Investigating Average Versus Worst-case Timing Behavior of Data Caches and Data Scratchpads

Jack Whitham and Neil Audsley
Real-Time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK
jack@cs.york.ac.uk

Abstract

This paper shows that a program using a time-predictable memory system for data storage can achieve a similar worst-case execution time (WCET) to the average-case execution time (ACET) using a conventional heuristic-based memory system including a data cache. This result is useful within any embedded system where time-predictability and performance are both important, particularly hard real-time systems carrying out intensive data processing activities. It is a counter-example to the conventional wisdom that time-predictable means "slow" in comparison to ACET-focused heuristics.

To carry out the investigation, 36 "memory access models" are derived from benchmark programs and assumed to be representative of typical code. The models generate LOAD/STORE instructions to exercise a data cache or scratchpad memory management unit (SMMU). The ACET is determined for the data cache and the WCET is determined for the SMMU. After improvements are applied, results show that the SMMU WCET is within 5% of the data cache ACET for 34 models. In 16 of 36 cases, the SMMU WCET is better than the data cache ACET.

1 Introduction

Time-predictable computer architectures have been proposed for the implementation of *embedded hard real-time systems* [20]. Within such systems, the *timeliness* of tasks is as important as functional correctness [4]: tasks have *deadlines*. Conventionally, the *worst-case execution time* (WCET) of each task is estimated in order to check that deadlines will be met [27]. WCET estimates need to be *safe* ($\geq$ the true WCET) and *tight* (close to the true WCET) [17]. Typical computer architectures aim to minimize the average execution time or energy consumption of programs: time-predictable architectures facilitate the determination of tight and safe WCET estimates [24].

This paper considers the combined execution time of program instructions that access bytes or words of data, known as LOAD and STORE, while ignoring all other instructions and instruction fetch. It shows that, for LOAD-/STORE instructions, a WCET-friendly time-predictable memory system can permit a similar WCET value to the *average-case execution time* (ACET) value obtained using a conventional heuristic-based memory system such as a data cache. It is known that there is often a significant discrepancy between the ACET and WCET using a data cache [24]. Additionally, estimating the WCET of a program using a data cache is a difficult problem because of the cache state uncertainty created by data dependent operation [9, 12]. The *scratchpad memory management unit* (SMMU) used in this paper solves both of these problems together, (1) making WCET estimation straightforward [25] and (2) reducing the WCET to the approximate level of the ACET achieved with an idealized data cache.

Like a cache, the SMMU enables the most frequently accessed *working set* of a program to be stored in fast memory [7], and also like a cache, it implements address transparency: objects stored in external memory retain their logical addresses as they are relocated to *scratchpad memory* (SPM) and vice versa. This allows a fast and time-predictable SPM to store the elements of any data structure, including those that are dynamically allocated, without needing to account for pointer aliasing effects. However, unlike a cache, the WCET of each LOAD/STORE is independent of previous memory accesses. This facilitates WCET analysis by giving every LOAD/STORE a safe and tight execution time bound, dependent only upon their location in the program. The access latency for data in SPM is the same as the hit latency for a data cache.

Earlier publications [24–26] have avoided direct comparisons with data caches due to the known difficulty of estimating the WCET of a program using a data cache [9]. For a single-path program like the ones used for comparison in this paper, the SMMU WCET and ACET are equivalent, but the cache WCET and ACET are very different, as the WCET depends on the memory access pattern. The cache ACET is easily determined by measurement, but it is very difficult to determine the cache WCET, which is typically much greater than the ACET [25]. For this practical reason, the cache ACET is used as a surrogate for comparison. If the SMMU WCET is close to the cache ACET, then this is clear evidence for the benefit of the SMMU, despite the anti-SMMU bias of the WCET/ACET comparison.

The first contribution of this paper is a detailed low level study of the difference between the SMMU's worst-

case behavior and an idealized data cache's average-case behavior. The second contribution is two improvements that bring the SMMU WCET closer to the cache ACET for many programs: the use of *tiling* to split accesses into large objects into smaller sections, and explicit support for read-only objects. Unlike previous work, this paper considers minimal *models* of loop kernels which reproduce memory access patterns while removing details that are not of interest, such as arithmetic computations that have no effect on the memory access pattern. The models are based on samples from SPEC 2000 [11] and Mibench [10] code. Like earlier work on the SMMU, this paper considers only data accesses, on the grounds that time-predictable instruction memory is a relatively well-understood problem [14, 15].

The structure of the paper is as follows. Section 2 summarizes relevant background information, and section 3 explains a method to obtain models of loop kernels from benchmark code. Section 4 describes what happens when these models are applied to two memory subsystems: an SMMU and an idealized data cache. This motivates two improvements to how the SMMU is used, and the implementation of these is described in section 5. Section 6 explores the effects of different hardware configurations and section 7 concludes.

2 Time-predictable Data Memory

A time-predictable data memory subsystem is an essential part of any embedded system supporting hard real-time tasks. 20-30% of the instructions executed by a typical program access data in memory [26]. WCET analysis of the program must take the WCETs of these instructions into account: a time-predictable architecture attempts to ensure that this is possible [1, 20]. Time-predictable architecture research may focus on the instruction memory subsystem [15], the CPU [13], or the data memory subsystem [8, 21]. SMMU research is in the latter category [24].

Obtaining tight but safe WCET estimates for data memory accesses is difficult. The simplest technique of disabling the data cache and accessing external memory is often too slow by a factor of 30 or more [6]. Some researchers have tried to account for the worst-case behavior of conventional data caches so that they can be used safely. Typically this is done by restricting the *reference string* generated by the program: this is the sequence of addresses used by LOAD and STORE operations [7]. If the reference string is predictable, then the state of a data cache is known, and hence the WCET of each LOAD/-STORE may be calculated. The reference string may be restricted by brute force, bypassing the cache for any LOAD or STORE with an unpredictable address [9], such as an access to a scalar variable in global or local memory, or an array access with an statically predictable index. However, this technique doesn't provide any benefit for memory accesses that are data dependent in some way.

Cache-aware memory allocation (CAMA) also attempts to restrict the reference string, but does not insist that every part of it must be predictable [12]. Rather, the WCET analysis problem is seen as an issue of determining where one element in cache may conflict with another. One solution to this problem is to ensure that elements *never* conflict, and CAMA carries out *shape analysis* to determine which data structures may be used simultaneously to ensure that they are allocated to memory locations that will not conflict [19]. This puts a bound on the total number of cache misses during execution without requiring the cache to be bypassed for any LOAD or STORE.

A third option is to bound the number of cache misses within loops, using the relationships between the addresses used in each iteration [18]. This avoids the need for whole-program shape analysis while providing a similar benefit within frequently-executed code.

However, the suitability of conventional data caches for hard real-time systems is not universally accepted. Some researchers advocate the use of novel cache designs targeted at specific problems [20]. Others suggest the use of SPM, often coupled with allocation algorithms that map code [15] or local and global variables [8, 21] to SPM. All these techniques have the advantage that on-chip memory contents are dependent only on the position within the program, *not* the reference string.

SPM techniques have difficulty with dynamic data structures, whether allocated dynamically or not. This is because the possibility of *pointer aliasing* makes it difficult to move objects into SPM as the physical address must change. Aliases of a particular pointer may continue to point to a stale copy of the object. To date, only two approaches have been proposed as solutions: the hardware-based SMMU (section 2.1) and the software-based technique described in [22]. The software-only approach treats the SPM contents as being local to particular regions of the program, but instead of statically fixing the contents in each region, dynamic memory allocation is permitted to use the space [22]. Shape analysis is used to ensure that a particular pointer is only used within regions where it is valid. The technique has two disadvantages: firstly, it is not time-predictable because objects are dynamically *spilled* into slow external memory when space runs out, and secondly, shape analysis must be applied to the entire program, raising tractability issues for large programs.

2.1 SMMU and Scratchpad

The SMMU is a hardware device implementing address transparency between the CPU and memory components. It allows the logical address of an object to remain unchanged as the object is moved between external memory and SPM. This retains the time-predictability of SPM while also solving the pointer aliasing problem [24]. Aliases of a pointer *never* point to a stale copy of the object because its logical address (as used by the program) is never changed. Any SPM allocation algorithm could be adapted to use the SMMU; the new benefit is time-predictable support for dynamic data structures.

The core of the SMMU is a translation table containing

a list of "OPEN" objects. These are memory areas that have been moved to SPM. Each entry in the table is expressed as a lower address bound b_i, a size s_i, a valid bit v_i and an offset t_i. When an object enters the working data set, a program can move it into SPM using the SMMU's OPEN operation, which carries out a fast *direct memory access* (DMA) copy and adds a new entry to the translation table, setting b_i, s_i, v_i and t_i at the same time. While the object is OPEN, *any* access within its address bounds is rerouted to SPM. The physical address p is generated from the logical address l by the translation table function $p = f(l)$, defined as:

$$f(l) = \begin{cases} l + t_i - b_i & \text{if } \exists i, (b_i \leq l < b_i + s_i) \wedge \\ & \quad v_i \ \wedge \ \forall j, (j \leq i \\ & \quad \vee \neg[b_j \leq l < b_j + s_j]) \\ l & \text{otherwise} \end{cases} \quad (1)$$

After a program OPENs an object, single-cycle access to it is absolutely guaranteed: the same access time as a data cache hit. The CLOSE operation reverses OPEN, moving the object back into external memory and clearing the valid bit v_i. It is possible to OPEN more than one object at once. If more than one address range matches a logical address, then a priority ordering implemented by the table is applied to determine which area of SPM contains valid data. As this ordering is also taken into account by OPEN and CLOSE, overlapping objects are permitted, as are arbitrarily interleaved OPEN and CLOSE operations. Extensive tests have verified correct functional and timing behavior in all circumstances [25].

Previous publications have studied the application of the SMMU to a case study [24] and a group of benchmark programs [26]. These studies have revealed some of the properties of the SMMU when used with typical programs, but relevant data is obfuscated by the complexity of the benchmark code. To truly understand the timing properties of cache behavior and SMMU behavior, it is necessary to look at the *micro* level: how the SMMU interacts with small functions and loops.

3 Identifying Memory Access Models

This paper examines the interaction between a *model* of a memory subsystem (e.g. an SMMU) and a *model* of a benchmark program. These models retain all the functional and timing properties that are considered important while abstracting other details. In the case of a benchmark program, the important detail is the memory access pattern: the reference string of the most commonly executed parts of the code. A model reproduces this pattern by issuing LOAD and STORE operations. In the case of a memory subsystem, a model implements LOAD and STORE and any other relevant operations, e.g. OPEN and CLOSE. It replicates the timing behavior of these operations when they are provided with a particular reference string. This approach provides micro-level information about the interactions between the memory and the program.

Instruction `x = A[i];` (LOAD)
base pointer `A`, maximum size 1000,
access pattern: sequential address, with $k = 1$.
Instruction `y = B[x];` (LOAD)
base pointer `B`, maximum size 256,
access pattern: random address.
Instruction `A[0] = y;` (STORE)
base pointer `A`, maximum size 1,
access pattern: constant address.

Figure 1: Memory references within Figure 1.

3.1 Capturing Patterns

A memory access pattern is only of interest if it is frequently executed, i.e. generated by code within a loop. The following process aims to create a model of frequently-executed *loop kernels* within benchmark code. Loop kernels commonly form the basis for other studies of memory and code optimization [2, 28]: in this paper, all operations other than LOAD/STORE are abstracted away. As all control flow is removed and the number of iterations is constant, each kernel is effectively a single-path program [16].

To identify suitable loop kernels, each benchmark program is split into *regions*. These are partitions of the *control flow graph* (CFG) of each program. Each basic block belongs to exactly one region. Region boundaries are introduced at the entrances and exits of every loop and before each basic block where a pointer is created [26].

As each benchmark program is executed in an instruction set simulator [3], counters track the total number of memory accesses initiated by a single LOAD or STORE operation within each region, ignoring instruction fetches and accesses to the local stack. (Each LOAD/STORE instruction may transfer between 1 and 8 bytes.) Once the program completes, or after one billion instructions have been executed, the regions are sorted into descending order of access count. This reveals suitable loop kernels within the program: the regions that account for the highest proportion of memory accesses. The list is truncated to include only those making up the top 95% of memory accesses performed by the benchmark.

The choices of constants ("one billion" and "95%") are arbitrary and have been made in order to ensure that this work is tractable. Otherwise the investigation would be open-ended. More programs could be studied over a longer time period, while considering a larger proportion of loop kernels, but since the 36 loop kernels already exhibit a wide range of different behaviors, it is not unreasonable to say that most common cases are covered here.

The process of capturing a memory access pattern is illustrated by example as follows – the short program shown in Listing 1 contains several regions, but the only one of interest is the loop body, which carries out three memory references for each loop iteration. These access two different objects in memory, `A` and `B`. (References to local variables `x`, `y` and `i` are ignored during analysis; these will be stored either in registers or the local stack.)

Next, a report is produced for each region in the list

Listing 1: Example of a loop kernel within a function ("main").

```
unsigned char A[1000];
unsigned char B[256];
int main(void)
{
    int x, y, i;
    for (i = 0; i < 1000; i++) {
        x = A[i];
        y = B[x];
        if (i==100) A[0] = y;
    }
    return y;
}
```

Listing 2: Model of Listing 1.

```
A = NEW();
B = NEW();
for (i = 0; i < N; i++) {
    LOAD(A + i);
    LOAD(B + RANDOM(256));
    STORE(A);
}
```

(e.g. Figure 1). The report gives information about the reference string $X_r = [r_0, r_1, ..., r_n]$ generated by each LOAD or STORE instruction X within that region. Each X is statically associated with a particular *base pointer* X_b representing a variable used in the source code, e.g. `A` or `B`. Each X also has a maximum size X_s and an access pattern X_p. At this point, conditions such as `if(i==100)` are removed, so that all accesses are assumed to take place on each iteration. The effect of this removal is to make the models pessimistic, tending to overestimate the number of memory accesses. Note that this is exactly how a single-path CPU would operate [16].

X_b, X_s and X_p are identified automatically from the machine code and reference string of the program. The following access patterns are recognized as distinct:

Constant Address. Each execution of X accesses the same location X_b+l, i.e. $X_r = [X_b+l, X_b+l, X_b+l, ...]$. This is typical of a repeated reference to a global variable.

Dynamic Address. There is exactly one execution of X for each time the base pointer is generated, i.e. $X_r = [X_b+l]$. This is typical of iteration through a linked list or traversal through a tree.

Sequential Address. Each execution of X advances the effective address by a constant step X_k, i.e. $X_r = [X_b + l, X_b + l + X_k, X_b + l + 2X_k, X_b + l + 3X_k, ...]$. This is typical of iteration through a buffer or array.

Random Address. Each execution of X is at an unpredictable location within the referenced object, somewhere between X_b and X_b+X_s. This is typical of random access to a dictionary or lookup table.

The three accesses performed by Listing 1 are sequential, random, and constant. The first iterates through `A` with constant step 1. The second accesses an element `B[x]` where `x = A[i]`: this is effectively random, since the contents of `A` are unknown. The third writes to `A[0]` and its condition is ignored.

Listing 3: The main part of the susan_smoothing function is a loop kernel accessing three objects.

```
for(x=-mask_size; x<=mask_size; x++) {
    brightness = *ip++;
    tmp = *dpt++ * *(cp-brightness);
    area += tmp;
    total += tmp * brightness;
}
```

Listing 4: Model of susan_smoothing.

```
cp = NEW();
ip = NEW();
dpt = NEW();
for (i = 0; i < N; i++) {
    LOAD(cp + RANDOM(286));
    LOAD(ip + i);
    LOAD(dpt + i);
}
```

The identified patterns are replicated within the model as follows. For each region, a `for` loop is constructed with a fixed, large number of iterations. Inside the loop, a model "memory access" is placed as a surrogate for each X, expressed as a LOAD or STORE operation for the address to be accessed on iteration i as follows:

Constant Address: LOAD($X_b + l$)

Dynamic Address: LOAD($X_b + l$), then X_b = NEW()

Sequential Address: LOAD($X_b + l + X_k i$)

Random Address: LOAD(X_b+ RANDOM(X_s))

Each base pointer X_b is generated before the `for` loop by the command X_b = NEW(), which can also be used to generate new base pointers within the loop to represent dynamic accesses, such as linked list iterations. NEW() generates a random location anywhere within the model memory space. RANDOM(n) generates a random number between 0 and $n-1$. The model for the example program is shown in Listing 2. It is expected that a model of a memory subsystem defines LOAD and STORE, while NEW and RANDOM are defined by library code.

3.2 Benchmarks

The model-generating process was applied to some of the C language programs in the SPEC 2000 [11] and Mibench [10] benchmark suites. The result is a corpus of model procedures similar to Listing 2. Each model is a single-path program that only accesses memory, and assumes that conditional accesses are always executed. The set of benchmark programs could be expanded to widen the search for possible memory access patterns, but it is important to keep this investigation within reasonable bounds, so the chosen set is assumed to be representative.

Table 1 summarizes the contents of the regions considered, using a code to represent the pattern generated by each model. As an example of this code, Listing 3 shows part of the "susan_smoothing" function, from the susan benchmark [10]. This is model number 32 and contains "1R[286] 1S+1 1S+1". This means there are three dis-

Nr	Program	Function	Contents
1	bitcount	ntbl_bitc...	8R[16]
2	gsm	ulaw_input	4C* 1C 1C
3	djpeg	jpeg_idct...	16R[134] 9R[160]
4	rijndael	encrypt	224R[4.0k]
5	bitcount	ntbl_bitcnt	1R[16]
6	djpeg	jpeg_idct...	16C* 9R[1.4k]
7	bitcount	AR_btbl_b...	4R[256]
8	gsm	Calculati...	40R[4.0k]
9	CRC32	crc32file	1C 4C* 1R[1.5k]
10	art	train_match	5C 1C 1C 1C 1C 1C 1R[1.0M] 1R[1.0M] 1S+8
11	art	train_match	8C 1C 5C 1C 1C 1R[1.0M] 5S+64*
12	bzip2	fullGtU	13C 6D 6D
13	bzip2	qSort3	2C 1C 1C 1R[683.6k] 1S+1
14	ammp	a_number	2C* 3D
15	ammp	a_m_serial	5C* 3D
16	patricia	main	10D
17	art	train_match	5C 5C 5S+64*
18	ispell	ichartostr	2R[20.5k]*
19	bzip2	generateM...	8C 1C 1C 1C 1R[683.6k] 1R[263] 2R[1.0k]* 1S+1 1S+1*
20	art	train_match	11C 7C 2C 1C 1C 7S+64*
21	art	train_match	5C 1C 1C 2C 1C 1R[88] 1S+8 1S+64 2S+1*
22	bzip2	getRLEpair	13C* 1S-1
23	bzip2	spec_getc	12C* 1S+1
24	mesa	clear	1C 1S+4*
25	ispell	linit	2C 1C 6S+24*
26	bitcount	main	1S+8
27	fft	fft_float	4S+8* 2S+8 3S+8* 3S+8*
28	adpcm	adpcm_coder	2R[404] 1S+1* 1S+2
29	djpeg	ycc_rgb_c...	2R[1020] 1R[1020] 1R[1020] 3R[1.4k] 3S+1 3S+3*
30	bzip2	fullGtU	16C 16C 4S+1 4S+1 4S+1 4S+1
31	gsm	Short_ter...	2S+2*
32	susan	susan_smo...	1R[286] 1S+1 1S+1
33	ispell	good	1R[88] 1S+1 1S+1
34	djpeg	h2v2_fanc...	2S+1 2S+2*
35	gap	ProdInt	4S+2 8S+2*
36	gsm	Weighting...	9S+2

Table 1: Summary of the memory access models generated from the benchmark programs. Each loop kernel is represented by a single table row containing one identifier for each base pointer X_b used within it. The identifiers are all of the form nTp, where n is the number of accesses to that base pointer, T is the access pattern X_p (**C**onstant, **S**equential, **D**ynamic, or **R**andom address), and p is the property of that access, if any. For a sequential access type, p is the step from one access to another (i.e. X_k) and for a random access type, p is the maximum object size (i.e. X_s). Identifiers are followed by * when STOREs are performed.

tinct base pointers, which are accessed (1) randomly, (2) sequentially, with step 1, and (3) sequentially, again with step 1. The base pointer that is accessed randomly is `cp`, and the size of the accessible memory space is 286 bytes. `ip` and `dpt` are both accessed sequentially. The model for this region is shown in Listing 4. All calculations are lost, leaving a model of the memory access pattern.

During examination of the benchmarks, it was found that some loop kernels are actually within library code, particularly formatted output functions such as `printf`. These are omitted (1) because producing output is not the main focus of each benchmark, and (2) different implementations of `printf` are possible, and some may be better suited to embedded hard real-time systems than others. Including them would bias the results in a way that depends entirely on the `printf` implementation, which is not relevant to the investigation.

Other commonly used library functions are related to copying and zeroing memory, e.g. `memcpy` and `memset`. These are also omitted from consideration because a cache or SMMU-based system should use customized versions of `memset` and `memcpy` to match the hardware.

4 Memory Subsystems

In Table 1, many memory access models use more than one base pointer. Unless those base pointers are specifically chosen to avoid cache conflicts [12], it will be very difficult to determine the WCET of the model when used with a data cache. This is particularly true for the models that use random or dynamic addresses, since there is no consistent access pattern between iterations. Intuitively, one would expect a large discrepancy between the true WCET for a cache and the ACET for a cache [25].

The SMMU exhibits worst-case behavior unless objects overlap in memory, which permits a minor improvement in ACET unless each LOAD/STORE operations is required to have a fixed execution time. Intuitively, one would expect the SMMU's WCET to be larger (i.e. *worse*) than the ACET obtained using a cache, because not all objects can be OPENed by the SMMU [26].

4.1 Replacing Intuition with Data

Using the memory access pattern models, the above intuitions can be replaced with quantitative data [23]. Figure 2 shows a comparison of the WCET of each model obtained using an SMMU, and the measured ACET of each model obtained using a data cache. This includes only memory operations, not control flow or calculations. The memory subsystems tested are (1) a 16kbyte fully-associative write-back data cache with 64 byte cache lines and LRU replacement policy, and (2) a 16kbyte SPM combined with a 16 entry SMMU. Objects are packed into SPM space in descending order of access count.

The data cache is idealized, with no time penalty for fully-associative cache lookup. If a more realistic set-associative data cache were used, then cache results would

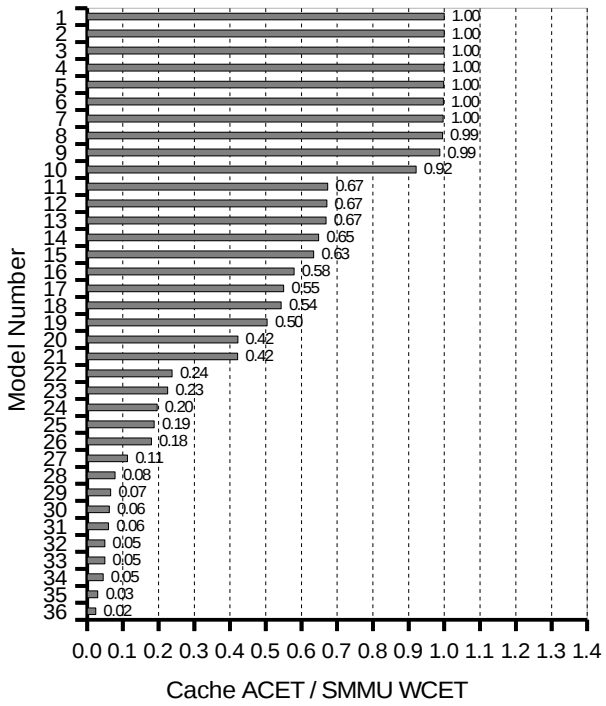

Figure 2: Bar graph illustrating cache ACET divided by SMMU WCET for each model. Values less than 1.0 indicate that the SMMU's WCET is *greater* than the cache ACET, and thus the SMMU's worst-case performance is poorer than the cache case.

be poorer, as conflicts could occur between different cache sets. Furthermore, the single-path assumption also improves the cache results, as there is no need to consider the cache state uncertainty introduced by multiple execution paths. In this experiment, uncertainty only comes from the address sequence. To deal with that, the cache ACET is gathered over 1000 model executions. These problems do not apply to the SMMU, for which it is possible to capture the WCET (and ACET) of a model by one measurement.

The size of the loop bound for each model is 20,000 iterations; this figure is chosen to ensure that sequentially-accessed objects cannot fit entirely within cache or SPM, as this would distort the results in favor of models using small constant steps. Both subsystems begin in an empty state, and are flushed after each model has completed: this time counts towards the WCET.

Some of the models make better use of data cache than others, as Figure 3 illustrates. Where the ACET of a LOAD/STORE is near to 1.0, practically all accesses are cache hits. However, this only occurs in a few cases. The cache is not always the best solution, even for ACET.

During the experiments, it is assumed that only memory access operations affect the ACET and WCET. On-chip memory accesses cost 1 time unit per 4-byte word. External memory access times for s bytes are defined by:

$$T(s) = 50\lceil \frac{s}{64} \rceil + \lfloor \frac{s \bmod 64}{4} \rfloor \qquad (2)$$

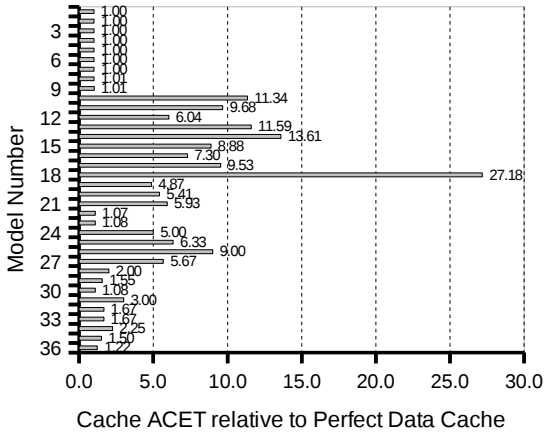

Figure 3: Mean execution time of a LOAD or STORE operation within each model, in relation to the execution time with a perfect data cache: a memory that always responds to LOAD/STORE in one time unit. The scale reflects the access time to external memory as defined by equation 2.

This equation is based on the cost of accessing memory on typical embedded systems [25], where one time unit is one CPU clock cycle. Bus transactions to external memory have a setup cost of 50 time units and a maximum size of 64 bytes. Up to four bytes are transferred during each time unit. Section 6 examines the effect of adjusting the constants in this equation (50, 64 and 4).

4.2 Confirming Intuitions

The data presented in Figure 2 shows that the SMMU comes very close to the ACET obtained with the cache in *some* cases, particularly models 1-9. These models use only constant-address and random-address accesses with small objects. They map very efficiently to both cache and SMMU because the entire working data set used by the model fits in on-chip memory.

The remainder of the models do not map so well to the SMMU. This is because the working data set is much larger than the on-chip memory. In most cases the cache handles this well because its contents change during execution of the model. In 22 models, the problem is caused by one or more sequential accesses. The very worst examples, e.g. models 26-36, use sequential accesses almost exclusively. The SMMU handles these very badly, and where performance appears to improve (e.g. between 26 and 36), it is mainly because of differences in the cache ACET. Due to the experimental design, the sequentially objects are *always* too large to be loaded on-chip, so accesses are redirected to external memory. (In reality, sequentially accessed objects sometimes do fit entirely on-chip, so this is further bias against the SMMU.)

However, for a sequential access, there is no need to load the entire object into SPM. Instead, a small "window" can be OPENed, loading successive small sections (*tiles*) as the sequence progresses. This is possible and time-predictable because the reference string of accesses

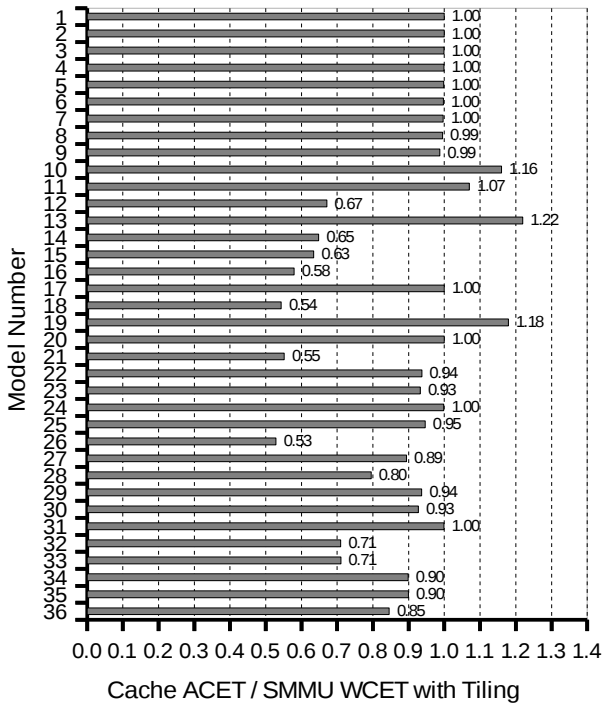

Figure 4: Bar graph illustrating cache ACET divided by SMMU WCET for each model when tiling is used. The SMMU WCET of 22 models has decreased by 5% or more versus Figure 2.

to that object is known to be a linear sequence.

Figure 4 shows the effect of loading each sequentially-accessed object in tile form. The memory subsystems are the same as in Figure 2, but each sequentially-accessed object is now loaded in 1kb tiles during iteration, ensuring that SPM space is always available. The effect is dramatic. 17 models now have an SMMU WCET within 5% of the cache ACET, and the WCET of 22 models has decreased by 5% or more versus Figure 2. Four models now have an SMMU WCET that is *less than* the cache ACET, appearing as data points > 1.0 in Figure 4. This happens because dynamic accesses and random accesses to large objects do not interact well with a cache. In general, cache line loads are beneficial only if the line contents are going to be reused. As the contents are not normally reused (there is no locality), it is much better to access external memory directly in these cases, and this is exactly what happens with the SPM and SMMU, which load the minimum amount of data required and no more. The cache assumes that there is locality, and therefore it is always worth loading an entire line of data. While this assumption is often effective, it increases the cache ACET relative to the SMMU WCET in some cases.

4.3 Closing the SMMU/Cache Gap

Even with tiling, the SMMU WCET is still often greater than the cache ACET. The issue is not related to large objects, as it affects models which use only dynamic data structures, e.g. model 16. These are processed more efficiently by the cache, which appears counter-intuitive because (1) the data structure elements are placed in effectively random memory locations, and (2) the sizes of the elements do not need to be rounded up to the nearest cache line size when the SMMU is used.

The problem here is the SMMU's CLOSE operation, which always writes back to external memory even if the data in SPM has not been updated. A "write-back" cache like the one simulated for Figures 2 and 4 skips this step unless it is absolutely necessary because a STORE operation has written to the cached data. If OPEN and CLOSE are redesigned to allow objects to be OPENed in a read-only mode without a write-back step during CLOSE, then the gap between cache performance and SMMU performance is narrowed. This is illustrated by Figure 5, where the SMMU's translation table is modified to support up to three read-only entries, which can be explicitly selected using a read-only flag given to the OPEN operation (which becomes OPEN_RO). When CLOSEd, the contents of these objects are discarded, not written back. The reason for limiting the number of read-only entries to three is explained in section 5.2.

However, Figure 5 shows yet another dramatic improvement in the Cache ACET / SMMU WCET ratio. The WCETs obtained using the SMMU are now within 5% of the cache ACET in 34 cases, and the WCET of 18 models has improved by more than 5% relative to Figure 4. Clearly, OPEN_RO provides a worthwhile improvement.

4.4 Random Accesses

Figures 4 and 5 illustrate that the remaining difficulty is random accesses to large objects. Random accesses to small objects are not a problem, because they can be handled by loading the entire object into SPM. However, models 18, 21, 28, 29, 32 and 33 all access objects that are too large to fit in SPM, and while the SPM can be made larger, its size is limited by hardware considerations such as clock frequency, silicon area and energy consumption. As long as the external memory has a higher capacity than on-chip memory, it will always be possible to access a working data set that is simply too large to fit on-chip. An SMMU and SPM are no help at all with objects of this size, and a cache is also unhelpful if the WCET is of interest, as the entire working data set cannot fit in cache and each access *could* use an address that is not already loaded. This is reflected in Figure 3: these models have poor performance with both cache and SMMU.

Model 18 is an unusual case. Here, the cache is significantly better than the SMMU. The only object used by this model is only slightly larger than the cache size: this is why the cache is relatively successful in comparison to the SMMU. The cache will *probably* contain a requested element in the average case. However, in order to be time-predictable, the SMMU must contain either *all of* or *none of* a randomly accessed object.

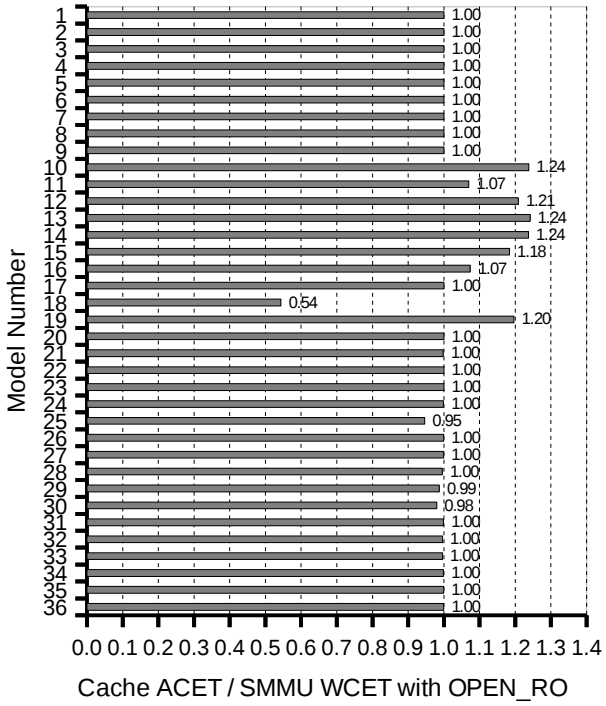

Figure 5: Bar graph illustrating cache ACET divided by SMMU WCET for each model when OPEN_RO is used for read-only objects. The WCET of 18 models has improved by more than 5% versus Figure 4.

5 Improving the SMMU

Results presented in section 4 indicate that most of the discrepancy between the SMMU and cache can be mitigated by (1) tiling and (2) using OPEN_RO whenever possible. Together, these improvements actually make the SMMU *better* than the cache for a significant number of models. The exceptions are models involving random accesses to large objects, which are not executed efficiently in any case. The challenge is to implement the features without losing time-predictability.

5.1 Implementing Tiling

Loop tiling is a form of *loop transformation*. It aims to improve code performance by increasing data locality [28]. A number of transformations, including loop scaling, skewing, interchange and reversal may be applied as part of the process within modern compilers such as GCC [2]. These transformations are a benefit to data caches because they ensure that accesses to memory are sequential where possible.

A new set of loop transformations are not required to make good use of the SMMU. As Figure 4 illustrates, code that works well with a cache works well with the SMMU. All that is needed is a mechanism to split sequential memory accesses performed by a loop into tiles that will fit within the SPM. It is only necessary to trigger a move to the next tile when the end of the current tile is reached.

Listing 5: Implementation of tiling for susan_smoothing's accesses to the "ip" object.

```
tile_size = 1024;
tile_step = 1;
ip_handle = OPEN(ip, tile_size, 0);
for(x=-mask_size, i=0;
        x<=mask_size; x++, i+=tile_step) {
    if (i >= tile_size) {
        CLOSE(ip_handle);
        ip_handle = OPEN(ip, tile_size, 0);
        i = 0;
    }
    brightness = *ip++;
    tmp = *dpt++ * *(cp-brightness);
    area += tmp;
    total += tmp * brightness;
}
CLOSE(ip_handle);
```

This can be done using nested "for" loops or a rarely-executed conditional statement within the inner loop, with a number of executions bounded as a function of the surrounding loop bound.

Listing 5 shows a trivial implementation for one of the sequential accesses carried out within `susan_smoothing` (Listing 3). The relevant code could be automatically inserted by a compiler upon identification of a sequential access, after loop transformations have been applied. Implementation of this compiler feature is future work.

5.2 Implementing Read-only OPEN

OPEN_RO, a read-only OPEN operation, is required to match cache performance in many cases (Figure 5). Like any other SMMU operation, this must be time-predictable. Implementing OPEN_RO is subtly non-trivial. To illustrate the difficulty, suppose that (1) an object Z is OPENed using OPEN_RO, but (2) a pointer alias $a = Z$ references it and (3) alias a is used to update Z. As in the general form of the pointer aliasing problem, the compiler does not know the relationship between a and Z and cannot guarantee either $a = Z$ or $a \neq Z$ [5]. It is important that situations similar to this are handled so that program semantics are preserved and time-predictability is retained.

The simplest correct design for an SMMU augments the existing design [24,25] with n_r translation table entries defined as read-only. These must be the lowest priority entries in the table according to equation 1. It is possible to assign entries as read-only or read-write at run-time, but this is significantly more complex as the priority encoder must give all read-write entries at higher priority.

If a STORE operation matches with one or more read-only entries, *and no read-write entries at all*, then the data is written directly to external memory *and* every matching read-only entry listed in the table. This ensures that no read-only entry contains stale data. When a read-only entry is CLOSEd, there is no write to external memory, and therefore no subsequent opportunity to bring the read-only entries up to date. If there are n_r read-only entries, then n_r SPM writes will be required in the worst case for each STORE. Provided that n_r is small, the cost of these writes

is entirely hidden by the cost of the external memory update, since $T(1) > n_r$.

However, CLOSE operations must also write to SPM, in cases where overlapping memory areas are OPEN and the object being CLOSEd is read-write. Consider a configuration where a memory area of size s bytes has been OPENed $n_r + 1$ times: n_r times in read-only mode, and once in read-write mode. The area is then updated by a STORE; this is serviced by a single write to SPM within the read-write copy of the object. Then the read-write table entry is CLOSEd. In order to preserve program semantics, every read-only entry must be updated for every byte in the read-write entry, because no further CLOSE operation will carry out a write back operation. This means sn_r writes to SPM in addition to s writes to external memory.

The time cost of the external memory writes is $T(s)$ (equation 2). The time cost of sn_r SPM writes is $\lceil \frac{sn_r}{4} \rceil$ if one 4-byte word can be written on each clock cycle. It is possible that the external memory access time will be *less* than the internal update time for the SPM, because contiguous writes to external memory are handled as burst transactions: 1 high latency write may be faster than n_r low-latency writes. The cost of the external memory update only subsumes the cost of the SPM update when inequality 3 is satisfied:

$$T(s) \leq \lceil \frac{sn_r}{4} \rceil \tag{3}$$

Using the definition of $T(s)$ from equation 2, inequality 3 is not satisfied for all $s > 0$ unless $n_r \leq 3$. Therefore the maximum number of read-only objects that can be *reliably* updated during a read-write CLOSE operation is 3. This is why 3 was chosen as the limit for the experiment used to generate Figure 5.

Alternative ways of solving this problem do exist. $T(s)$ could be increased, e.g. by reducing the maximum burst size, but this would also increase every WCET. Or the time cost of the read-write CLOSE operation, currently $T(s)$, could be redefined as $max(T(s), \lceil \frac{sn_r}{4} \rceil)$. Finally, whole-program shape analysis could be used to detect the places where read-only objects may overlap read-write objects or be updated by aliased pointers. But these alternatives are quite poor. Increasing the WCET for all operations or introducing shape analysis breaks the design goals of the SMMU [24]. Redefining the cost of CLOSE operations *seems* acceptable because read-only objects appear to be much more common than read-write objects (90 versus 24 within the models considered) but in fact the additional CLOSE cost becomes extremely significant as n_r is increased beyond 3. The SMMU WCET of nearly half of the models increases substantially.

Therefore, restricting n_r to a value satisfying inequality 3 for all $s > 0$ appears to be the best possibility, particularly as SMMU allocation algorithms can use OPEN in place of OPEN_RO without changing program semantics. The change described above has been implemented in hardware and validated by the same test process used to check the original SMMU design [25]. The result is slightly increased hardware usage.

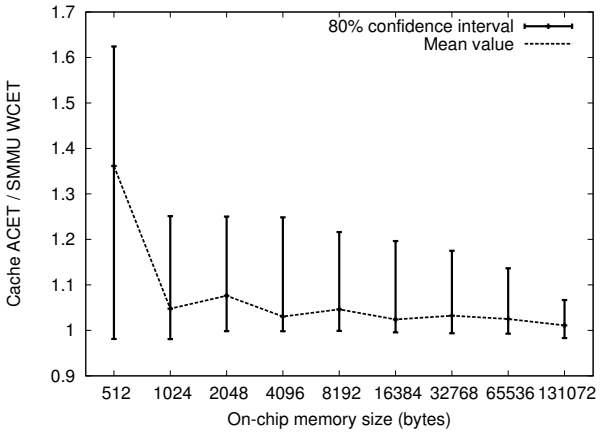

Figure 6: Effect of changing the amount of on-chip memory on Cache ACET / SMMU WCET. The chart shows the mean value and 80% confidence interval for all models (calculated using the 10th and 90th percentiles).

6 Exploring the Parameter Space

Various system-specific parameters are chosen in section 4. These include the size of the on-chip memory, selected as 16 kbytes (cache or SPM), and the transaction setup time of 50 time units (equation 2). Although both of these choices are based on real embedded systems [25], it is worth considering the dependence of the results presented (e.g. Figure 5) on these particular values.

The on-chip memory size affects the results as shown in Figure 6. This graph is drawn using the assumption that (1) all items in Table 1 are equally important members of a population of all significant loop kernels, and (2) that all other parameters are as specified in section 4. The Cache ACET / SMMU WCET value were determined for each model as for Figure 5, and used to compute the mean value and 80% confidence interval.

As the results show, the SMMU is typically better than cache with small memory sizes because it can store small objects efficiently, without consuming an entire cache line. It retains some advantage over cache as the amount of space is increased. As the on-chip memory size becomes very large, the cache and SMMU results converge towards the same value.

The bus setup latency has very little effect on the results. Keeping all other parameters as specified in section 4, Cache ACET / SMMU WCET was evaluated for each model with bus latencies of 25, 50 and 100. The mean, median and standard deviation of the results for each bus setup latency turned out to be within 4% of each other.

7 Conclusion

This paper has described a low-level study of the SMMU based on single-path models of memory access

patterns extracted from benchmark programs. It has compared the SMMU to a fully-associative data cache of the same size by executing the models using a cache and SMMU, determining ACET and WCET. The SMMU WCET and ACET are equal as the models are single-path.

It has shown that the SMMU's WCET can be very close to a cache's ACET in almost all of the cases examined. Figure 5 shows an SMMU/cache comparison with an on-chip memory size of 16kb in which the SMMU's WCET is lower than the cache ACET in 16 cases, and in 34 cases, the SMMU's WCET is less than or equal to the cache ACET plus 5%. Figure 6 indicates that this result is typical of other on-chip memory sizes. Poor results are only obtained for models that make poor use of a cache, e.g. 18.

Unlike a data cache, an SMMU is fully time-predictable, giving a tight execution time bound for each LOAD or STORE, even those using data-dependent addresses and accessing dynamic data structures. The results show that time-predictable systems do not need to be significantly slower - *on average and in the worst case* - than those using heuristic mechanisms such as caches.

Future work will involve adding SMMU loop tiling support to a compiler as per section 5.1 and exploring alternative SMMU designs that improve WCET and ACET further.

Acknowledgments

This work was supported by the EU ICT project eMuCo, no. 216378, and the EPSRC project TEMPO, no. EP/G055548/1. The authors would like to thank the ECRTS reviewers, Rob Davis and Alan Burns for their comments and advice on this paper.

References

[1] S. Bandyopadhyay, F. Huining, H. Patel, and E. Lee. A scratchpad memory allocation scheme for dataflow models. Technical Report UCB/EECS-2008-104, EECS Department, UCB, Aug 2008.

[2] D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for gcc. In *Proc. GCC Developers Summit*, pages 37–54, 2004.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[4] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[5] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. PLDI*, pages 296–310, 1990.

[6] Christian Ferdinand and Kai Richter. Tutorial on Timing Analysis and Optimization. `http://www.embedded.dk/download/Tutorial_EW08_AbsInt.pdf`.

[7] P. J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, 1970.

[8] J.-F. Deverge and I. Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *Proc. ECRTS*, pages 179–190, 2007.

[9] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *Proc. LCTES*, pages 16–30, 1998.

[10] 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 *Proc. IISWC*, 2001.

[11] J. Henning. SPEC CPU2000: measuring CPU performance in the New Millennium. *Computer*, 33(7):28–35, Jul 2000.

[12] J. Herter, J. Reineke, and R. Wilhelm. CAMA: Cache-Aware Memory Allocation for WCET Analysis. In *Proc. ECRTS (WIP)*, pages 24–27, 2008.

[13] S. Mohan and F. Mueller. Merging state and preserving timing anomalies in pipelines of high-end processors. In *Proc. RTSS*, pages 467–477, 2008.

[14] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2-3):217–247, 2000.

[15] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. Technical Report PI 1818, IRISA, 2007.

[16] P. Puschner. Experiments with WCET-oriented programming and the single-path architecture. In *Proc. WORDS*, Feb. 2005.

[17] P. Puschner and A. Burns. Guest editorial: A review of wcet analysis. *Real-Time Syst.*, 18(2-3):115–128, 2000.

[18] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proc. RTAS*, pages 71–80, 2006.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[20] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[21] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. RTSS*, pages 223–232, 2005.

[22] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys.*, 5(2):472–511.

[23] J. Whitham. Source code and raw data for experiments. `http://www.jwhitham.org.uk/c/smmu.html`.

[24] J. Whitham and N. Audsley. Implementing Time-Predictable Load and Store Operations. In *Proc. EMSOFT*, pages 265–274, 2009.

[25] J. Whitham and N. Audsley. The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing, and Case Study. Technical Report YCS-2009-439, University of York, 2009.

[26] J. Whitham and N. Audsley. Studying the Applicability of the Scratchpad Memory Management Unit. In *Proc. RTAS*, pages 205–214, 2010.

[27] 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. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[28] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. PLDI*, pages 30–44, 1991.