Implementing Time-Predictable Load and Store Operations [1]

Jack Whitham
jack@cs.york.ac.uk
Real-Time Systems Group
Department of Computer Science
University of York, York

Neil Audsley
neil@cs.york.ac.uk
Real-Time Systems Group
Department of Computer Science
University of York, York

ABSTRACT

Scratchpads have been widely proposed as an alternative to caches for embedded systems. Advantages of scratchpads include reduced energy consumption in comparison to a cache and access latencies that are independent of the preceding memory access pattern. The latter property makes memory accesses time-predictable, which is useful for hard real-time tasks as the worst-case execution time (WCET) must be safely estimated in order to check that the system will meet timing requirements.

However, data must be explicitly moved between scratchpad and external memory as a task executes in order to make best use of the limited scratchpad space. When dynamic data is moved, issues such as pointer aliasing and pointer invalidation become problematic. Previous work has proposed solutions that are not suitable for hard real-time tasks because memory accesses are not time-predictable.

This paper proposes the scratchpad memory management unit (SMMU) as an enhancement to scratchpad technology. The SMMU implements an alternative solution to the pointer aliasing and pointer invalidation problems which (1) does not require whole-program pointer analysis and (2) makes every memory access operation time-predictable. This allows WCET analysis to be applied to hard-real time tasks which use a scratchpad and dynamic data, but results are also applicable in the wider context of minimizing energy consumption or average execution time. Experiments using C software show that the combination of an SMMU and scratchpad compares favorably with the best and worst case performance of a conventional data cache.

Categories and Subject Descriptors

C.3 [**Special Purpose and Application-based Systems**]: Real-time and Embedded Systems

General Terms

Design, Experimentation

EMSOFT'09, October 12–16, 2009, Grenoble, France.

1. INTRODUCTION

A *time-predictable memory operation* is defined as a *load* or *store* instruction within a task with a *latency* that can be precisely predicted before execution [28]. Time-predictable memory operations are particularly important in embedded systems that execute *hard real-time* [3] tasks, because each task must complete before a deadline. The *worst-case execution time* (WCET) is estimated for hard real-time tasks in order to assure this [25]. The process of WCET analysis (estimating the WCET) is greatly simplified if each memory access operation is time-predictable. It is easier to produce an estimate that is *tight* (close to the true WCET) while still being *safe* (no less than the true WCET) as the WCET estimation process does not need to account for the variable latency of memory access operations.

This paper *only* considers data memory access operations, i.e. load and store instructions. It is assumed that all other instructions have predictable latency, which could be arranged in a practical embedded system through the use of a simple CPU design [39] with a scratchpad to store instructions [23]. The latency of memory access operations is not so easy to assure. Accesses to external memory (e.g. static RAM) can have fixed latency. However, increases in memory bus frequency to achieve higher bandwidth have made that latency costly: 30 or more clock cycles being typical (Table 1), mostly due to the bus transaction setup time. Data caches [11] are the conventional solution to this problem, but the latency of each access depends on the preceding *reference string*, which is the sequence of *effective addresses* used by memory accesses within a task [5]. This dependence limits the effectiveness of WCET analysis for data caches, since it is only possible when (1) the reference string is independent of input data and (2) only statically allocated and stack data are used [17, 34].

Scratchpads [6, 32] can be used to store data as well as instructions. They implement time-predictable memory accesses, since scratchpad access latencies are independent of the preceding reference string. However, programs must explicitly manage the scratchpad space. This can be done at the programmer's direction ("overlay programming" [23]), but this is time-consuming and error-prone [15], so previous work has described algorithms to automatically allocate a subset of a program's data to scratchpad. This subset may be *static* [32] or *dynamically updated* during execution [6, 15, 33], but the forms of data and programs that can be supported are restricted. Specifically, time-predictable

[1]This work was supported by EU ICT projects eMuCo and JEOPARD, nos. 216378 and 216682.

scratchpad allocation algorithms only support statically allocated or stack data [6, 32], while non time-predictable algorithms can only support dynamic data if whole-program pointer analysis can identify every memory operation that could access each variable [33].

The contribution of this paper is a description and evaluation of the *scratchpad memory management unit* (SMMU), a new technology for embedded systems. The SMMU moves data between scratchpad and external memory, but also stores details of each move that is made so that the *logical address* of each data item is unchanged by the move. This eliminates pointer aliasing and invalidation. In this paper, the characteristics of the SMMU are demonstrated using code that could not be accepted by previous approaches. The new work satisfies three objectives that are not met together by any previous work. Firstly, any form of data can be supported, including dynamic data. Secondly, accesses are time-predictable since every load or store operation can be classified offline as a "scratchpad" or "external memory" access. Thirdly, whole-program pointer analysis is not required: previous work required such analysis [6, 33] or eliminated pointers entirely [32]. The SMMU lifts some of the restrictions forced on embedded hard real-time tasks by the limitations of WCET analysis, such as the lack of support for dynamic data [14]; a topic of importance as the complexity of embedded software grows, and object-oriented languages such as Java are applied to real-time systems [28].

Although the focus of this paper is on hard real-time systems, other embedded software may benefit. The SMMU acts on entire objects rather than fixed-size cache lines or pages, so the size and energy cost of the SMMU is proportional to its maximum number of objects rather than the size of the scratchpad memory. Consequently, the combination of an SMMU and scratchpad retains the energy advantage of a scratchpad-only design [31, 33], which is important in any battery-powered system. Additionally, any scratchpad allocation algorithm (e.g. [15, 33]) can be adapted to use the SMMU, since the SMMU implements the functionality of the *direct memory access* (DMA) controllers that are conventionally used to copy data between scratchpad and external memory. Explicit control of memory has been shown to benefit a range of embedded applications [21].

The layout of this paper is as follows. Section 2 gives a motivating example for the rest of the paper: a function that is not fully supported by previous work. Section 3 explains the problems that prevent data scratchpads being used to implement time-predictable support for pointers, and section 4 discusses the features of a time-predictable solution for these problems. In this paper, the suggested solution is the SMMU (section 5), which is evaluated in section 6. Section 7 has related work and the paper is concluded by section 8.

2. MOTIVATING EXAMPLE

Figure 1 shows the contents of the `ycc_rgb_convert` function from libjpeg [12]. This code forms part of the process of decoding a JPEG file into RGB data for display on a screen. It has been chosen as an example because previous techniques would force a time-predictable implementation to use external memory, with a high latency for each access.

Firstly, none of the data used by this function would be suitable for data scratchpad allocation using techniques described by Suhendra et al. [32] or Deverge and Puaut [6] because each variable is accessed using a pointer from dynamically allocated memory. The surrounding library code would change significantly to accommodate static allocation.

System	Latency/ CPU clock cycles	CPU freq/ MHz	Bus freq/ MHz
ARM PB11MPcore [1]	79	210	70
StrongARM-110 [30]	17	50	50
PPC 405 (FX12) [36]	33	100	100
Microblaze (ML505) [40]	31	125	125

Table 1: Measured latency for a load operation on four embedded systems, given in CPU clock cycles with data caches disabled.

Secondly, a time-predictable implementation could not use a data cache. The pointer values are unknown *and* some of the effective addresses are dependent on input data (e.g. `Crrtab[cr]`, since `cr := inptr2[col]`). Current WCET analysis techniques for data caches would force the majority of memory accesses in the function to use external memory [17, 34].

Future WCET analysis techniques for data caches might be able to support code of this kind. However, the results of tight WCET analysis would still be disappointing, as Table 2 illustrates. In Table 2, the approximate "best" and "worst" numbers of cache misses (and related data) are shown for Figure 1 and two cache sizes. These were computed using a genetic algorithm which searched the space of possible offsets for each array used in Figure 1[2]. The algorithm used the measured execution time as a fitness value and attempted to minimize (or maximize) it, finding an estimate for the best (or worst) case. The huge difference between the "best" and "worst" cases observed here is entirely due to *conflict misses* [11]. A conflict miss occurs in any data cache whenever two or more items of data are competing for a single cache line. The combination of a data cache and full support for pointers forces WCET analysis to account for all possible conflict miss scenarios (or use some other safe upper bound). Although the figures in Table 2 are specific to this example, a similar disparity between the best and worst case will be found whenever more than n unknown addresses are being accessed within a loop given an n-way associative cache.

The remainder of this paper shows how the SMMU is able to support code such as Figure 1 and approach the "best" case of Table 2 for *any input data* and *any pointer values*.

3. DATA SCRATCHPAD PROBLEMS

In principle, a scratchpad *could* be used to store all of the objects used by `ycc_rgb_convert`. Consider an *object* as a contiguous block of memory, such as the space allocated for `output_buf` or `Crrtab`, and consider a *pointer* as a reference to an object. All of the objects used in `ycc_rgb_convert` could be copied to scratchpad at the start of the function, then used from scratchpad by changing all pointers to reference the

[2]To generate this data, Figure 1 is executed on an image of size 1152 by 864 pixels. The cache line size is fixed at 16 bytes. The access times are based on the assumption that a cache hit costs 1 clock cycle, a cache miss costs 50 (an optimistic example according to Table 1), and code that does not access memory costs nothing. It is optimistically assumed that other code only affects the cache by invalidating `input_buf`.

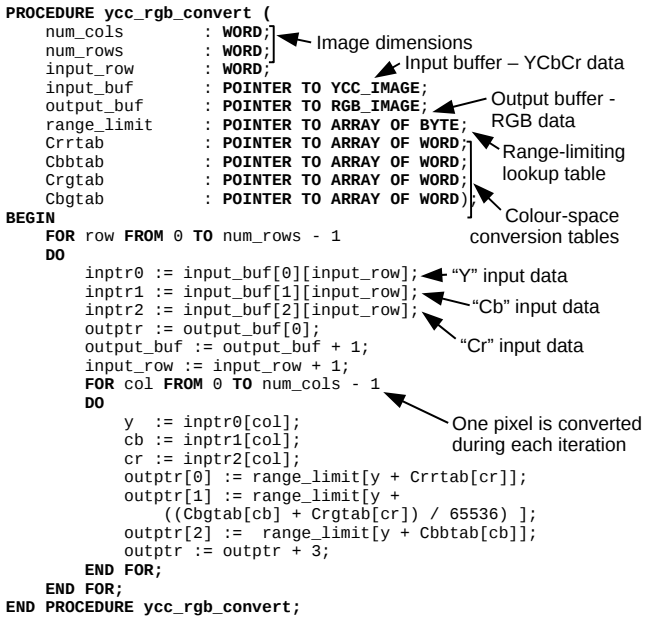

Figure 1: A program fragment from the libjpeg software [12] making heavy use of pointers (pseudocode translated from C).

Cache size (bytes)	8192	16384
Memory accesses	12942720	
"Best" case miss count	123951	106058
"Worst" case miss count	4281635	4304087
"Best" access time	19016319	18139562
"Worst" access time	222742835	223842983
"Worst"/"Best" ratio	11.7	12.3

Table 2: Approximate "best" and "worst" cases of data cache behavior for Figure 1 obtained by search.

scratchpad memory instead of the external memory. After execution, the modified copy of `output_buf` could be written back to external memory.

While straightforward, this approach will not be taken by time-predictable scratchpad allocation algorithms as proposed by Suhendra [32] or Deverge [6] because the objects are dynamically allocated. Suhendra and Deverge omit support for pointers with addresses that cannot be predicted offline because of the additional complexity that they introduce. This complexity falls into three categories: *pointer aliasing*, *pointer invalidation* and *object sizing*.

3.1 Pointer Aliasing

Suppose that the objects used by `ycc_rgb_convert` were copied to scratchpad just after the function was called, as suggested above. In this arrangement, correct behavior requires that the memory allocated for `output_buf` is not shared by any of the inputs. This happens to be a safe assumption in this case, but *not* in general. It is an instance of the *pointer aliasing* problem [4], where two or more pointers refer to the same data. The pointer aliasing problem is not specific to C: it also exists in languages where pointers are strongly typed, e.g. the references used within Java [35].

The issue of pointer aliasing becomes a problem when an assumption must be made regarding the nature of two or more pointers, viz. whether they point to the same object or not. Sometimes, this can be determined by pointer analysis at compile time, and this is part of the purpose of the restrictions applied by [6]. However, such analysis is not possible in general. In other research fields, unknown pointer aliasing is handled by making a safe assumption. For example, compilers can ensure that all memory access operations are performed in program order, which inhibits some optimizations. In the case of scratchpad allocation, the only safe assumption is to avoid moving possibly-aliased objects to a new physical location.

3.2 Pointer Invalidation

Some scratchpad allocation algorithms [31, 32] are *static* in the sense that the set of objects stored in scratchpad is fixed throughout execution. *Dynamic* allocation algorithms [6] make better use of scratchpad space because the set of objects stored in scratchpad can change during execution, matching the requirements of the current function. Pointer invalidation can occur when an object is moved from scratchpad to external memory (or vice versa). Pointers created *before* the move may not be correct afterward. In the case of scratchpad allocation, the safe assumption is to either avoid moving objects (which would force a static allocation) or identify every usage of a specific object.

3.3 Object Sizing

Time-predictable scratchpad allocation algorithms depend on knowledge of the size of each object used by the program. The restrictions applied by [6, 32] make it easy to determine the size of each object used by the program offline, because there is a 1-1 relationship between the variables in the source code and the objects they represent. Variables can only be created statically (i.e. as global variables) or on the stack, so each has a size that is fixed at compile time. The size can be computed at the point of use by examining the type associated with the variable. When pointers and dynamic data are introduced, this is not possible. While a dynamically allocated object usually *does* have a fixed size, dynamically-sized arrays being the exception, the size information is not exposed to the compiler and it cannot be computed where a pointer to the object is used. This means that a scratchpad allocation algorithm cannot know (1) how much space is required in scratchpad, or (2) the time taken to transfer the data between external memory and scratchpad. The safe assumption is to insist that every object has a compiler-known size, i.e. that all objects are allocated statically or on the stack.

3.4 Previous Solutions

Two of the three pointer issues listed above have been handled by one scratchpad allocation algorithm, which is described by Udayakumaran, Dominguez and Barua [33]. It supports dynamic data, and the subset of objects allocated to scratchpad can change dynamically.

The Udayakumaran approach avoids pointer aliasing by assigning each object to a fixed location, either in scratchpad or external memory. This fixed location is determined at runtime by a new implementation of `malloc`. Addresses do not change so pointer aliasing is not an issue, and because allocations are carried out at runtime, object sizing is not

an issue. However, the approach is not time-predictable because the latency of memory access operations depends on the locations of objects, which are unknown offline.

The Udayakumaran approach uses whole-program pointer analysis to prevent the effects of pointer invalidation. Pointers are still invalidated when objects are moved, but whole-program pointer analysis ensures that accesses to those objects only occur while they are loaded into scratchpad. This is not a perfect solution; inefficient safe assumptions will be made in some cases.

An alternative approach would eliminate pointers altogether, e.g. by using a restricted language subset [14] or by removing the assumption of a single uniform memory space shared by all parts of a program, so that programs can be efficiently compiled for the scratchpad paradigm [2]. However, these approaches could be seen as too restrictive, since they reduce the available language feature set and break compatibility with legacy software, inhibiting code reuse.

4. TIME-PREDICTABLE SOLUTION

Hard real-time tasks can benefit from scratchpads if every memory access operation can be classified offline as "external memory" or "scratchpad", since this simplifies WCET analysis while providing a way to reduce the latency of most memory accesses.

Scratchpad allocation algorithms demonstrate that a subset of the data used by a program can be allocated to the scratchpad in order to minimize the estimated WCET [6,32]. However, solutions to the problems posed by *object sizing*, *pointer aliasing* and *pointer invalidation* need to be found before pointers and dynamic data structures can be used in hard real-time tasks. The solution must retain time-predictability (every memory access must have a known latency) and avoid any requirement for whole-program pointer analysis, since this would restrict the set of programs that could be supported.

4.1 Objects, Base Pointers and Offsets

Conventionally, an object i of size s_i occupies a range of contiguous logical memory addresses $[b_i, b_i + s_i]$, where b_i is the *base pointer* of i. The base pointer holds the lowest address allocated to i.

Previous scratchpad allocation approaches have tracked each object i using whole-program pointer analysis, finding all accesses to i. However, this is unnecessary. Consider that all memory accesses to i can be decomposed into the form of base pointer b_i plus an *offset* o. The effective address of a memory access operation x using i is:

$$A_{i_x} = b_i + o_x \tag{1}$$

In a correct program [13], $A_{i_x} \in [b_i, b_i + s_i]$. Additionally, every section of code that uses i will obtain b_i first.

This gives an opportunity to load i into scratchpad before it is accessed. The actual value of b_i and the identity of object i are unknown offline. b_i is determined at runtime, and at that point, the memory at $[b_i, b_i + s_i]$ can be moved into scratchpad before any address A_{i_x} is accessed. This guarantees that every access to i is directed to scratchpad.

Base pointers have been useful to earlier research [9], where parts of objects have been prefetched into cache using memory locations that are computed dynamically by adding offsets (derived from loop iteration counters) to a base pointer that is unknown before execution. The same principle is used here to ensure that an object is moved into scratchpad before access at runtime without knowing the object's location (b_i) or its identity (i) offline.

4.2 Object Sizing

The maximum possible size s_i of each object i that might be accessed by instruction x must be known offline, so that (1) the amount of scratchpad space required can be determined, and (2) the copying time can be calculated. s_i is easy to determine at compile time if x only refers to a known subset of statically-allocated data (section 3.3). s_i is also easily determined if every object that could be referenced by x has a fixed size. However, x may also refer to dynamically-allocated data with a size unknown to the compiler.

The problem of determining s_i is similar to the issue of *loop bounds* within hard real-time tasks. Loop bounds are needed for WCET analysis [26] and may be obtained automatically in some cases [7], but the programmer is usually expected to specify them using code annotations. The same applies to object sizes when they cannot be determined automatically. The problems are so similar that s_i could even be derived from a loop bound in the case of iteration though an array.

4.3 Logical and Physical Addresses

The key to solving both *aliasing* and *invalidation* problems is to separate *logical* and *physical* addresses, since this allows an object i to be resident at an unchanging logical address b_i even if it is moved to a new physical address (e.g. in scratchpad). Let f be a remapping function $f : L \mapsto P$ that translates a logical address L to a physical address P. An object i always occupies a range of logical addresses $[b_i, b_i + s_i]$. Before the object is mapped to scratchpad, it also exists at physical address b_i. When the object is moved to scratchpad, a new copy is created at physical address t_i. The remapping function accounts for this change by adjusting logical addresses in the range $b_i \leq L < b_i + s_i$ so that the copy in scratchpad is used by every memory access referencing i:

$$P = f(L) = \begin{cases} L + t_i - b_i & \text{if } \exists i, b_i \leq L < b_i + s_i \\ L & \text{otherwise} \end{cases} \tag{2}$$

The logical address of the object i remains L, so pointers are not invalidated by relocation. Aliases of i work correctly, as the mapping between logical addresses and physical addresses is always 1-1.

For example, imagine a computer with 100 words of memory. Physical addresses 0 through 89 refer to external memory, while 90 through 99 refer to scratchpad. Suppose object j is located at *physical* address 10 and has size $s_j = 5$. j is not mapped to scratchpad, so an access to *logical* address 10 goes to external memory: $f(10) = 10$. j occupies physical and logical address range $[10, 15]$. Next, j is mapped to scratchpad at physical address $t_j = 90$. j now occupies physical address range $[90, 95]$, and an access to logical address 10 is routed to the new location: $f(10) = 90$. A program accessing address 10 sees no change in functionality. The operation is faster but produces the same result. j was moved, but retained logical addresses $[10, 15]$.

f can be easily extended to allow more than one object $(i_0 \ldots i_{N-1})$ to be loaded into scratchpad simultaneously, although the maximum number of objects N is fixed at design time. f must account for the possibility that objects might

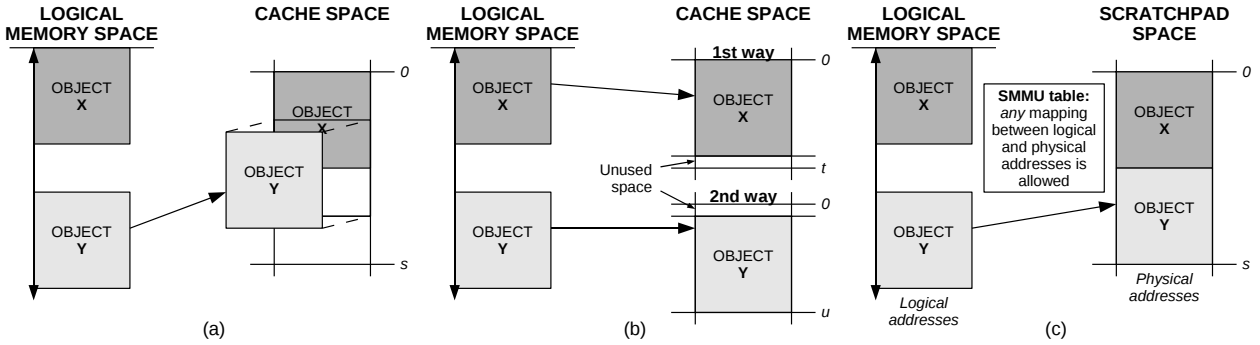

Figure 2: (a) Conflict misses can occur during preloading, evicting objects that were preloaded earlier. (b) An N-way associative cache can be used to isolate each object in its own cache way, but this leads to inefficient use of cache space when objects do not fill an integer number of ways. (c) The SMMU table decouples associativity from storage, so resources are used more efficiently.

overlap with each other in the logical address space due to pointer aliasing. This is handled by enforcing a priority encoding on the copies. If the memory used by two objects overlaps, the highest-numbered object is always considered the most recent (equation 3). f requires $2N$ comparisons in the worst case. We note that hardware can implement all comparisons in parallel, as N is constant.

$$P = f(L) = \begin{cases} L + t_i - b_i & \text{if } \exists i, (b_i \leq L < b_i + s_i) \wedge \\ & \quad \forall j, (j \leq i \\ & \quad\quad \vee \neg[b_j \leq L < b_j + s_j]) \\ L & \text{otherwise} \end{cases} \tag{3}$$

4.4 Possible Implementations

The logical to physical address mapping performed by f (equation 3) could be implemented by the address comparison hardware within an associative cache (at the cost of some granularity, as each range $[b_i, b_i + s_i]$ would need to be aligned to the cache line size). Instead of copying i into scratchpad, i could be *preloaded* into a *locked* cache [22]. Given sufficient cache space, this would guarantee that every access to i would be a cache hit [9].

However, if N objects needed to be preloaded, this would be prone to conflict effects: a preloading operation for object Y could evict part of object X. Like a conflict miss, the extent of this effect would be entirely dependent on the base pointer values b_{X} and b_{Y} (Figure 2(a)). A higher degree of cache associativity (say, N-way) would allow up to N objects to be preloaded and locked in cache simultaneously, but each object would only be able to take up an integer number of "ways". Unless the number of ways was very large, this would be inefficient, with wasted space wherever an object did not exactly fill its allotted space (Figure 2(b)).

A fully associative cache would seem to be an ideal solution, because the size of each "way" is a single cache line [11]. However, the high energy usage and high logic area usage of the required *content addressable memory* (CAM) limits the practical size of a fully associative cache [8]. An object of size s would require $O(s)$ cache lines, and therefore $O(s)$ CAM resources, so the cache would be limited to small numbers of small objects.

Resources would be used more efficiently if there were

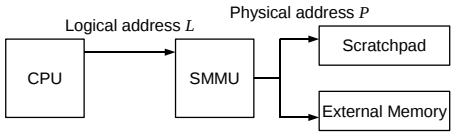

Figure 3: The relationship between the CPU, SMMU and external memory.

$O(N)$ comparators for N objects. Caches are no substitute for f as implemented by equation 3, because f decouples the associativity which matches addresses from the actual storage space for objects, meaning that $O(1)$ comparators are required for an object regardless of its size.

The proposed hardware is the *scratchpad memory management unit* (SMMU), which implements remapping function f (equation 3) and a form of DMA controller that moves data between the scratchpad and the external memory. Using the SMMU, each object can be of *any* size, subject to scratchpad space limitations (Figure 2(c)). This means that fewer comparators are needed - instead of $O(\sum s_i)$, just $O(N)$ are required for N objects.

5. THE SCRATCHPAD MMU

The *scratchpad memory management unit* (SMMU) translates *logical* addresses (from a CPU) into *physical* addresses which can be directed to a data scratchpad or external memory (Figure 3). It is a hardware implementation of equation 3, combined with a DMA controller for moving data between scratchpad and external memory. For simplicity, it is assumed that the external memory is static RAM, so each DMA transfer is time-predictable.

The SMMU implementation is a table of *minimum* (b_i) and *maximum* ($b_i + s_i$) logical address pairs, one for each object i mapped into scratchpad. Objects are only mapped and unmapped at the direction of the program. The SMMU provides *transparent* access to memory, i.e. the logical address of an object is unchanged by mapping and unmapping.

Figure 4 shows the internal structure of the SMMU, which implements equation 3. A logical address L is generated by the CPU and received on the left of Figure 4. It is compared to the SMMU table (A) which contains $N = 2^n$ groups of the following registers:

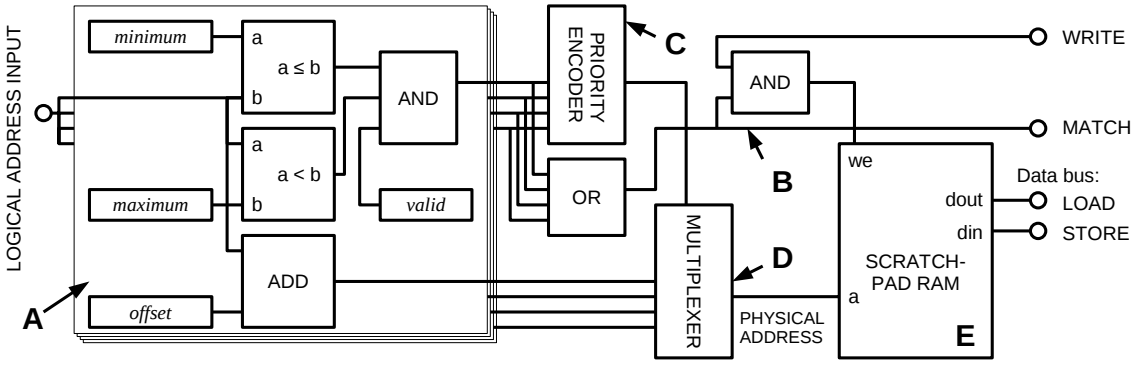

Figure 4: Scratchpad memory management unit (SMMU) hardware.

- *minimum*: minimum logical address of the object (b_i).
- *maximum*: minimum plus the object size ($b_i + s_i$).
- *offset*: the value to be added to L to convert it into a physical address in scratchpad memory (t_i).
- *valid*: a single bit that is 1 for valid table entries.

L matches if it is between *minimum* and *maximum* for some entry. If any of the 2^n groups matches the incoming address, then the match output (B) is asserted. If two or more match, then two objects overlap, and the highest-numbered match is selected by a priority encoder (C) and used as the n-bit select input for a multiplexer (D). This gives the physical address P; it is passed to the scratchpad memory (E).

Addresses that match inside table A are serviced quickly. Not all addresses will match. For example, some logical addresses may be used for IO and some objects may not be mapped to the scratchpad at all (e.g. because they are too large to fit, or because the cost of transferring them into scratchpad is greater than the time saved by not accessing external memory). If an address does not match, then $P = L$, and the SMMU services the access operation using external memory.

5.1 SMMU Operations

The SMMU is a little more complex than illustrated in Figure 4 because two further operations are required. These are OPEN and CLOSE; they map and unmap areas of external memory into scratchpad. This process must copy data from/to external memory and consider the objects that are already mapped into scratchpad in order to deal with the issue of overlapping memory in the logical address space. A priority encoding in equation 3 ensures that the highest-numbered row in the SMMU table is considered to contain the most up to date copy in the event of overlapping. (External memory always contains the least up to date copy.) OPEN and CLOSE can be used directly by a programmer (e.g. Figure 5) or placed automatically by a scratchpad allocation algorithm (section 5.2).

OPEN takes three parameters: logical base pointer address b_i, object size s_i, and physical copy location t_i. The contents of logical addresses in the range $[b_i, b_i + s_i]$ are copied using DMA from external memory into the scratchpad (physical address range $[t_i, t_i + s_i]$). The DMA process uses remapping function f to translate each logical address

```
size_t strlen(const char *s)
{
    size_t i;
    table_ref sr = OPEN(s, 100, 0);
    for (i=0; *s != '\0'; i++, s++) {}
    CLOSE(sr);
    return i;
}
```

Figure 5: OPEN and CLOSE, added by hand.

$L \in [b_i, b_i + s_i]$ to a physical address $P = f(L)$. If the logical address is not mapped to scratchpad, then $L = P$ and the data is taken from external memory. If $L \neq P$, then P must be a scratchpad memory address containing the previous most recent copy of the data at L. In this case, the data must be taken from scratchpad.

The OPEN operation creates an entry in the table for i (Figure 4, A) so that future memory access operations can access the contents of i from scratchpad instead of external memory. To use OPEN, the program must guarantee that the physical scratchpad space $[t_i, t_i + s_i]$ is unused. OPEN is time-predictable, as it has a maximum time bound which can be computed as a function of s_i. t_i can be specified by the programmer, but generating this value is really a task for a scratchpad allocation algorithm: existing algorithms [6] already specify methods for doing this.

CLOSE takes one parameter: a table reference i as returned by OPEN. It reverses the OPEN operation associated with that reference, writing the scratchpad contents $[t_i, t_i + s_i]$ back to external memory ($[b_i, b_i + s_i]$). As with OPEN, the DMA process uses f to translate each logical address $L \in [b_i, b_i + s_i]$ to a physical address $P = f(L)$, and if $L \neq P$, then the data is written to scratchpad memory at address P. CLOSE has the same time bound as OPEN.

The timing of a memory access operation may be calculated for WCET analysis purposes by inspecting the code to determine the base pointer used to generate the effective address (equation 1). If the object associated with the base pointer has been OPENed, then the memory access can be guaranteed to be serviced quickly (section 4.1). Otherwise, the WCET analysis tool must assume that the memory access requires the external memory, which will be slow.

5.2 Allocation Algorithm

An existing scratchpad allocation algorithm can be extended to support the SMMU. For example, the algorithm specified by Deverge [6] may be extended by:

Component	Critical path	Max. freq/ MHz
SMMU	Across a table comparator	143
Microblaze	In "Shift_Logic_Module"	162
Both	Microblaze ALU + Table	111

Table 3: The maximum clock frequency for various components of the FPGA implementation, identified using Xilinx synthesis tools.

- Using OPEN and CLOSE to transfer data between external memory and scratchpad: these replace the DMA commands currently in use.
- Placing an upper limit of 2^n on the total number of objects that can be in use simultaneously. This is an additional constraint in the integer linear program.
- Replacing the notion of "variables" within the algorithm with "objects". From section 4.1, an object i comes into existence whenever a base pointer b_i is created by an instruction. It continues to exist as long as that base pointer may form part of an effective address. The link between a memory access operation and the variables that it might access becomes irrelevant. With the SMMU, it is only important to link memory access operations with base pointers: a simpler problem that does not require whole-program pointer analysis, only local analysis within a single function.

Other parts of the algorithm can be retained, such as the assignment of objects to locations in the scratchpad (used to generate each t_i), and the iterative process of WCET reduction to account for new worst-case execution paths.

6. SMMU EVALUATION

This section presents evidence to show that the SMMU is practical. It is evaluated in three different ways. The first of these looks at the properties of a *hardware implementation* (section 6.1). The SMMU is then applied to a single function (section 6.2) and an entire program (section 6.3).

6.1 Hardware Implementation

The SMMU hardware is quite different to a conventional data cache. In a typical cache design, the low bits of the address are sent directly from the CPU to the tag and data memory of each "way" of the cache [11]. Suppose that this happens in clock cycle z. The mechanism only becomes aware of whether the access is a hit or miss during clock cycle $z+1$, when the tag can be compared with the high address bits. Conversely, the SMMU design must determine whether an access matches within the table during clock cycle z, because this information is required in order to compute the physical address (the *offset* must be known). Therefore, the use of an SMMU can change the location of the *critical path*, since more combinational logic is needed *before* the on-chip memory is accessed. (The critical path sets the upper bound on the clock frequency, since it is the slowest path through combinational logic.)

Experiments with an FPGA implementation of the SMMU indicate that this can be a problem for some CPUs. A VHDL model of the SMMU was implemented and synthesized for a Xilinx Virtex-5 FPGA using the Xilinx Embedded Development Kit (EDK) software [38]. The Xilinx ML505 prototyping board was used as a host [40]. The SMMU model connects via a Processor Local Bus (PLB) master interface to any memory controller that is supported by EDK. Four Virtex-5 block RAMs are used as the scratchpad memory. Virtex-5 FPGA implementations of hardware are not as fast or as dense as fully custom silicon implementations, so all of the clock frequencies could increase for an *application specific integrated circuit* (ASIC) implementation.

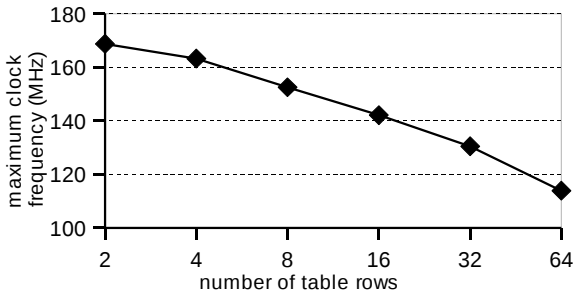

Figure 6: The effect of the table size and the number of address bits on the maximum clock frequency.

Table 3 shows how the maximum clock frequency of the SMMU changes when the Microblaze CPU [39] is introduced. In isolation, the clock frequency of the SMMU and Microblaze are both reasonable, but when the two are combined, the critical path passes through the Microblaze ALU because the ALU computes the effective address for every load or store operation. This limits the maximum clock frequency; the ALU is already one of the slower parts of the CPU due to the carry chain that runs through it.

This effect can be reduced by changing the table size (Figure 6). It is clear that large tables (32, 64 entries) have a negative impact on the maximum frequency; an extra delay exists because of the spacing between the elements and the larger multiplexer. The ideal size for the table is a question for future work.

6.2 Timing of a C Function

This section returns to the example of Figure 1 to see what benefit is provided by the SMMU and its associated scratchpad. Manual placement of OPEN and CLOSE operations is used to map the nine objects used by the function into scratchpad memory. The sizes of each array are given in Table 4 along with sample transfer costs. In this case, all of the objects are dynamically allocated.

The hardware model used for Table 2 is extrapolated to find the general cost of a scratchpad operation as follows. From footnote 2, a cache hit costs 1, a non-memory operation costs 0, and a cache miss costs 50 clock cycles. Assume that a cache miss transfers exactly one line (16 bytes). If the memory bus frequency is the same as the CPU core frequency and 4 bytes are transferred per clock cycle, then the bus transaction setup time is 46. Assuming that burst-mode transactions can have any size, the cost of transferring x bytes (e.g. for OPEN or CLOSE) is $46 + \lceil \frac{x}{4} \rceil$. Notice that the *bandwidth* of the memory bus is high (one word per clock cycle), but *so is the access latency*, because it takes 46 clock

Name	Size/ bytes	Transfer cost OPEN	CLOSE
`inptr0`	1152	334	334
`inptr1`	1152	334	334
`inptr2`	1152	334	334
`outptr`	3456	910	910
`range_limit`	1408	398	398
`Crrtab`	1020	301	301
`Crgtab`	1020	301	301
`Cbgtab`	1020	301	301
`Cbbtab`	1020	301	301
Total	12400	3514	3514

Table 4: The sizes and OPEN/CLOSE costs of the nine objects referenced by Figure 1, assuming the hardware model of Table 2.

cycles to begin a transaction. This appears to be typical of real designs (Table 1).

The OPEN and CLOSE operations allow the SMMU and scratchpad to act as a perfect data cache for a subset of the program's data: specifically, the OPENed objects. The uncertain cost of cache misses is replaced by the known cost of the transfer operation, requiring 7028 clock cycles for each execution of Figure 1. Since Figure 1 is called 864 times (once for every row of the image), the total transfer cost is 6072192. This leads to an execution time of 19014912 when calculated using the criteria for Table 2. This is 1.05 times slower than the "best" cache case, but 11.8 times faster than the "worst" cache case (Figure 7).

Better performance would be achieved if `num_rows` $\neq 1$, because the five objects that represent lookup tables (`Crrtab`, etc.) could stay resident between iterations [38]. These tables remain in cache if there is (1) enough space and (2) no conflict miss occurs. This persistence is one of the ways that data caches can outperform the SMMU and scratchpad *in ideal conditions*. On-demand loading of data can also be more efficient than simply preloading everything; for example, most elements of `range_limit` are never used, but the entire object has to be loaded to ensure that every access operation has constant timing.

These disadvantages mean that the SMMU and scratchpad can be slower than a data cache in the average case. The combination benefits hard real-time tasks: 19014912 is *both* the worst case and the best case. However, the energy benefits gained through the use of a scratchpad must also be considered [31]. The energy overhead of the SMMU is small because relatively little hardware is needed to implement it [38], so the SMMU can simplify the process of automatic scratchpad allocation for any program, not just real-time tasks, enabling data caches to be removed from non real-time embedded systems.

6.3 JPEG Decoding Program

In this section, evidence is taken from a case study [38] using a complete JPEG decoder [12] to support the claim that the SMMU is widely applicable and indicate the cases where it is less effective. This part of the evaluation makes no distinction between dynamically allocated data and data on the stack (automatic variables): the SMMU is used to map both into scratchpad within significant libjpeg functions (Figure 8). These account for 87% of the execution

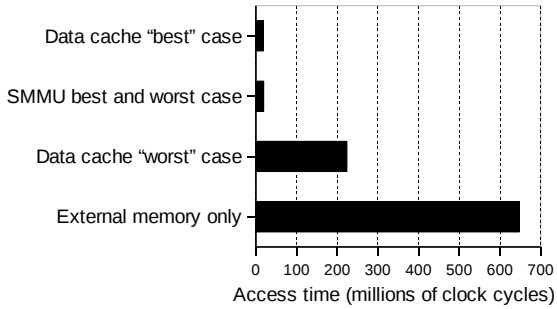

Figure 7: Data access times for Figure 1 in four scenarios: "best" and "worst" cases for a 16kb data cache (Table 2), using external memory only, and using the SMMU.

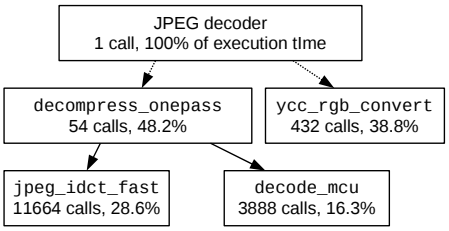

Figure 8: JPEG decoding functions [12] considered by the case study [38].

time during decoding of the image used for Table 2. OPEN and CLOSE operations were added to these functions, assuming a scratchpad size of 16kb and a table size of 16. Table 5 shows the effect of OPEN and CLOSE on the execution time of each function.

Overall, 90% of memory operations are routed to scratchpad, but the remaining 10% are still significant, accounting for 61% of the execution time, as each external memory access costs more than a scratchpad access. The result is 4.23 times slower than a perfect data cache. In order to improve this, the remaining functions must also use the SMMU in order to reduce this. Automatic scratchpad allocation would be important here, as there are many rarely-used functions in libjpeg. Within the most commonly used functions, the SMMU is useful, but two problems limit its benefit:

Firstly, *some objects are too small to be effectively mapped.* 18% of accesses in `jpeg_idct_fast` are store operations to the output buffer. These are a problem for OPEN and CLOSE because of their *fragmented* access pattern: `jpeg_idct_fast`

Function	Execution Time External memory	SMMU	Coverage
`jpeg_idct_fast`	15.5	1.72	99.8%
`decode_mcu`	10.1	4.77	59.4%
`decompress_onepass`	13.1	3.18	86.7%
`ycc_rgb_convert`	29.2	1.25	99.9%
JPEG decoder	16.1	4.23	89.8%

Table 5: Normalized execution times of functions from Figure 8. "Coverage" indicates the percentage of memory operations routed to scratchpad.

produces square blocks of 8 by 8 pixels, so the largest object is 8 bytes. Small objects do not respond well to OPEN and CLOSE because the bus transaction setup time becomes significant. In this case, knowledge about the code can be applied to combine blocks together. However, this is not always possible. `next_input_byte` references input data from the JPEG file, an object of unbounded size that is accessed one byte at a time. Here, OPEN and CLOSE are no better than direct access to external memory. This problem could only be resolved by the programmer, who could refactor the code to make more effective use of resources (e.g. copying input data into a small temporary buffer).

Secondly, *some objects are too large to be mapped*. For example, `decode_mcu` uses lookup tables referenced by pointers named `actbl` and `dctbl`. Many possible tables may be referenced by these pointers, making OPEN and CLOSE extremely inefficient, and accesses are sparse: only a few bytes may be used from each. The obvious solution is to OPEN at an earlier point but this is ineffective because of the number of tables that could be referenced. All would need to be OPENed together. The tables are too large (11392 bytes in total) and accesses within them are too sparse to make good use of OPEN and CLOSE. Again, this problem could only be resolved by the programmer.

The price of time predictability is a greater average execution time in many cases. However, the increase is not as great as might be feared. Within the functions considered, most memory operations could be routed to scratchpad with corresponding decreases in both the average and worst-case execution times. In two cases, `jpeg_idct_fast` and `ycc_rgb_convert`, the result approaches that of a perfect data cache, and the other cases are still far better than accesses to external memory.

7. RELATED WORK

This work fits into a recent research effort to specify computer architectures where time predictability is as important as functional correctness [2, 28]. In this field, much research has been focused on the CPU [10,19,37], where worst-case behavior can be difficult to determine. Specific problems found in conventional designs include *timing anomalies* [18] and effects that can occur in branch predictors. Some examples of these problems in real CPUs are discussed by Heckmann et al. [10]. CPU analysis often assumes a time-predictable memory subsystem (e.g. perfect data caches [19]), while explicitly time-predictable CPU designs assume that scratchpads will be used [16, 24, 37]. All of this work could benefit from the SMMU.

Previous work has attempted to address the problem of bounding the latency of memory access operations through cache analysis. Instruction cache analysis is now quite a mature topic [20], but data cache analysis remains limited to code that uses predictable reference strings [29, 34].

Schoeberl proposes improvements to the data cache subsystem to reduce conflict misses and simplify WCET estimation [27]. The proposal splits the data cache into several parts, each for a different form of data, e.g. heap, static and stack. Alternate designs can be used in each case, such as a fully associative cache for heap data to solve the conflict miss problem. However, the SMMU may be a better solution for static and heap data because (1) timing is not dependent on the reference string, and (2) the limited resources of the SMMU table can be applied to objects of any size, whereas the limited resources of a practical fully associative cache impose a strict size limit (section 4.4).

Full support for pointers may not be necessary in time-predictable code. It can be argued that hard real-time tasks are a special case where restrictions such as "no pointers" are not a problem [14]. This work also presumes that C and Java code should be largely unchanged by the use of a scratchpad, but other researchers believe that languages should evolve beyond C and Java to better match the capabilities of real hardware. For example, *actor oriented* programming removes the assumption of a single uniform memory space that is shared by all parts of a program, allowing for an efficient mapping onto the scratchpad paradigm [2].

Application-specific control of resources is often advantageous in energy-constrained embedded systems. Consequently, some scratchpad management techniques optimize for energy reduction rather than WCET [15, 33], which is also facilitated by the replacement of data caches with a scratchpad [31]. The benefits of application control of memory resources are explored in [21].

8. CONCLUSION

This paper has described problems facing the implementation of time-predictable load and store operations. Some of these are consequences of data caches, while others occur when scratchpads are used to store dynamic data. This paper has explained that previous solutions have attempted to solve these problems in ways that do not meet the requirements of hard real-time tasks. It has proposed and evaluated a new solution in the form of a scratchpad combined with a scratchpad memory management unit (SMMU).

The SMMU acts as a perfect data cache for a subset of the data used by a program. Using the SMMU, the execution time of a program may be independent of its reference string, any form of data can be used including dynamic data structures, and whole-program pointer analysis is not required. The SMMU allows hard real-time tasks to take advantage of pointers and reduces the effort required to add scratchpad support to a program. It can be used with scratchpad allocation algorithms in previous work.

Evidence shows that the SMMU works as specified, has a reasonable hardware cost, and can be applied to real C code. Its design is quite different to that of a conventional cache (section 6.1). The SMMU has been successfully applied to functions within an example program (sections 6.2 and 6.3). These functions could not be fully supported by WCET analysis for data caches or previous automatic scratchpad allocation approaches. While the SMMU and scratchpad are not as effective as a data cache in ideal conditions, they are considerably better than a data cache in worst-case conditions. As the size of the SMMU is proportional to the number of objects it can contain rather than the maximum scratchpad size, the energy and space advantages of scratchpads versus caches are retained by the SMMU approach. Further work is being carried out to understand how often the SMMU can be useful in typical programs.

9. ACKNOWLEDGMENTS

Thanks go to Martin Schoeberl, Andy Wellings and Ian Gray for their reviews of drafts of this paper and helpful discussions. Thanks also to the EMSOFT reviewers whose comments were helpful in revising this paper.

10. REFERENCES

[1] ARM. Platform Baseboard for ARM11 MPCore. `http://www.arm.com/products/DevTools/PB11MPCore.html`.

[2] 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.

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

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

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

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

[7] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. *LNCS*, 1300:1298–1307, 1997.

[8] S. Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[9] E. H. Gornish and A. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. *Int. J. Parallel Program.*, 27(1):35–70, 1999.

[10] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE*, 91(7):1038–1054, 2003.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[12] Independent JPEG Group. `http://www.ijg.org/`.

[13] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers. In *Proc. AADEBUG*, pages 13–26, 1997.

[14] R. Kirner and P. Puschner. Discussion of Misconceptions about WCET. In *Proc. WCET*, pages 61–64, 2003.

[15] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Proc. PACT*, pages 329–338, 2005.

[16] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proc. CASES*, pages 137–146, 2008.

[17] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *Proc. RTCSA*, page 255, 1999.

[18] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proc. RTSS*, page 12, 1999.

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

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

[21] A. Patil. Application-specific resource management in real-time operating systems. PhD Thesis YCST-2007-25, University of York, 2007.

[22] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proc. WCET*, Vienna, Austria, June 2002.

[23] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, pages 1484–1489, 2007.

[24] P. Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. WCET*, 2002.

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

[26] P. Puschner and A. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Syst.*, 13(1):67–91, 1997.

[27] M. Schoeberl. Time-predictable cache organization. In *Proc. STFSSD*, March 2009.

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

[29] R. Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *Proc. EMSOFT*, pages 203–212, 2007.

[30] Simtec. EB110ATX (codename CATS). `http://www.simtec.co.uk/products/EB110ATX/`.

[31] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proc. ISSS*, pages 213–218, New York, NY, USA, 2002. ACM Press.

[32] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. RTSS*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[33] 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.

[34] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *Trans. on Embedded Computing Sys.*, 7(1):1–38, 2007.

[35] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. PLDI*, pages 131–144, 2004.

[36] J. Whitham. Virtual Lab - Board Server Hardware. `http://www.jwhitham.org.uk/c/vlab/fx12hw.html`.

[37] J. Whitham and N. Audsley. Predictable Out-of-order Execution Using Virtual Traces. In *Proc. RTSS*, pages 445–455, 2008.

[38] 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.

[39] Xilinx. Microblaze processor reference guide. Manual UG081, Xilinx Corporation, 2005.

[40] Xilinx. ML505 User Guide. Manual UG347, 2008.