\log(1 + M/B)}) \text{ time}$$

where M,B,L are the cache size, block size and miss latency, respectively. However, such asymptotic bounds are computed at the algorithm level. They cannot be used directly as timing properties in the design of real-time applications, where the worst/average case execution time of a program needs to be given as a concrete quantity. Furthermore, they do not consider the real implementation of the algorithm and the underlying micro-architectural features (other than data cache) that may affect the execution time of the cache-efficient programs. In particular, the following important issues are clearly not considered in the asymptotic bounds.

- The real implementation of an algorithm may introduce many additional or auxiliary variables. Memory references to these auxiliary variables might conflict with the buffer variable Buf used in the cache efficient algorithms, and invalidates the buffer's cache-resident property assumed in the cache emulation theorem.
- Depending on the underlying architecture, register spilling may occur when there are more live variables than the number of registers in the processor. For example, in the case that the Buf access index is stored in the memory due to register spilling, and its memory block conflicts with the Buf in the cache, the actual number of cache misses is no longer bounded by the given asymptotic upper bound.
- The cache efficient algorithms implicitly assume that the memory block references are aligned. Sequential access of B data items is considered as referencing within a single memory block, which might not be true (depends on the data layout).
- Many other micro-architectural features of a modern processor affect the memory accesses during program execution, e.g., branch prediction, pipelining, and out-of-order execution. Branch prediction may wrongly speculate and execute certain code, which may have constructive or destructive effect on subsequent cache accesses.

[6] shows that the execution time of I/O efficient algorithms outperforms the original algorithms on general-purpose processor architectures with large input size. However, the

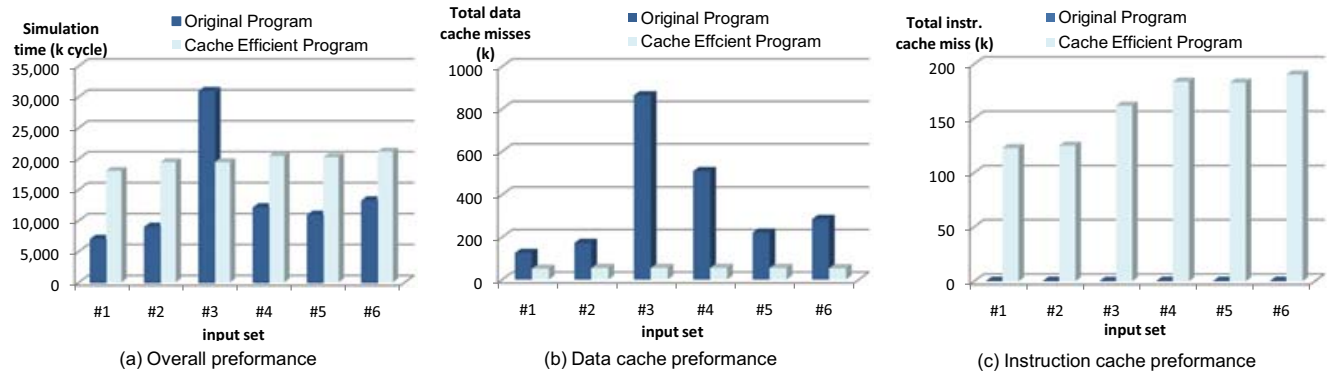


Figure 2: Performance comparison between original mergesort and cache efficient mergesort

same results may not be applied in the context of embedded systems, where available computing resources (instruction/data cache size) are often limited. We implement a cache efficient M/B-way mergesort program by introducing cache emulation and bounded copy into the I/O efficient mergesort algorithm described in [16]. We compare the cache efficient mergesort with original mergesort program in terms of overall execution cycles, number of data cache misses and instruction cache misses. The comparison is done via SimpleScalar ([3]) simulations. The processor configuration used is as follows — direct mapped 2 K-Byte L1 instruction cache, direct mapped 1K-Byte L1 data cache, perfect branch predictor, 5-staged pipeline, in-order execution, 32 Byte memory block size and 30 cycle cache miss penalty. We simulate the original and the cache-efficient mergesort programs with a fixed input size and the same set of randomly generated input data. Figure 2 shows the simulation results.

The overall average performance of cache efficient mergesort is worse than the original mergesort (Figure 2(a)). In the worst case scenario where the input data causes cache thrashing in the original mergesort. Cache efficient mergesort performs better by preventing data cache conflicts in the merge phase. In terms of data cache behavior, cache efficient mergesort has better utilization of the data cache and has fewer total data cache misses (Figure 2(b)). The main reason for the overall average case performance degradation in cache efficient mergesort is the increase in instruction cache misses (Figure 2(c)). The original mergesort program consists of 85 lines of source code, which can be entirely fitted into the instruction cache. On the other hand, the cache efficient M/B-way mergesort consists of 215 lines of source code (due to the use of n-way mergesort algorithm and bounded copy procedure for the cache emulation). The total number of instruction cache misses increases dramatically and compromise the benefits from improved data cache behavior.

The results in Figure 2 allow us to make two nuanced observations about cache-efficient algorithms. First of all, even though cache efficient algorithms have been proposed (by the algorithms research community) for efficiency, the overall execution time of cache-efficient style programs may indeed be worse than their normal counterparts. This is because the asymptotic analysis adopted by algorithms researchers only concentrate on the data cache, while leaving out timing effects of other architectural features such as instruction cache. Secondly, and more importantly, the overall performance of a program P is comparable to a the performance of the the program P' obtained by transforming P to cache-efficient style. In other words, there is no drastic performance degradation in transforming P to P'. This indeed is the key thesis behind our approach — we propose cache efficient style programs since they ensure better timing predictability without drastic performance degradation. We now discuss our program transformation for converting programs to cache efficient style. As we establish via experiments, the transformed programs demonstrate lesser variation in execution time (across inputs) and hence higher timing predictability.

5. PROGRAM TRANSFORMATION

In the design of real-time embedded software, it is often important for the program to be timing-predictable, rather than being very efficient in the average/worst case execution times. Now, what are the characteristics of a program which is timing-predictable? We believe a timing predictable program typically satisfies the following criteria.

- The program should have relatively small variation between the best and worst case execution time to process each set of input data (of fixed size),or between the normalized processing time of each individual input data block for variable input data length (e.g., each frame in the MPEG decoding).
- The program should be analysis-friendly, such that a static analysis can be used to estimate a tight WCET bound. For example, the “single-path" program transformation technique ([7, 11]) reduces the WCET overestimation due to path analysis. In this sense, programs having only one path do have a certain degree of predictability. However, as observed in Section 2 programs having a single path may still have very unpredictable cache behavior. Thus, we state that a timing-predictable program should not only be analysis-friendly

in terms of program path analysis, but also in terms of overall timing analysis (which includes micro-architectural modeling).

5.1 Transformation Guidelines

Although we have shown that the real implementation of cache-efficient programs might not give better overall performance on embedded architectures, the idea of using programmer-controllable cache buffer for large array accesses to reduce data cache conflict misses in the cache efficient algorithms is still valuable to improve embedded software timing predictability. However, developing the the I/O efficient or cache-efficient version of an arbitrary program is in general challenging and timing-consuming. For our purpose of predictability-oriented program transformation, we adopt the idea of cache emulation and bounded copy from cache-efficient algorithms, without explicitly requiring the program to be also I/O-efficient (minimized I/O operations).

The purpose of our program transformation is to make a (data-intensive) program to have the following characteristics — (i) less execution time variation across different input values (less normalized execution time across different input sizes), and (ii) easy-to-analyze via static timing analysis techniques. We propose the following *guidelines* which we use to develop the cache-efficient version of a given program. Thus, these guidelines drive our program transformation. Even though our program transformation is *not* fully automated, these guidelines greatly simplify the effort in developing a cache-efficient program from a given program. It should be noted that in the past researchers have devoted specific effort for converting individual algorithms to cache-efficient versions (*e.g.*, see [5]); instead here we distill the general intuition behind cache-efficient algorithms to develop a common set of guidelines for developing cache efficient programs.

Our *guidelines for transforming an arbitrary data-intensive program to a cache-efficient program* are as follows. First, to facilitate the address computation in static data cache analysis, we use a set of cache buffers $Buf = \{buf_1, \ldots, buf_n\}$ such that $MEM(buf_1) \cap \ldots \cap MEM(buf_n) = \emptyset$ and $size(buf_1) + \ldots + size(buf_n) \leq C$ where $MEM(buf_i)$ and $size(buf_i)$ are the memory locations and size of buf_i respectively and C is the capacity of available data cache size. Thus, the memory range of each cache buffer is easy-to-compute, and no two memory blocks referenced within a single cache buffer will map to the same cache set.

Secondly, data in large arrays (say for input/output arrays) must be visited through the cache buffers to prevent conflict misses. Thus, data must be copied to a particular cache buffer via the bounded copy procedure discussed earlier (refer Fig. 1).

Thirdly, the buffer copying should be outside the loop scope of the computation which accesses the buffers, so that the cache-resident property of the buffer variables is retained. Indeed, this property is exploited in our proposed static timing analysis for cache efficient programs.

```
...
L1 for ( i=0; i<16; i++)
L2    for ( j=0; j<256; j++)
         a[j] = b[c[i]+j];
```

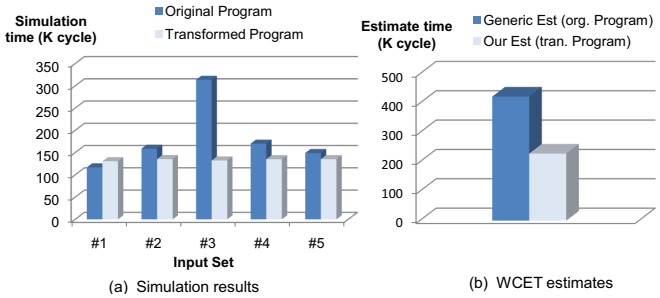
(a) Original program

```
int BufA[BLK_SIZE];
int BufB[BLK_SIZE];
int BufC[BLK_SIZE];

...
L1  for (i=0; i<128; i+=BLK_SIZE)
      bounded_copy( &BufC, &c[i]);
L2    for (j=0; j<BLK_SIZE; j++)
L3      for (k=0; k < 256; k+=BLK_SIZE)
          bounded_copy( &BufB, &b[BufC[j] +k]);
L4        for (h=0; h<BLK_SIZE; h++)
            BufA[h] = BufB[h];
          bounded_copy( &a[k], &BufA);
```

(b) Transformed program

Figure 3: A program and its transformed version.



(a) Simulation results (b) WCET estimates

Figure 4: Timing predictability of transformed program of the example in Figure 3.

Finally, the original bounded copy procedure as shown in Figure 1 has three program paths that have different computation workloads (the outer *else* corresponding to the situation where $(set(A) \neq set(B))$ has less computation required). It causes possible execution time variation while processing different inputs. Furthermore, a static WCET analyzer (without additional user annotation) will conservatively take the path that contributes most computation workload as the longest path for each invocation of the procedure, which leads to overestimation of the program's execution time. We propose to use the following direct copy

```
bounded_copy ( int *b, int *c) {
  for(i=0; i< BLOCK_SIZE; i++)
    b[i] = c[i]
}
```

as the bounded copy procedure whenever the cache associativity $A > 1$. For example, when $A = 2$, both $b[i]$ and $c[i]$ are resident in the cache even they are mapped to the same cache set. In this case, both execution time variation and static analysis overestimation can be reduced.

5.2 An Example

Figure 3 shows the original and transformed version of an example program. In the original program, access pattern of input array $b[]$ is input-dependent (on the value in another input array $c[]$). Even for a fixed memory layout and cache configuration, the data cache conflicts between memory blocks referenced by $a[]$ and $b[]$ varies across different

data values in $c[]$. On the other hand, the transformed program guarantees that the data copy between $BufA[]$ (cache buffer for $a[]$) and $BufB[]$ (cache buffer for $b[]$) in the scope of loop L4 incurs no conflict misses, by following the above-mentioned transformation guidelines. Note that the transformed program may not obey cache-efficient transformation by definition, due to the dissatisfaction of the underlying I/O-efficiency of minimizing I/O operations.

We compare the transformed program's timing predictability with the original program in terms of both execution time variation and tightness of static WCET analysis estimate by SimpleScalar simulation on some randomly selected input sets. The experimental results in Figure 4(a) shows that among the shown input sets, the execution time variation ratio is

$$(obs.\,WCET - obs.\,BCET)/obs.\,BCET$$

where $obs.\,BCET$ is the observed best-case execution time (among the input sets simulated against) and $obs.\,WCET$ is the observed worst-case execution time (again among the input sets for which the program was simulated). Of course for fair comparison, we simulate the original and the cache-efficient program against the same set of inputs. The variation ratio is 168.5% for the original program, compared to only 3.7% for the transformed program. For the static WCET analysis, we adopt the state-of-the-art data cache analysis [18] for general C programs and integrate it into our static worst-case execution time (WCET) analysis tool [14]. We also propose a static timing analysis that exploits the properties in the program transformation; this analysis will be presented in next section). Our timing analysis exploits certain characteristics of cache-efficient programs to deliver tighter estimates. As shown in Figure 4(b), we can achieve 45.9% WCET estimate reduction by performing our new WCET analysis on the transformed program (labeled as "Our Est (tran. Program)"), compared to the generic WCET analysis for data cache [18] on the original program (labeled as "Generic Est (org. Program)"). Thus, by adopting the cache-efficient version of a program we not only enjoy lesser execution time variation, but also lower WCET estimates (which can help the program meet tighter deadlines).

6. TIMING ANALYSIS

Worst-case execution time (WCET) is the *maximum* execution time of a program on a micro-architecture for all possible inputs, which is an essential program characterization in real-time embedded system design. Static analysis based WCET estimation proceeds by finding the longest path in the program's control flow graph, satisfying certain loop bounds. The execution time estimate of each basic block is found by micro-architectural modeling where we develop timing models of the processor micro-architecture (e.g., pipeline, cache, branch prediction) to find the WCET of a sequence of instructions.

Pessimism of static WCET estimation comes from both program path analysis (e.g., infeasible path, inaccurate loop bound), as well as micro-architectural modeling. In data-intensive applications, main overestimation of WCET analysis is due to imprecise data cache modeling/analysis. In this section, we first discuss the well-known persistence data cache analysis, and its potential drawbacks in analyzing the cache-efficient programs. We propose an improved multi-level persistence analysis to achieve fast and accurate data cache analysis, which leading to a tighter WCET estimate.

6.1 Persistence Analysis

Persistence data cache analysis is introduced in [8] to determine the persistence of a memory block. A memory block m is guaranteed to be persistence if no other memory references could evict m out from the cache during program execution. Thus, a persistent memory block incurs cold miss when first reference, and all further references to it result in cache hits.

Given the data cache parameters as follows:

- Capacity C: size of the cache in number of bytes
- Block (line) size B: number of contiguous bytes to be loaded from memory to cache on each cache miss.
- Associativity A: A-way set associative cache means that information stored at some address in memory could be loaded into any of A locations in the cache (depends on the cache replacement policy).
- Cache set $F = \langle f_1, \ldots, f_{C/B/A} \rangle$: A cache set f_i is a group of cache blocks which contains all the A ways that can be addressed with the same index.

Persistence analysis is based on a fixed-point computation of the *abstract cache state (ACS)* $\hat{c}$ over the program's control flow graph (CFG). Abstract cache state consists of *abstract set states* for each cache set. Considering the LRU (Least Recently Used) replacement policy, the abstract set state $\hat{s}_i$ of a cache set f_i captures the upper bound of the positions (the relative ages) of the memory blocks that could possibly reside in the cache set. An *abstract line state* $\hat{s}_i(l_a)$ contains memory blocks that have maximal relative age of a in the abstract set state $\hat{s}_i$, where $1 \leq a \leq A$. For example, $\hat{s}_i(l_2) = \{m_a, m_b\}$ denotes that memory blocks m_a or m_b could reside in cache set f_i with relative age of 2 at certain program execution point. Furthermore, an additional abstract line state $\hat{s}_i(l_\top)$ is introduced to each abstract set state to keep track of memory blocks that have been referenced before but evicted out from the cache by other later memory references.

The analysis traverses the program's CFG and manipulate the ACSs via *update* and *join* functions to determine the persistence of each memory block references. The *update* function takes an input ACS and a set of memory blocks possibly referenced at the current program location, and produces an output ACS which captures (conservative) cache behavior due to the memory references. Intuitively, the *update* function inserts a possibly referenced memory block to abstract

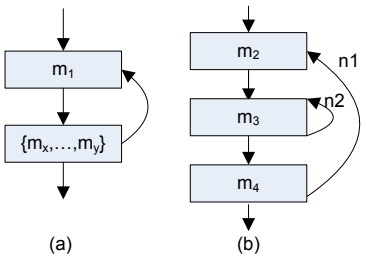


Figure 5: **Overestimation in conventional persistence analysis.**

line state $\hat{s}_i(l_a)$ if the memory block maps to cache set f_i and has a maximal relative age of a. It also update the relative ages of existing memory blocks in the input ACS due to the possibly newly referenced memory blocks, by relocating them to the appropriate abstract line states. If a control flow node has more than one predecessors in the program control flow graph (CFG), the *join* function is applied to compute the ACS of the control flow node by join the output ACSs of all its predecessors. The relative age of a memory block in the joined ACS is set to be the maximum relative age of all its occurrences in the predecessor's ACSs. Readers are referred to [8] for the details of the analysis.

Overestimation in persistence analysis. Two major sources of overestimation in persistence data cache analysis come from (i) array access of multiple possibly memory blocks, and (ii) only global persistence is captured in the analysis. Without precise array index computation, an array access is treated conservatively by using a reference to the set $M = \{m_1, \ldots, m_x\}$ of all memory blocks of the array [8]. If n memory blocks in M mapped to the same cache set f_i, the *update* function simply increases the relative age of other memory blocks in the abstract set state $\hat{s}_i$ by n (or move into $l_\top$ if age is greater than cache associativity). On the other hand, when program contains (multiple levels of) loops, memory blocks may be persistent only within certain loop scope, but could be replaced by other memory references outside the loop. Each time the (inner) loop scope is entered, these memory blocks result in one cold miss, while subsequent accesses within the loop scope will be cache hits. If only the global persistence of memory blocks is computed, these memory blocks are considered non-persistent, and the analysis counts all these references as cache misses.

Two illustrative examples are shown in Figure 5. Assume all shown memory block references, are mapped to the same cache set with associativity $A = 2$. The set $\{m_x, \ldots, m_y\}$ in Figure 5(a) contains possibly referenced memory blocks by an array access. The original persistence analysis overestimates the data cache misses in the following ways.

- It cannot classify memory block $m1$ in Figure 5(a) to be persistent, if the size of the memory block set $\{m_x, \ldots, m_y\}$ is greater than cache associativity. However, it is clear that since exactly *one* memory block in $\{m_x, \ldots, m_y\}$ will be referenced in each iteration of the loop, so $m1$ is persistent.

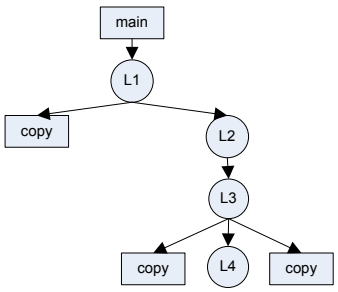


Figure 6: **LPHG of the transformed program in Figure 3.**

- It classifies memory block $m3$ in Figure 5(b) to be non-persistent. However, $m3$ is persistent with in the scope of the inner loop. Thus, assume the loop bound for the outer and inner loops to be $n1$ and $n2$, the total cache miss for $m3$ should be $n1$ (1 miss for each time the inner loop is entered), comparing to $n1 \times n2$ as in the original persistence analysis ($m3$ is not persistent globally).

6.2 Multi-level Persistence Analysis

We propose an improved persistence analysis which gives tighter estimation on the cache-efficient programs, by reducing the above-mentioned two major overestimation sources in the original persistence analysis. The basic idea is to perform multi-level analysis to capture the persistence of memory blocks at different levels of the program execution. Meanwhile, we show that the cache-efficient style of program transformation will naturally reduce the overestimation due to array access of multiple memory blocks in each level of our analysis, which leads to tighter cache behavior estimates. We build an loop-procedure hierarchy graph (LPHG) [15] for our multi-level persistence analysis. Each node in the graph represents the scope of a loop or a procedure call. The graph is context-sensitive so that procedure calls in different program location create separate nodes in the graph. A directed edge in the acyclic graph denotes loop inclusion or procedure invocation, where the root node is the main procedure of the program. Figure 6 shows the LPHG of the transformed program in Figure 3.

Algorithm 1 shows the multi-level persistence analysis, which traverses the LPHG bottom-up and computes the memory block persistence for each scope. We use a modified version of the control flow graph in our analysis, such that for scope sc_i, each node in $CFG[sc_i]$ is either a dummy node representing a child scope of sc_i in the LPHG as a black box, or a normal basic block outsides any of the children scopes of sc_i. The update ($updateACS()$) and join ($joinACS()$) functions are adopted from the original persistence analysis in [8].

Empty initial ACS is assumed for analysis of each scope to capture the persistence within the scope (line 1). The analysis starts from the main procedure by calling $MPA(CFG[main])$, recursively compute the ACS of each child scope (line 3). Once ACS of a child scope sc_j is determined, how its cache behavior affects memory block persistence in the outer scope

Algorithm 1 MPA($CFG[sc_i]$) — Multilevel Persistence Analysis Algorithm. CFG denotes control-flow graph and sc_i is a scope.

1: $\hat{c}_{sc_i} = \perp$ {start persistence analysis with empty ACS for each scope}
2: **for** each child scope sc_j of sc_i **do**
3: MPA($CFG[sc_j]$)
4: $\hat{c}^{exit}_{sc_j} = \{joinACS(\hat{c}^{out}_n) | \forall n \in exitBlock(sc_j)\}$
5: **end for**
6: **while** $\hat{c}_{sc_i}$ not reach fixed-point **do**
7: **for** each node n in $CFG[sc_i]$ **do**
8: $\hat{c}^{in}_n = \{joinACS(\hat{c}^{out}_{n_i}) | \forall n_i \in predecessor(n)\}$
9: **if** n represents a child scope sc_j **then**
10: $\hat{c}^{out}_n = mergeACS(\hat{c}^{in}_n, \hat{c}^{exit}_{sc_j})$
11: **else** {n is a normal basic block outside any children scopes}
12: $\hat{c}^{out}_n = updateACS(\hat{c}^{in}_n, \{m_i | m_i \in reference(n)\})$
13: **end if**
14: **end for**
15: **end while**

is captured by $\hat{c}^{exit}_{sc_j}$, which is the joined union of the *output* ACSs of sc_j's exit blocks to the outer scope(line 4). In order to compute the ACS of current analyzing scope sc_i, for each node in $CFG[sc_i]$, we compute its input ACS (line 8), and update with the memory references to get the output ACS of the node (line 9-13).

If the node is a normal basic block, the original update function in [8] is applied to compute the output ACS after the sequence of memory references in the basic block. Otherwise if the node represents a child scope, we propose a merge function ($mergeACS()$) which compute the output ACS of the node in current scope without re-calculation of the child scope's ACS. The *merge* function describes the effects on the ACS of the current scope by execution the entire child scope, given the input ACS and the child scope's output ACSs at its exit blocks. Merging two ACSs is equivalent to merging the abstract set states for each cache set, i.e.,

$$mergeACS(\hat{c}_1, \hat{c}_2) = \{merge(\hat{s}_{1,i}, \hat{s}_{2,i}) | \forall 1 \leq i \leq C/(B/A)\}$$

where C is the cache size, B is the block size and A is the associativity. Thus, $C/(B/A)$ is the number of cache sets. Given the LRU replacement policy, two abstract set states $\hat{s}_1$ and $\hat{s}_2$ can be merged as follows.

- A memory block m_i that *only* appears in $\hat{s}_2$ will retain its relative age ($\{1, \ldots, A, \top\}$) in the merged abstract set state.
- A memory block m_i that *only* appears in $\hat{s}_1$ will have its relative age increased by by $N_2 = size(\hat{s}_2)$, which is the number of distinct memory blocks referenced in the child scope that are mapped to cache set f_i. If the new relative age of m_i is greater than cache associativity A, m_i is possibly evicted out from the cache during execution of the child scope, thus will be placed into the $l_\top$.
- A memory block m_i that appears both in $\hat{s}_1$ with maximal relative age $a1$ and $\hat{s}_2$ with maximal relative age $a2$ will have its new maximal possible relative age to be $\max\{a1+(n-1), a2\}$. Note that the abstract cache state ACS only captures the memory block's maximal relative age *if* it is referenced, but does not guarantee the memory block will be eventually referenced (e.g., in presence of conditional branches, or array accesses maps to multiple possibly referenced memory blocks). Thus, if m_i is not referenced in the child scope, its new relative age will be $a1 + (N_2 - 1)$ because other $N_2 - 1$ memory blocks may be referenced in the child scope and thus increase age of m_i. Otherwise if m_i is indeed referenced in the child scope, its new age is $a2$. Worst case between the two possibilities is considered to ensure the analysis is conservative.

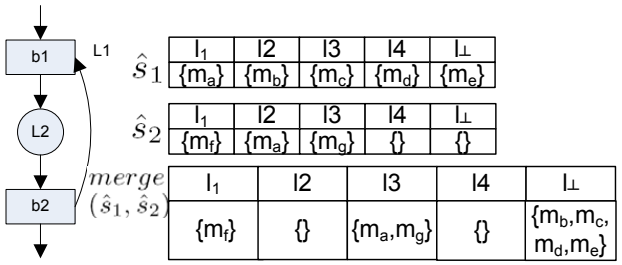


Figure 7: **Illustration of the *merge* operation.**

Thus, we have the *merge* function for two abstract set states as:

$$merge(\hat{s}_1, \hat{s}_2) = \begin{cases} l_h \mapsto \{m_a | m_a \in \hat{s}_2(l_h) \wedge m_a \notin \hat{s}_1\} \cup \\ \{m_b | m_b \notin \hat{s}_2 \wedge m_b \in \hat{s}_1(l_x) \wedge h = x + N_2\} \cup \\ \{m_c | m_c \in \hat{s}_2(l_x) \wedge m_c \in \hat{s}_1(l_y) \wedge \\ \quad h = \max\{y + (N_2 - 1), x\}\} \\ \qquad\qquad \forall 1 \leq h \leq A \\ l_\top \mapsto \{m_a | m_a \in \hat{s}_2(l_\top)\} \cup \\ \{m_b | m_b \notin \hat{s}_2 \wedge m_b \in \hat{s}_1(l_x) \wedge x + N_2 > A\} \cup \\ \{m_c | m_c \in \hat{s}_2(l_x) \wedge m_c \in \hat{s}_1(l_y) \wedge \\ \quad \max\{y + (N_2 - 1), x\} > A\} \\ \qquad\qquad otherwise. \end{cases}$$

where $N2 = size(\hat{s}_2)$ and A is the cache associativity. An example of applying the merge function to two abstract set states is shown in Figure 7 for a particular cache set of a 4-way associativity cache, where $\hat{s}_1$ is the abstract set state before entering an inner scope $L2$, $\hat{s}_2$ is the abstract set state at the exit block of $L2$, which is computed by the persistence analysis for scope L2 with assuming empty ACS. $merge(\hat{s}_1, \hat{s}_2)$ captures the conservative abstract set state after execution of $L2$ during the persistence analysis for the outer scope $L1$. Consider the memory block m_a that has relative ages 1 and 2 in $\hat{s}_1$ and $\hat{s}_2$, respectively. The worst case relative age of m_a in the merged ACS is 3, which corresponds to the scenario where only m_f and m_g are referenced in the scope of $L2$ (m_a may still appear in $\hat{s}_2$ due to conditional branches or set of possibly referenced memory blocks with an array access). Thus, m_a's new age is computed by add 2 ($N_2 - 1$) to its original age in $\hat{s}_1$.

6.3 Suitability for Cache-efficient Programs

Our multi-level persistence analysis is a generic data cache analysis that can be applied to any program with well-structured

scopes (two scopes are either disjoint or one is included in the other). By capturing the memory block persistence at each scope, our analysis results are at least as accurate as the the original persistence analysis [8] (where only global persistence is computed). The fixed-point computation of persistence analysis is required only once for each scope in the LPHG. During the fixed-point computation of an outer scope, the inner scope is treated as a black box, with its cache behavior summarized by the union of all its exit blocks' ACS. Such multi-level fixed-point computation converges faster than the single fixed-point computation used in the original persistence analysis. We note that [2] proposed a multi-level persistence analysis for instruction cache. Their multi-level analysis captures the multi-level persistence by performing n persistence analysis simultaneously for n-level of nested loops, which usually makes the analysis very slow. We avoid this problem by summarizing cache behavior of inner scopes via our *merge* operator.

Although our multi-level persistence analysis can be used to obtain tighter data cache timing estimation for general programs, we gain extra benefits by applying it to our transformed cache-efficient programs. By exploiting our program transformation guidelines, we show that the two major sources of overestimation in persistence analysis can be greatly reduced.

First of all, given our guidelines on usage of cache buffers discussed in section 5, the restriction of large array access via cache buffers during computation steps produces a good chance for the cache buffers to be persistent during a certain scope in the computation. For example, in our running example shown in Figure 3, memory blocks referenced by $BufA[]$ and $BufB[]$ are persistent within the loop scope L4.

Secondly, note that persistence analysis overestimates the cache behavior when an array access references several memory blocks mapped to the same cache set. However, our transformation guidelines prevent such a situation for the buffer variable accesses (such as $BufA[]$, $BufB[]$, $BufC[]$ in Figure 3). The buffers are required to be allocated into contiguous memory regions which are smaller than the total cache size. Thus, we infer that at most one memory block will be loaded to each cache set for (several executions of) a given buffer access.

Finally, sequential loading/storing of data between external input/output arrays and buffer variables also guarantees that at most two memory blocks of the input/output array will be loaded to each cache set within the bounded copy procedure. For example, in the bounded copy procedure shown in Figure 1, $BLOCK\_SIZE$ of contiguous data are referenced by array $A[i]$ each time the function is called. As a result, at most two contiguous memory blocks in array $A[]$ can be accessed (depending on the memory data alignment) within the scope of the function, regardless of how large $A[]$ is.

Thus, cache-efficient programs help reduce the overestimation caused by cache persistence analysis. This makes our analysis particularly suited for this class of programs.

```
L1: for (i := 0; i<N; i++)
L2:   for (k:=0; k<N; k++)
L3:     for (j:=0; j<N; j++)
          c[i][j] +=a[i][k]*b[k][j]
```

(a) Original matrix multiplication

```
L1:for (kk:=0; kk<N; kk+=BLK_SIZE)
L2:   for (jj:=0; jj<N; jj+=BLK_SIZE)
L3:     for (i:=0; i<N; i++)
L4:       for (k:=kk; k<kk+BLK_SIZE; k++)
L5:         for (j := jj; j<jj+BLK_SIZE;j++)
              c[i][j] += a[i][k]*b[k][j]
```

(b) I/O-efficient matrix multiplication

```
int bufC[BLK_SIZE];
int bufB[BLK_SIZE*BLK_SIZE];
int bufA[BLK_SIZE];

L1: for (kk:=0; kk<N; kk+=BLK_SIZE)
L2:   for (jj:=0; jj<N; jj+=BLK_SIZE)
L3:     for (k:=0; k<N; i++)
          bounded_copy(&bufB[k*BLK_SIZE], &b[k][jj]);
L4:     for (i=0; i<N; i++)
          bounded_copy(&bufA, &a[i][kk] );
          bounded_copy(&bufC, &c[i][jj] );
L5:       for (k:= 0; k<BLK_SIZE; k++)
L6:         for (j :=0; j<BLK_SIZE;j++)
              bufC += bufA[k]*bufB[k*BLK_SIZE+j]
          bounded_copy( &(c[i][jj], &bufC);
```

(c) Transformed program (cache-efficient version)

Figure 8: Original, I/O-efficient, and cache-efficient matrix multiplication.

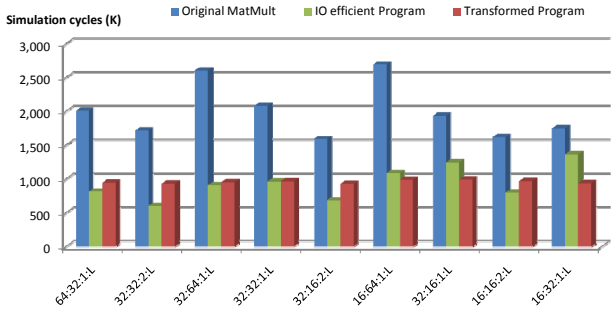


Figure 9: Simulation results for matrix multiplication program.

7. EXPERIMENTS

In this section, we employ our approach on three subject programs — (i) matrix multiplication (ii) Fast Fourier Transform (FFT) computation and (iii) n-way mergesort. The first two are programs with a single control flow path, whereas the last one has many program paths.

7.1 Matrix Multiplication

Figure 8 shows the pseudo-code for the original, I/O-efficient, and cache-efficient version of the matrix multiplication program. The program has a single path with pre-determined access patterns. As a result there is little execution time variability due to variation of input matrices. However, given an input, we can observe variation in execution time due to variation of the cache size. Figure 9 shows the SimpleScalar simulation results over different data cache settings for a fixed input size of 32×32, with the configuration of direct mapped 2 K-Byte L1 instruction cache, perfect branch predictor, 5-staged pipeline, in-order execution, and 30 cycle cache miss penalty. The data cache configuration is shown on the horizontal axis as a 4-tuple $S : B : A : L$, where S is the number of cache sets, B is the block size, A is the associativity, and L represents LRU replacement policy.

Figure 8 shows that I/O-efficient and cache-efficient programs perform better than the original program. Furthermore, by varying the data cache configuration (size, block size, associativity), we observe a variation in execution time (defined as in section 5) of 64% in the original program, 126% in the I/O-efficient program, as opposed to a small

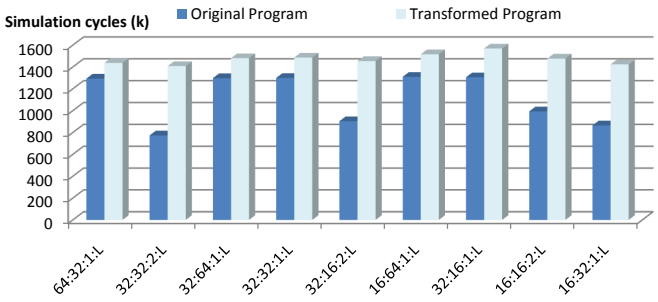


Figure 10: Simulation results for FFT program.

variation of 4.9% in the cache-efficient program. The results show that the I/O-efficient program has large execution time variation even though it minimizes the total number of I/O operations, as it is unaware of the presence of data cache in the processor architecture. Apart from showing lower execution time variation across different cache configurations for the cache-efficient matrix multiplication, the results have another *important* ramification. We can execute our cache-efficient program on a data cache size of 0.5 K-byte (corresponding to the last three data cache configurations in Figure 9), without performance loss compared to using a 2 K-byte data cache (corresponding to the first three configurations). In other words, we can maintain good program performance with *much less* on-chip cache.

As discussed in section 6.3, our multi-level persistence analysis must produce tighter (or at least equal) WCET estimates on any program compared to a generic WCET estimation of data cache behavior via a global persistence analysis. This is because our multi-level analysis maintains more fine-grained persistence information across program scopes. We have employed our multi-level persistence analysis on both the original program and the transformed program. For a data cache configuration of 32 cache sets, block size 32, 2-way associativity data cache (32:32:2:L), our multi-level persistence analysis reports WCET of $4,676,960$ cycles for the original program and $2,406,530$ cycles for the cache-efficient matrix multiplication program. This corresponds to a 48.5% reduction in WCET estimate by adopting the cache-efficient style of programming.

7.2 FFT

Due to the large variation in execution time in I/O efficient programs discussed above, we do not consider them any more in our experiments. We only consider the original program and its cache-efficient version. We transform the original FFT program from MiBench [12] into a cache-efficient style program following our transformation guidelines. The simulation results for original FFT and transformed FFT program are shown in Fig 10, with the previously mentioned processor configurations. Similar to the results shown for matrix multiplication, the execution time variation of the original program over different data cache configurations is 42.6%, compared to 11.1% variation for the transformed program. For the data cache configuration of 32 cache sets, block size 32, 2-way set associative cache (32:32:2:L), the WCET estimate produced by our multi-level persistence analysis for the original FFT program is $2,662,190$ cycles. The WCET estimate obtained via the same analysis for the cache-efficient FFT program is $1,803,340$ cycles — amounting to a 32.3% reduction in WCET estimate by adopting the cache-efficient style of programming.

7.3 Mergesort

Both original and cache-efficient mergesort program contain multiple paths, so that the execution time can vary due to different input data values. In Figure 2, we showed that execution time variation for the cache-efficient mergesort program is *much smaller* than that of the original program (3.7% variation in the cache-efficient program as opposed to 168.5% variation in the original program), even though the average case performance of cache-efficient mergesort is worse partly due to increased number of instructions (and hence increased misses in the instruction cache). As far as WCET analysis goes, our multi-level persistence analysis obtained WCET estimate of $84,915,800$ cycles for the original mergesort program. For the cache-efficient mergesort program, the WCET estimate was $29,285,700$ cycles — amounting to a 65.5% reduction in WCET estimate by adopting the cache-efficient style of programming.

8. CONCLUDING REMARKS

In this paper, we have presented a program transformation approach for developing embedded software with predictable data cache behavior. Our conceptual novelty is to adopt cache-efficient algorithms (which have been proposed by algorithms researchers for achieving fast cache access) to a program development strategy for writing programs which balance predictability with performance. We show that the transformed cache-efficient programs demonstrate less execution time variation due to variation in input data as well as cache configurations. Due to less execution time variation across cache configurations, our cache efficient programs can be executed in comparable times with smaller on-chip cache resources — indeed an important issue in resource-starved embedded platforms. Our experiments indicate that we can also achieve a reduction in Worst-case Execution Time (WCET) estimates by transforming a program to its cache-efficient form.

In future, we will experiment our approach with other benchmarks. Moreover, we plan to further study the interplay between instruction and data cache in determining the execution time of cache-efficient programs. Interestingly, our experiments with cache efficient programs reveal a stable cache behavior across cache configurations all of which share a certain cache block size (mostly the block size parameter is exploited by the cache-efficient program). This throws up an open question — whether we can adapt our transformation strategy to consider *cache-oblivious* style programs [9] which show efficient cache behavior for "tall caches" (caches where the number of cache blocks is larger than the

block size) but without encoding any cache parameters into the program.

Acknowledgments

This work was partially supported by a NUS University Research Council grant R252-000-321-112.

9. REFERENCES

[1] A. Aggarwal and S.V. Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 331(9), 1988.

[2] C. Ballabriga and H. Casse. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *ECRTS*, 2008.

[3] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3), 1997.

[4] S. Chatterjee, E. Parker, P.J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. *SIGPLAN Notices*, 36(5), 2001.

[5] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *HPCA*, 2000.

[6] R.A. Chowdhury. *Algorithms and Data Structures for Cache-efficient Computation Theory and Experimental Evaluation*. Ph.D thesis, University of Texas at Austin, 2007.

[7] J. Fauster, R. Kirner, and P.P. Puschner. Intelligent editor for writing worst-case-execution-time-oriented programs. In *EMSOFT*, 2003.

[8] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES*, 1998.

[9] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

[10] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM TOPLAS*, 21(4), 1999.

[11] J. Gustafsson, B. Lisper, R. Kirner, and P.P. Puschner. Code analysis for temporal predictability. *Real-time Systems*, 32(3), 2006.

[12] 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. In *IEEE 4th Workshop on Workload Characterization*, 2001.

[13] E.A. Lee. Computing needs time. *Communications of the ACM*, 52, 2009.

[14] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3), 2007. `http://www.comp.nus.edu.sg/~rpembed/chronos`.

[15] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC*, 2000.

[16] A. Maheshwari and N. Zeh. A Survey of Techniques for Designing I/O-Efficient Algorithms. *Algorithms for Memory Hierarchies*, page 36, 2003.

[17] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *RTAS*, pages 148–157, 2005.

[18] R. Sen and Y.N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT*, 2007.

[19] S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6), 2002.

[20] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *RTSS*, 2005.

[21] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems*, 18(2/3), 2000.

[22] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-time Systems*, 28(2-3), 2004.

[23] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *SIGMETRICS*, 2003.

[24] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *DATE*, 2005.

[25] R.T. White, C.A. Healy, D.B. Whalley, F. Mueller, and M.G. Harmon. Timing analysis for data caches and set-associative caches. In *RTAS*, 1997.