The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing, and Case Study[1]

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 report proposes the scratchpad memory management unit (SMMU) to act as a perfect data cache for a known subset of the data used by a program. This enables the execution time for each load or store operation in the program to be precisely determined. The SMMU must be explicitly controlled by the program, which commands the addition and removal of objects from the SMMU and its associated scratchpad memory.

This report explains why the SMMU enables memories with high access latency to be used efficiently within hard real-time embedded systems. It describes the SMMU in abstract form, explaining how it solves the memory latency and pointer aliasing problems and how programs could use it. Then, it describes the implementation of a version of the SMMU for the Microblaze processor, enabling SMMU experiments within real embedded systems. Lastly, it conducts a case study involving a large C program to understand the costs and benefits of the SMMU for real code. A memory profiler is used to detect objects that can be usefully allocated within the scratchpad by identifying their base pointers.

The implementation for Microblaze is found to be practical in terms of clock frequency and logic area. The case study shows that the SMMU capabilities cannot be applied to all variables, but the performance of the SMMU within the functions being considered is within a factor of two of the performance of a perfect data cache.

1. Introduction

Time-predictable execution of *memory accesses* (i.e. *load* and *store*) is an important architectural feature for *hard real-time embedded systems*. Hard real-time means that the *worst case execution time* (WCET) for a task is *guaranteed* to be less than or equal to its deadline [1]. WCET analysis estimates an upper bound on the execution time of a task [2]. This upper bound is never less than the true worst case: analysis must be *safe*. It should also be as close as possible to the true worst case (i.e. the analysis should be *tight*).

This report describes challenges that affect the prediction of memory access times in hard real-time embedded systems (sections 1.1-1.3). It lists the problems with previous approaches (section 1.4). It suggests a solution for these problems in the form of a *scratchpad memory management unit* (SMMU) (section 2). The solution is implemented and interfaced to a Microblaze soft CPU core (section 3), then evaluated using an FPGA (section 4). A case study is carried out to evaluate the effects of the SMMU on a large C program (section 5). Section 6 gives a conclusion.

1. This work has been supported by the EMUCO and JEOPARD projects which are funded by the European Commission Seventh Framework Programme.

1.1. Data caches

Data caches [7] are present in most computers because of the high latency of access to external memory. They store a recently-used subset of memory elements, which is a severe problem for WCET analysis as the execution time of most memory accesses is dependent on the preceding *reference string* [8]. This is the sequence of memory addresses that have been used by the program. The timing of a load or store operation depends on the relationship between its *effective address* and the effective addresses of earlier operations. The time cost of a cache miss is high due to the latency of external memory accesses (Table 1).

Conventionally, WCET analysis for data caches aims to identify where misses cannot occur, since the pessimistic assumption that all accesses result in a cache miss will result in a WCET estimate that is safe but not tight. However, WCET analysis must account for all possible data cache behaviours, even though these may be dependent on input data. WCET analysis has always accounted for data dependence *at the level of the program structure*, because the input data will select a particular path through a program [9]. Data caches force WCET analysis to account for a *lower-level* form of data dependence, since the effective address used by *any* access could affect the timing of *any* future access. A related issue is *conflict misses* [7], which occur when one *object* (memory area) competes with another for a cache line. Conflict miss behaviour is entirely dependent on the lower-order bits of the effective addresses being used.

This data dependence limits accurate (tight) WCET analysis to systems without data caches or programs in which most addresses are predictable *before execution starts* [10]. When the effective address of a load or store is data-dependent in some way, the memory access operation is said to be *dynamic* [11], and its effect on the data cache is unknown. Dynamic accesses need to be either eliminated, or identified so that the data cache can be bypassed when they execute [12]. Otherwise, they will leave the data cache in an unknown state. In this form of analysis, *a program will not benefit from the data cache* if all of its memory accesses are dynamic. A high execution time must be assumed for every dynamic load or store operation.

Time-predictable computer architectures [13]–[16] attempt to offset these issues by changing computer hardware. For

System	Direct	Cache hit	Cache miss	CPU speed
	(time in clock cycles)			(MHz)
ARM PB11MPcore [3]	79	1	97	300
StrongARM-110 [4]	17	1	24	50
PPC 405 (FX12) [5]	33	1	41	100
Microblaze (ML505) [6]	31	1	38	125

TABLE 1. Measured worst-case execution time for a load operation on four embedded systems in clock cycles, if the cache is bypassed (Direct), and in the event of a cache hit and a cache miss.

example, different classes of data can be directed to different data caches, such as a stack cache, a static (global variable) data cache, and a heap data cache [17]. This is helpful because the reference string for each class can be considered separately, and different cache hardware can be used appropriately for each class. For example, a *fully associative* cache is used for heap data [7] in order to solve the conflict miss problem. However, the high area and energy cost of the *content-addressable memory* (CAM) required for such caches [18] sets a strict limit on the size. This does not eliminate other problems caused by a data-dependent reference string, although a solution is proposed for the stack cache, where cache misses are only permitted during operations that move the stack frame (e.g. "return").

The restrictions imposed by WCET analysis for data cache analysis can be justified by observing that WCET analysis is only necessary for hard real-time tasks. These are a special case. Some hard real-time programs do not use dynamic accesses at all [19]. Others rarely use them [12]. It is doubtful that this trend will continue. Java is now a popular language for programming real-time systems [15]. Being object-oriented, it relies heavily on dynamic references created by the `new` keyword. A Java programmer would have to make a special effort to write code without dynamic references, which would negate the advantages of using Java in the first place.

1.2. Data Cache Partitioning and Locking

Cache *partitioning* [20] and cache *locking* [21] have been suggested as ways to improve the time-predictability of cache accesses. Both involve disabling the cache update mechanism for parts of the cache, reducing the number of possible cache states that must be considered during WCET analysis.

Partitioning [20] approaches divide a cache into subsections, each of which is used exclusively by a single task in a multitasking system. Partitioning improves time-predictability by preventing interference between tasks: updates and accesses are disabled for all but one cache subsection during each task. This allows any single-task data cache analysis approach to be used in a multi-tasking environment, but it does not simplify any part of that analysis. Cache partitioning is *orthogonal* to this report, which only considers a single task. A partitioning-like process could be applied to the work described in this report in order to allow it to be used with multiple tasks.

Cache locking [21] approaches disable the cache update mechanism. Cache locking can be used to handle dynamic accesses without disturbing the cache state [12], and it can also be used to ensure that particular memory access operations are cache hits, as data can be preloaded into a locked cache. However, for a non-fully associative cache and more than one object, the possibility of conflict misses must be considered *during preloading*, since one preloaded object could evict another from the cache. This limits the practicality of using locked caches. The technology is well-suited to static data (e.g. global variables) and stack data, but not dynamic references and heap data.

1.3. Scratchpads

Researchers have been considering *scratchpad* technology as an alternative to data caches for hard real-time embedded systems [22], [23]. Scratchpads are small on-chip memory areas that may be used in place of instruction or data caches [24]. Accesses to scratchpad are always as fast as data cache hits. There is no equivalent of a cache miss. Hence, scratchpads offer a time-predictable replacement for a cache. Crucially, *the behaviour of the scratchpad is independent of the reference string*. The time taken for a load or store operation can be computed *offline* (before execution) just by knowing whether its address will be in scratchpad or not, so scratchpads effectively act as perfect data caches for a known subset of the memory accesses in a program. Accesses outside the scratchpad are direct accesses to external memory: these incur a large time penalty that is similar to a cache miss (Table 1).

The advantage of scratchpads is not limited to simplification of analysis. Even if data cache WCET analysis *were* possible without the restrictions used in [10]–[12], there would be a large variance between the best case (or average) execution time and the WCET due to the possibility of conflict misses. Scratchpads do not have this problem. The memory access latency is fixed, so the best case, average, and worst case execution times can be identical.

However, scratchpads place an additional burden on the programmer, compiler, or WCET optimisation tool, because *something* must decide which *objects* will be stored in scratchpad, and which will be kept in the external memory. This decision needs to be made offline because it affects the WCET.

In principle, any item of data can be stored in the scratchpad, but only by explicitly managing the computer's resources within a program. Techniques such as *overlay programming* [25] are historical approaches for this. Overlay programming typically makes use of a procedure that copies data between one type of memory and another; `memcpy` would be suitable. Overlay programming can be time-predictable [26], but it is specific to one platform and places an extra burden on the programmer, who must state when data should be moved between the external memory and the scratchpad (or, historically, the backing store and the main store).

1.4. Automatic Scratchpad Allocation

Researchers have proposed ways to allocate scratchpad space automatically and transparently to the programmer, e.g. as a post-compilation step. Static schemes provide a single allocation map for an entire program [23], while dynamic schemes allow the allocation map to change [22], [27]. The dynamic schemes make better use of limited scratchpad space [28]; static schemes are generally only suitable for small programs. The dynamic approaches partition the program into *regions*, each with a local scratchpad memory map. Data is moved by software between scratchpad and external memory whenever execution crosses from one region to another. Region boundaries may be formed at loop entrances and exits [28] or at procedure entrances [27].

Unfortunately, all time-predictable scratchpad management schemes currently lack support for pointers. They support static and stack data including arrays and scalar variables. These are automatically allocated between scratchpad and external memory with the intention of minimising the WCET [22], [23].

Data-dependent accesses are permitted if they only reference static and stack data, since the reference string has no effect on the scratchpad contents. This is an improvement on the use of data caches, even if multiple data caches are used for different classes of data [17]. However, dynamic memory allocation is not permitted, and a pointer is only permitted if *whole-program pointer analysis* can identify *every* variable that might be addressed [22]. There are four reasons for these restrictions:

1) *Timing*. If a pointer could point to any object, it might reference the scratchpad or the external memory depending on the pointer value. In [22], restrictions ensure that every memory access operation can be guaranteed to reference either external memory or scratchpad.
2) *Invalidation*. Pointers might become invalid when a region boundary is crossed. Consider a pointer to an object in scratchpad; that pointer will become invalid when the object is unloaded, because the object has moved to a new address in external memory. In [22], whole-program pointer analysis identifies all of the variables that might be used by each memory access operation in order to avoid this.
3) *Sizing*. The objects referenced by pointers always have a known size at runtime, but sometimes it is hard to compute the size offline, e.g. if an object is dynamically allocated. The size needs to be known in order to (1) reserve scratchpad space, and (2) determine the time taken to copy an object between the scratchpad and external memory. In [22], pointers can only reference static or stack objects, each of which has a fixed size that is determined by the compiler.
4) *Aliasing*. When two different pointers contain the same value (i.e. they refer to the same object), they are said to be *aliases*. Aliases are a problem in compiler design [29] and in high performance CPU design [30], but the scratchpad case is worse than either of these well-known cases, because aliases can allow the same data item to exist in two separate scratchpad locations. Since accesses to one of these locations do not update the other, aliasing will cause incorrect behaviour. [22] relies on whole-program pointer analysis to identify aliases and ensure that all refer to the same physical location.

To the authors' knowledge, only one dynamic scratchpad management scheme has full support for pointers and heap data. Udayakumaran, Dominguez and Barua [27] propose an extension for the C runtime implemented by `malloc` along with new compiler functionality. Their version of `malloc` is able to allocate memory in either scratchpad or external memory.

Profiling is used to gather data about the number of accesses to each object. The profiler traces each object to its allocation point (the place in the code where `malloc` was called). Each allocation point is assigned a slice of the scratchpad space: the size of this slice depends on the estimated benefit of allocating scratchpad space at that point. At runtime, `malloc` allocates space from this slice of the scratchpad memory first. If more space is needed, it is allocated in external memory. This is a solution to the aliasing problem, because each object has a single location that is fixed by `malloc`.

The program is also partitioned into regions, each with a different set of objects to be loaded into scratchpad. When an object is not loaded into scratchpad, it cannot be accessed, and all pointers to it are invalid. Udayakumaran's software applies whole-program pointer analysis to the statically linked program binary in order to ensure that each object is not accessed outside the valid region. It is assumed that analysis will find every possible access to each object: unidentified accesses would introduce run-time errors.

Udayakumaran's scheme is not suitable for hard real-time tasks because its performance depends on decisions that are made during execution. It is not possible to be certain that a particular memory operation will refer to scratchpad memory, because (1) a number of different pointers could be used, and (2) any of those pointers could refer to either scratchpad or external memory.

2. Scratchpad Memory Management Unit

The *scratchpad memory management unit* (SMMU) provides a time-predictable subsystem for memory access in which every load or store operation can have a precisely known WCET regardless of the preceding reference string. Unlike previous solutions for data scratchpad management, pointers are fully supported. Pointer aliasing and pointer invalidation

```
unsigned strlen ( const char * s )
{
    unsigned i ;
    unsigned e1 = OPEN ( s , 100 , 0 ) ;
    for ( i = 0 ; *s != '\0' ; i ++, s++ ) {}
    CLOSE ( e1 ) ;
    return i ;
}
```

Figure 1. Programmer-directed usage of OPEN and CLOSE.

are solved problems. Whole-program pointer analysis is unnecessary. The timing of SMMU operations is not dependent on input data or the values of pointers.

Briefly, the SMMU allows a program to “OPEN” (*map*) and “CLOSE” (*unmap*) objects that are stored in external memory. An object that is OPENed can be accessed quickly. OPENed objects form a subset of the program’s data, and all accesses within that subset are guaranteed to be serviced as fast as cache hits. The SMMU acts as a perfect data cache for the subset. This property is retained until the objects are CLOSEd.

The SMMU is linked to both scratchpad space and external memory. The scratchpad space is initially unused. OPENing an object *reduces the latency of each access* at the cost of (1) scratchpad space and (2) copy operations that move data from external memory to scratchpad, and back again when CLOSE is called. OPEN does not change the address of the object. It causes the object to be copied to scratchpad, and then added to a table of address mappings so that subsequent accesses to the object are redirected to the scratchpad. CLOSEing is the reverse process. It is expected that OPEN and CLOSE will be added to a program by an algorithm in order to minimise the program’s WCET. This algorithm will ensure that space is always available for each OPEN operation by planning memory usage offline.

The innovative aspects of the SMMU are (1) the address remapping table and its implementation in hardware, and (2) the mechanism that ensures correct behaviour when mappings overlap, which is caused by pointer aliasing. Pointer invalidation is solved by never requiring pointers to change: an object has the same address if it is mapped or unmapped. The SMMU forms part of a basis for building time-predictable computer systems that facilitate WCET analysis but can be used with any C code.

2.1. Example: OPEN and CLOSE

OPEN and CLOSE are used to map and unmap objects in the scratchpad. A programmer can use these operations directly. In Figure 1, the C function `strlen` is modified to make use of the SMMU. The object addressed by `s` is mapped into scratchpad by the OPEN operation, and unmapped by CLOSE. In Figure 1, OPEN and CLOSE might be implemented by inline functions or preprocessor macros; they would carry out their functions by memory-mapped IO, special instructions, or via a co-processor interface.

2.2. Solution for Sizing Problem

All accesses performed by a C program can be decomposed into the form of a *base pointer* plus an *offset*. The base pointer is invariant; it points to the beginning of the object being accessed. It is the offset that changes, either because of an array subscript (e.g. `a[x]`), or because a pointer to the object is modified (e.g. `++a`). In either case, the base pointer is unchanged.

It is the base pointer that should be passed to the OPEN or CLOSE operation. Representing accesses in the form of base pointer + offset has been useful to earlier research where the base pointer value is effectively irrelevant to analysis [31]. It is very useful to be able to ignore data-dependent base pointer values during WCET analysis. The SMMU behaviour is the same *regardless of the base pointer’s value*.

However, the SMMU operations also need to know the size of the object being mapped. The offset will be bounded by the size in a correct program [32], so the size is needed: the SMMU must know how much data to copy during OPEN and CLOSE.

The object addressed by a base pointer `a` can have any size, even if `a` has not been allocated on the heap (e.g. by `malloc`). Dynamic allocation is also possible on the stack (e.g. by `alloca`), and `a` might point to one of several different static array variables. Sometimes, `a` can be traced to a particular source with complete accuracy [11], but this is not possible in general. Even if `malloc` is not used, the compiler and WCET analysis tool may not know the size of `a`.

This problem is similar to the issue of *loop bounds* within hard real-time tasks. Loop bounds are needed for WCET analysis [9] and may be obtained automatically in some cases [33], but the programmer is usually expected to specify them using code annotations. The same applies to object sizes: in general, the programmer must specify them, but in some cases they could be obtained automatically. For instance, object sizes could sometimes be derived from loop bounds, as in the case of an iteration through the elements of an array.

2.3. Solution for Pointer Aliasing Problem

The solution to the scratchpad pointer aliasing problem is to give the program a *logical* address space where pointers are never required to change. OPEN loads objects into scratchpad without changing the logical address - although the working copy of the object may move to a different *physical* location, its location from the program’s perspective is unchanged. Thus, two aliases still refer to the same place.

Logical addresses are also used in virtual memory systems [8], so the time-predictability of the solution described here must be emphasised. In a typical virtual memory system, page faults are triggered whenever the data at a logical address is not present in physical memory. In the SMMU system there are no page faults, because the data at every logical address is present in external memory or scratchpad. Memory accesses via the SMMU will always be time-predictable.

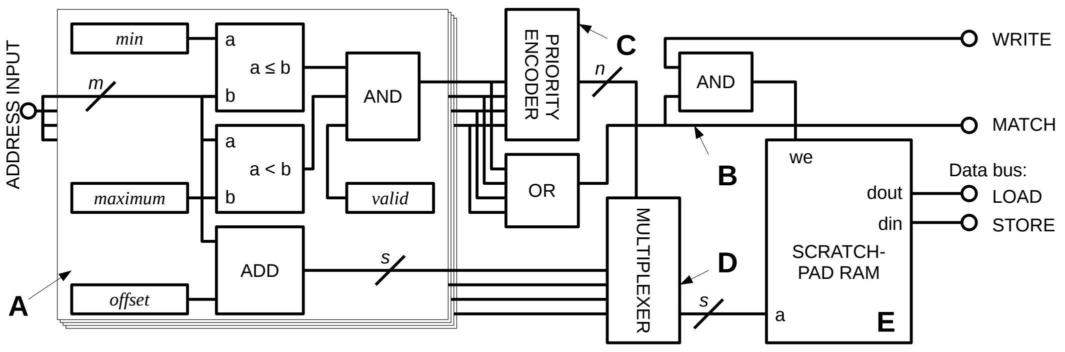


Figure 2. Scratchpad memory management unit (SMMU) hardware

2.4. Solution for Timing and Invalidation Problems

The SMMU also solves the timing and invalidation problems discussed in section 1.4. Every memory access operation can be classified offline as either "scratchpad" or "external memory". This can be done by tracing the base pointer that is being used.

If the base pointer has been OPENed, then the access *always* uses the scratchpad. Otherwise, it *may* use the external memory: it might use the scratchpad if an alias of the base pointer happens to be mapped to scratchpad. An SMMU implementation has a choice of strategy for the latter case, as it can serve the access quickly, or insist that it must always take exactly as long as it would take if external memory needed to be used. The second option might be necessary in systems where memory accesses need to be fully deterministic (e.g. [34]–[36]). This would require new opcodes for "slow" memory accesses.

The invalidation problem is solved by the use of logical addresses. Pointers are not invalidated by OPEN and CLOSE: regardless of an object's physical location, its logical address is the same.

2.5. Abstract SMMU Hardware

Figure 2 shows part of the internal structure of the SMMU. Logical addresses (m bits wide) are generated by the CPU and received on the left of Figure 2. They are compared to the SMMU table (A) which contains 2^n groups of the following registers:

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

An incoming address matches if it is between *minimum* and *maximum* in one of the valid entries. If one of the 2^n entries matches, then the match output (B) is asserted. If two or more match, then OPENed memory regions overlap. 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 scratchpad address being accessed; it is passed to the scratchpad memory (E).

2.6. OPEN and CLOSE operations

The SMMU implementation is more complex than illustrated in Figure 2, which does not show the hardware needed to implement OPEN and CLOSE. The technical details of these operations are given in Figure 5; they are complicated because they consider the data which is already in scratchpad during copy operations, assuring correct semantics when OPENed memory areas overlap.

The OPEN operation takes three parameters: a base pointer, a size, and a copy location. The object identified by the base pointer is copied from external memory into the scratchpad. The OPEN operation creates an entry in the table (Figure 2, A). To use OPEN, the program must guarantee that there is space for the copy and specify a physical location for it within the scratchpad. OPEN has a maximum time bound which can be computed as a function of the size parameter. It is *perfectly legal* to have overlapping regions of memory open at the same time: *this has no effect on timing or program correctness.*

For example, consider the program in Figure 3, which opens two overlapping areas of memory (named Y and Z) from the larger area X. Figure 4(a) shows how the expressions for `i` and `j` are implemented by the scratchpad and SMMU after the first OPEN. After the second OPEN, part of Y and Z overlap. Figure 4(b) and (c) show how this affects accesses to variables `k`, `l` and `m`. Two copies of the overlapping space exist in the scratchpad, but only one is used (Y&Z).

If Y's table reference is lower than Z, then Y takes priority over Z. Accesses to the overlapping space (e.g. `l` = Z[1]) are routed to the copy at the end of Y. The copy at the end of Z is inaccessible (Figure 4(b)). Otherwise, Z takes priority over Y. Z[1] is routed to the copy at the end of Z (Figure 4(c)).

The process that copies data in and out of scratchpad (Figure 5) respects this remapping. The second OPEN operation will take data from the scratchpad copy of Y while processing the overlapping region, so that any modification of that region before the second OPEN will be preserved.

```
void example ( int * X )
{
     int * Y ;
     int * Z ;

     Y = & X [ 4 ] ;
     OPEN ( Y , 7 , 0 ) ;
     i = Y [ 1 ] ;
     j = Y [ 6 ] ;

     Z = & X [ 9 ] ;
     OPEN ( Z , 7 , 7 ) ;
     k = Y [ 1 ] ;
     l = Z [ 1 ] ;
     m = Z [ 6 ] ;
     ...
}
```

Figure 3. Example program for Figure 4.

The copy location for each OPEN operation may be specified by the programmer. However, this value should be generated by a scratchpad allocation algorithm, which should either be executed offline and directed by WCET analysis (section 2.7) or during execution by a time-predictable algorithm (section 5.2). Scratchpad space could also be managed dynamically by virtual memory techniques, but this would not be time-predictable, so the topic is not considered further in this report.

The CLOSE operation takes one parameter: a table reference, as emitted by OPEN. It reverses the OPEN operation associated with that reference, writing the scratchpad contents back to external memory. CLOSE has a time bound that depends on the object size. CLOSE respects overlapping regions and will update them as necessary according to the priority order defined by SMMU table. For example, if Z were closed in Figure 4(c), the contents of the overlapping region Y&Z would be written back to Y.

Table references are similar to file handles. Although they define a priority order within the SMMU, they are opaque from the perspective of the program. Each OPEN operation must have a corresponding CLOSE, so the table reference returned by OPEN is stored. C code that makes use of "unstructured" programming techniques (e.g. `longjmp`, `sigaction`) must be aware of this. Both OPEN and CLOSE are thread-safe, provided that their internal operations are atomic. The issues for multithreaded programs are that (1) each thread must avoid using physical scratchpad space that could be used by other threads, and (2) there is a global limit on the total number of entries that can be OPEN at once.

2.7. SMMU/Scratchpad Allocation Algorithm

In general, manual modifications to a program (e.g. Figure 1) would be time-consuming and error-prone, requiring changes in most functions just like early schemes for overlay programming [25]. The SMMU could only be practical if OPEN and CLOSE operations were generated automatically. Existing scratchpad allocation algorithms can be extended with the necessary features. Deverge and Puaut describe a scratchpad allocation algorithm that supports static and stack data [22]. Using the SMMU, this algorithm can also support heap data referenced by pointers. The extensions that are required are:

- Using OPEN and CLOSE to transfer data between external memory and scratchpad. These replace a simple copy operation.
- Placing a limit on the total number of objects that can be in use simultaneously to match the size of the SMMU's translation table. This is an additional constraint in the integer linear program (ILP) model.
- Replacing "variables" within the algorithm with "pointers to objects". In the Deverge algorithm, "variables" refer to statically allocated objects (or objects on the stack), which are either present throughout the program or throughout a single function. In the SMMU version of the algorithm, the same notion of a "variable" would be applied to a "pointer to an object", which would come into existence when a base pointer was created, and leave existence when that base pointer was overwritten. This would destroy the 1-1 mapping between variables and objects, but the problems that this could introduce (pointer aliasing, object sizing, pointer invalidation) would be solved as described in sections 2.2-2.4.

Other parts of the algorithm would be retained, such as the assignment of objects to locations in the scratchpad, and the iterative process of WCET reduction that accounts for changes in the worst-case execution path. A subsequent paper will describe the effects of applying the modified Deverge algorithm to various programs.

3. Implementation: SMMU For Microblaze

The SMMU can only be fully evaluated using a hardware implementation. This section describes the addition of the SMMU to the Microblaze soft CPU core [37]. The SMMU replaces the Microblaze data cache. It acts as a data bus master.

3.1. Microblaze

Microblaze [38] is a 32-bit RISC-like soft CPU core sold by Xilinx as part of the Embedded Development Kit (EDK) software. It is commonly used within FPGA designs. Versions of Linux exist for Microblaze [39], and the Xilinx Platform Studio (XPS) software can be used to build entire embedded systems using Microblaze and FPGA-based peripherals.

Microblaze HDL source code is available, but only at considerable cost [40], so the SMMU can only make use of the external interfaces provided by the core. Four types of external bus interface are available, and three are wholly unsuitable because they effectively require the presence of a data cache:

- The Processor Local Bus (PLB) interface [41] is well-suited to high bandwidth links with peripherals and memory.

  PLB accesses have a high latency (16 clock cycles or more). This setup cost is too high for PLB to be useful

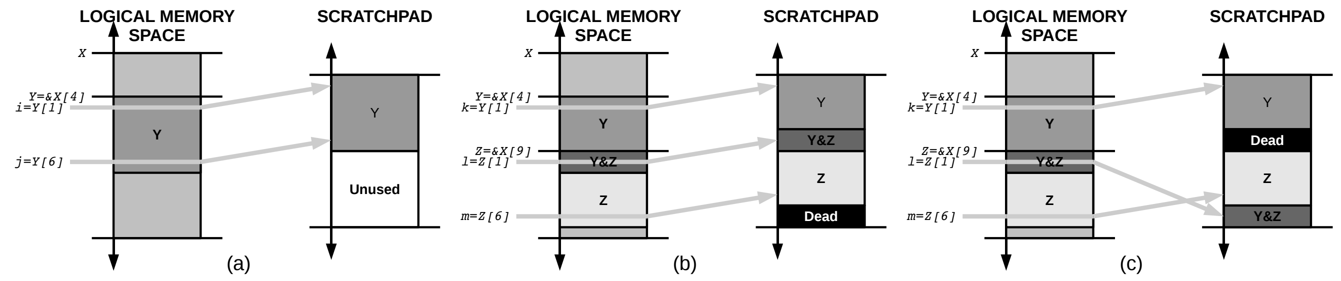


Figure 4. How accesses to overlapping OPEN objects are implemented by the SMMU and scratchpad.

for instruction or data access, because such accesses will limit the CPU speed to the bus latency. PLB serves these purposes only through the use of instruction and data caches.

- The On-chip Peripheral Bus (OPB) interface [41] is also unsuitable for the SMMU for the same reasons as PLB.
- The Xilinx Cache Link (XCL) interface [38] is designed to be used to fill instruction and data caches, and cannot be used without them. For this reason, it is unsuitable for the SMMU.

3.2. Local Memory Bus (LMB)

Fortunately, the fourth type of bus interface is suitable. This is the Local Memory Bus (LMB). The interface is shown in Table 2. Since LMB allows single-cycle access to memory, an LMB-based implementation of the SMMU is just as effective as an SMMU implementation integrated into Microblaze.

LMB is a very simple bus. An example of its use can be seen within the `lmb_bram_if_cntlr` component distributed with EDK: the VHDL source is available for this IP core. This component connects an FPGA block RAM to Microblaze for use as a scratchpad, boot ROM, or as the only memory in small systems. It responds to memory requests in a single clock cycle, i.e. during clock cycle n, Microblaze loads valid information onto its outputs, including `LMB_ABus`. It asserts `LMB_AddrStrobe` to indicate that a memory operation should proceed. `lmb_bram_if_cntlr` decodes these during clock cycle n; by the rising clock edge, it has already decided whether the address is within the range of the block RAM, and whether the write and enable inputs of the block RAM should be asserted. If the address was within range, then the `Sl_DBus` output will be valid in clock cycle $n+1$. The `Sl_Ready` output will also be asserted for clock cycle $n+1$, indicating that the transaction is complete. The CPU can begin a new access immediately; this will complete in clock cycle $n+2$.

Address decoding is performed within `lmb_bram_if_cntlr`. Additionally, Microblaze will wait indefinitely for `Sl_Ready`, *provided that* no other data bus is in use. If PLB or OPB is also present, then their timeout signals can cause LMB accesses to be cancelled. Microblaze sends a data request along all available buses simultaneously, and assumes that the LMB will respond first if able. These facts make LMB an ideal interface for the SMMU, because the SMMU can cover the entire address space and it might carry out very long transactions (such as an OPEN of hundreds of words).

3.3. Xilinx Platform Studio and EDK

`lmb_bram_if_cntlr` is a component provided in Xilinx Platform Studio (XPS) “pcores” format. This format combines VHDL or Verilog source with metadata to describe the bus interfaces and configuration parameters. It is the format used by Microblaze and all the peripherals supported by XPS.

XPS considers components at the bus level. The user specifies a bus structure using a GUI or a text file. Then, XPS generates the necessary VHDL to connect the components together. Users can build embedded systems without needing to write any VHDL.

Microblaze can be interfaced to other components using pure VHDL, but it is easiest to make the bus connections within XPS, not least because Microblaze is only one part of the system - IO peripherals (e.g. UARTs) and an external memory controller will also be required.

The simplest way to implement the SMMU is as an XPS component, with an LMB slave interface on one “side”, and a PLB master interface on the other. The LMB interface will connect to Microblaze, while the PLB interface will connect to IO peripherals and external memory. This means that the implementation task is limited to the SMMU; existing Xilinx components can be relied on to provide all other system features.

3.4. Implementation - Stage 1

The starting point for implementation is within XPS. The “Create and Import Peripheral Wizard” is able to generate template components with PLB interfaces. These components always have a slave interface, but a master interface is also required because the SMMU must be able to initiate memory transactions.

In the first stage of implementation, the slave interface provides control and status registers (Figure 6). These allow a program running on Microblaze to initiate any of the SMMU operations, which obtain parameters from the control registers,

Signal name	Width (bits)	Source	Purpose
`LMB_Clk`	1	system	Bus clock
`LMB_Rst`	1	system	Reset
`LMB_ABus`	32	CPU	Address output from CPU
`LMB_WriteDBus`	32	CPU	Data output from CPU
`LMB_AddrStrobe`	1	CPU	Strobe: CPU outputs are valid
`LMB_ReadStrobe`	1	CPU	Read operation flag
`LMB_WriteStrobe`	1	CPU	Write operation flag
`LMB_BE`	4	CPU	Byte lane enables (write only)
`Sl_DBus`	32	RAM	Data input to CPU
`Sl_Ready`	1	RAM	Strobe: Data input is valid

TABLE 2. Local memory bus (LMB) signal lines

and send results to the status registers. If the SMMU operates incorrectly, e.g. returning invalid data, the program will not crash. The program is only *driving* the SMMU through a test framework instead of using it directly.

Implementation begins with a design that is only able to LOAD and STORE (not OPEN or CLOSE). The design does not need to include a scratchpad or the SMMU table (Figure 2); it consists only of a state machine.

3.5. Testing - Stage 1

The hardware design is tested by a Microblaze program that tests functionality through device driver routines (Figure 7). At this stage, the hardware only supports LOAD and STORE operations, so the test process is typical of memory testing techniques. It reads and writes data from memory, and ensures that the LOADed data matches expectations.

3.6. Implementation - Stage 2

The next stage is to add support for OPEN and CLOSE. This is a substantial extension to the SMMU component, since the table and scratchpad must both be added.

The table is implemented as shown in Figure 2 and Figure 5. The scratchpad is implemented using four FPGA block RAMs, one for each byte lane. This gives 16kbytes of scratchpad space. Larger scratchpads would be a trivial extension provided that the new size is a power of two. Smaller scratchpads would be non-trivial because block RAMs have a fixed size.

OPEN and CLOSE are implemented as extensions to the state machine. They use the burst transfer mode of PLB to copy up to sixteen words in a single transaction; this is the maximum for PLB in the default configuration.

3.7. Testing - Stage 2

OPEN, CLOSE, LOAD and STORE can be tested by comparison with a software model of the SMMU (e.g. Figure 5). This verifies that the hardware implementation matches the software specification. The software model can also be validated at the same time, since the SMMU has well-defined behavioural semantics:

- Memory accesses go to external memory (and complete slowly) unless the logical address matches in the SMMU table, in which case they are redirected to scratchpad and complete quickly.
- The data at each logical address is the same, regardless of whether that address is OPEN or not.
- The SMMU semantics hold even if two or more mapped regions overlap, regardless of the order of OPEN and CLOSE operations.
- (Implicit) No part of the scratchpad or external memory is corrupted by SMMU activity.

The first round of tests check the SMMU behaviour by generating many different arrangements of overlapping objects taken from the basic patterns shown in Figure 8.

The test controller modifies, loads and unloads each of the objects in a random order, thousands of times for each test pattern. This exposes any edge cases which introduce data corruption. Each test pattern can be scaled for any number of objects and any size of object. In each case, memory is modified before an OPEN or CLOSE, and then read back afterwards to check that the hardware arrangement matches a software model of expectations. Errors such as data corruption in the scratchpad or external memory will manifest themselves as differences between software and hardware. Incorrect behaviour with overlapping data is also swiftly detected when objects are OPENed or CLOSEd.

Test patterns such as those shown in Figure 8 could mask some errors by limiting the conditions that are checked, so a second test phase generates OPEN, CLOSE and STORE operations entirely at random, periodically comparing the outputs of hardware and software models of the SMMU. This test uses a "shadow" copy of the logical address space to check the result of any LOAD, detecting data corruption.

For this test, the "external" and second memory sizes were 512 words, and the scratchpad memory size was set at 64 words. (These arbitrary choices keep the overall size small enough to make errors easy to detect.) Ten million test cycles were performed, each consisting of the following three phases:

1) An OPEN or CLOSE operation with valid parameters, i.e. valid copy location, address and size in the case of OPEN, and a valid reference in the case of CLOSE.
   The valid copy locations are obtained by maintaining a list of free blocks within the scratchpad similar to `malloc`. In practical programs, this approach would not be time-predictable, so allocations would need to be

```
PROCEDURE OPEN (
        base_pointer     : ADDRESS ,
        size             : WORD ,
        copy_location    : ADDRESS )
    : table_ref
VAR ref : table_ref ;
BEGIN
    ref := INVALID;
    FOR i := 1 TO num_entries
    DO
        IF NOT table[ i ].valid
        THEN
            ref := i;
        END;
    END;

    RAISE EXCEPTION IF ref = INVALID ;

    FOR i := 0 TO size − 1
    DO
        scratchpad_memory[ i + copy_location ] :=
            LOAD ( i + base_pointer ) ;
    END;

    table[ ref ].valid := TRUE;
    table[ ref ].minimum := base_pointer;
    table[ ref ].maximum := base_pointer + size;
    table[ ref ].offset := copy_location − base_pointer;
    RETURN ref;
END OPEN;

PROCEDURE CLOSE (
        ref : table_ref )
VAR data : WORD;
BEGIN
    RAISE EXCEPTION IF NOT table[ ref ].valid;

    FOR i := 0 TO table[ ref ].size − 1
    DO
        data := scratchpad_memory[ i +
                table[ ref ].offset +
                table[ ref ].minimum ];
        STORE ( i + table[ ref ].minimum ,
                        data , ref );
    END;
    table[ ref ].valid := FALSE;
END CLOSE;

PROCEDURE STORE (
        address          : ADDRESS ,
        data             : WORD ,
        search_limit     : WORD := num_entries )
VAR ref : table_ref ;
BEGIN
    ref := INVALID;
    FOR i := 1 TO search_limit
    DO
        IF ( table[ i ].valid
        AND ( table[ i ].minimum <= address )
        AND ( adddress < table[ i ].maximum ))
        THEN
            ref := i ;
        END;
    END;

    IF ref = INVALID
    THEN
        main_memory[ address ] := data ;
    ELSE
        scratchpad_memory[ address
            + table[ ref ].offset ] := data;
    END;
END STORE;
```

Figure 5. Pseudocode for SMMU operations. LOAD is omitted for space reasons; its differences from STORE are trivial.

```
#define STATUS_ACTIVE       ( 1 << 29 )
#define STATUS_TRIGGERED    ( 1 << 28 )
#define CONTROL_READ        ( 1 << 0 )
#define CONTROL_WRITE       ( 1 << 1 )
#define CONTROL_OPEN        ( 1 << 2 )
#define CONTROL_CLOSE       ( 1 << 3 )

typedef struct Reg_struct {
    unsigned    address ;
    unsigned    status ;
    unsigned    control ;
    unsigned    data ;
} Reg ;

static volatile Reg * reg = (Reg *) 0x60000000 ;
```

Figure 6. The early implementations of the SMMU provide memory-mapped control and status registers to allow programs to test the SMMU features without depending on the SMMU for data access.

```
static unsigned Await_Done ( void )
{
    unsigned x ;
    while (reg −> status & STATUS_ACTIVE) {}
    x = reg −> data ;
    reg −> control = 0 ;
    return x ;
}

unsigned HW_LOAD ( unsigned address )
{
    reg −> address = address ;
    reg −> control = CONTROL_READ ;
    return Await_Done () ;
}

void HW_STORE ( unsigned address ,
                unsigned data )
{
    reg −> address = address ;
    reg −> data = data ;
    reg −> control = CONTROL_WRITE ;
    Await_Done () ;
}
```

Figure 7. Device driver routines for LOAD and STORE operations that are carried out via the SMMU. These use the registers listed in Figure 6.

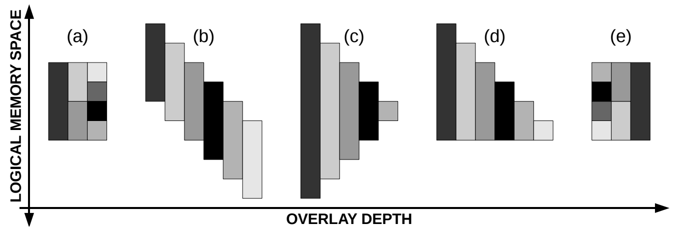


Figure 8. (a)..(e): Test patterns for overlapping objects. Each of the filled rectangles represents the memory location occupied by an object.

```
unsigned HW_OPEN ( void * base_pointer ,
            unsigned size , unsigned copy_loc )
{
    (* CONTROL) = (size << 16) | (copy_loc & 0xffff);
    return (* ((unsigned *) base_pointer )) ;
}

void HW_CLOSE ( unsigned table_ref )
{
    (* CONTROL) = 0 ;
    (* MEMORY) = table_ref ;
}
```

Figure 9. OPEN and CLOSE protocols used by the final SMMU implementation for Microblaze. Each OPEN or CLOSE operation is preceded by a write to a memory-mapped control register. OPEN operations are started by reads from memory (the address of the read is the base pointer, and the data returned is the table reference). CLOSE operations are started by writes to memory (the data is the table reference).

planned offline. The address is generated randomly. The size is also random but subject to available space.

2) 128 STORE operations to random logical addresses.
3) One LOAD operation for each logical address, checking the result returned from the SMMU with the expected value in the second memory.

These tests proved to be highly effective at detecting errors that were deliberately introduced into the SMMU model to verify the test process. These included not copying all words, copying too many words, using addresses that were "off by one" or delayed by a clock cycle, incorrect address comparisons, and providing invalid copy locations. In particular, the software implementation shown in Figure 5 passes the tests, and its behaviour matches the hardware implementation of the SMMU.

3.8. Implementation - Stage 3

The final implementation stage is a direct LMB connection between Microblaze and the SMMU. In this mode, the SMMU must handle all data accesses from Microblaze. No other data bus can be connected. If another data bus (e.g. PLB) is in use, then one of the following issues will be introduced:

- The PLB transaction initiated by Microblaze will block the PLB transaction initiated by the SMMU. This occurs if one PLB is shared between them.
- Accesses to an address X initiated by Microblaze will time out on PLB before the SMMU has been able to complete the transaction. This timeout will cancel the LMB transaction. This occurs if X cannot be serviced in the PLB memory space, because X is outside the address range of all peripherals.
- Accesses to an address X initiated by Microblaze will be completed on PLB before the SMMU has been able to complete the transaction. This completion will cancel the LMB transaction. This occurs if X can be serviced in the PLB memory space, because X is within the address range of some peripheral.

The second and third issues have a "catch 22" nature - both courses of action still lead to failure for OPEN, CLOSE and some LOAD and STORE operations. This is solved by disconnecting all data buses from Microblaze except for LMB.

An LMB interface is added to the SMMU. It replaces the control registers; the status registers are retained for testing purposes. The Microblaze CPU is able to directly carry out LOAD and STORE operations in the usual way[2], and if the effective addresses fall within a memory range that is OPEN, then they are redirected to scratchpad and serviced quickly. Otherwise, they are sent to external memory via PLB. The program running on Microblaze is also able to carry out OPEN and CLOSE operations via the protocol shown in Figure 9.

The Xilinx tools will prevent the connection of unofficial LMB hardware to Microblaze unless the `IGNORE_CUSTOM_LMB_IP_ERROR` override is set within the Microblaze metadata. The reason given by Xilinx is that LMB hardware can reduce the maximum frequency of Microblaze, which can be a problem (section 4.1).

3.9. Testing - Stage 3

The final testing stage is the replacement of the LOAD and STORE device driver routines (Figure 7) with direct accesses to memory, and the use of the OPEN/CLOSE protocol (Figure 9). This allows the test programs of section 3.7 to operate the SMMU without using memory-mapped registers for LOAD and STORE. They run exactly as an SMMU-aware program would (e.g. Figure 1). This completes implementation of the SMMU for Microblaze.

4. Evaluation: SMMU Hardware

In this section, the SMMU design for Microblaze is evaluated. The functionality was extensively tested during development (sections 3.5, 3.7 and 3.9), so this section concentrates on evaluation of the other properties of the SMMU.

4.1. Critical Path

The third stage of implementation reveals an important fact about both Microblaze and the SMMU. The address output of Microblaze is directly sourced from the *arithmetic/logic unit* (ALU) as the *critical path* information shown in Figure 10 reveals[3]. The critical path is the physical path through combinational logic that has the greatest *propagation delay*. It defines the maximum frequency of the design, because higher frequencies may lead to incorrect behaviour as logic signals fail to reach their destinations on time.

The critical path through the SMMU passes through a comparator (i.e. "less than"), a priority encoder and a multiplexer.

2. That is, using opcodes such as `lbu`, `sw`, etc.
3. The timing information in Figure 10 is estimated; the Xilinx toolset cannot produce exact timings until the place and route phase of FPGA synthesis. However, in recent versions of the Xilinx toolset, these numbers provide a reasonable guide to the final propagation delay.

```
Clock period: 9.531ns (frequency: 104.916MHz)
Delay:              9.531ns (Levels of Logic = 39)
Source:           microblaze_0/microblaze_0/Performance.Decode_I/EX_ALU_Op_0 (FF)
Destination:      scache_0/scache_0/USER_LOGIC_I/scache/lane_gen[3].lane_block.scratchpad (RAM)
Source Clock:     clock_generator_0/clock_generator_0/PLL0_CLK_OUT<0> rising
Destination Clock: clock_generator_0/clock_generator_0/PLL0_CLK_OUT<0> rising

Data Path: microblaze_0/microblaze_0/Performance.Decode_I/EX_ALU_Op_0
   to scache_0/scache_0/USER_LOGIC_I/scache/lane_gen[3].lane_block.scratchpad
                          Gate     Net
Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
---------------------------------------  ------------
FDRE:C->Q            34   0.471   0.838  microblaze_0/Performance.Decode_I/EX_ALU_Op_0
LUT4:I1->O            0   0.094   0.000  microblaze_0/Performance.Data_Flow_I/ALU_I/ex_subtract_op1
MUXCY_L:DI->LO        1   0.362   0.000  microblaze_0/,,,Use_Carry_Decoding.CarryIn_MUXCY
MUXCY_L:CI->LO        1   0.026   0.000  microblaze_0/...Not_Last_Bit.MUXCY_I
... 26 identical lines omitted ...
MUXCY_L:CI->LO        1   0.026   0.000  microblaze_0/...Not_Last_Bit.MUXCY_I
XORCY:CI->O           8   0.357   1.011  microblaze_0/...Not_Last_Bit.XOR_I
LUT5:I0->O            2   0.094   0.581  scache_0/USER_LOGIC_I/ctrl_mode_not0001111
LUT4:I2->O            3   0.094   0.491  scache_0/USER_LOGIC_I/ctrl_select_SW0
LUT6:I5->O           29   0.094   0.916  scache_0/USER_LOGIC_I/ctrl_select
LUT4:I0->O            2   0.094   1.074  scache_0/USER_LOGIC_I/lmb_we_3_and00001
LUT6:I0->O            1   0.094   1.069  scache_0/USER_LOGIC_I/scache/match196_SW3
LUT6:I0->O            4   0.094   0.352  scache_0/USER_LOGIC_I/scache/cpu_side_write_0
RAMB16:WEA0               0.624          scache_0/USER_LOGIC_I/...scratchpad
---------------------------------------
Total                     9.531ns (3.200ns logic, 6.331ns route)
                              (33.6% logic, 66.4% route)
```

Figure 10. The critical path through the SMMU hardware also passes through the Microblaze ALU, resulting in a maximum frequency of 104.9MHz on the ML505 FPGA prototyping board [6].

In stages 1 and 2 of SMMU implementation, the source of the signals is a control register and the destination is the address input of a block RAM (Figure 11(a)). When the Microblaze is introduced, *the source of the address is the ALU rather than a register* (Figure 11(b)). This means that the critical path is greatly extended, lowering the maximum frequency of the design. On the ML505 FPGA prototyping board, the new maximum frequency becomes 104.9MHz with a 16 entry table (Figure 10).

This is a serious problem. A conventional cache implementation would not face this issue because the address input could be sent directly to two block RAMs, one for the tag memory and another for the cache contents. The Microblaze manual suggests that this arrangement is used [38]. In both the conventional design and the SMMU, the decision of "hit" or "miss" does not have to be sent to the CPU until the *next* clock cycle after the request is made. However, the SMMU needs the "match"/"no match" decision on the *same* clock cycle so that the physical address can be computed. This is a disadvantage of the SMMU when combined with Microblaze, because there is no register between the ALU and the effective address output.

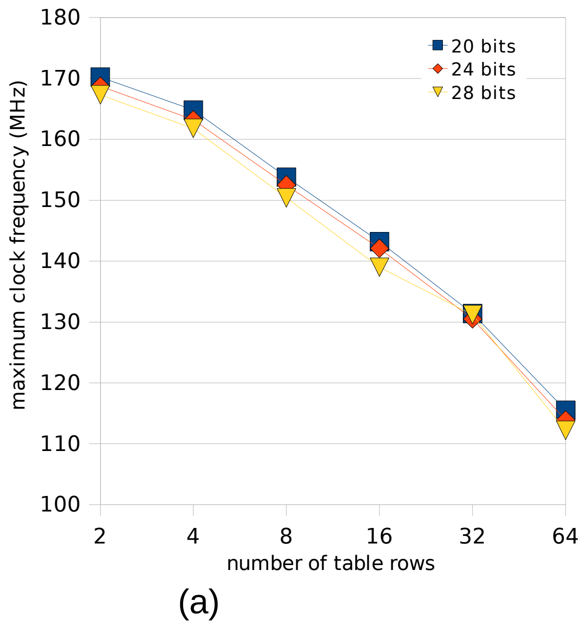
There is a simple workaround: reduce the CPU clock frequency, e.g. to 100MHz on the ML505 prototyping board. Another workaround would introduce a register between the address input and the SMMU table. This *pipelining* approach splits each memory access into two stages: (1) address computation (within the ALU), and (2) table lookup (within the SMMU). This increases memory latency by one clock cycle, making the SMMU access latency twice that of a data cache.

Eliminating the problem is a question of CPU design. In a classical RISC design, a pipeline register may be present between the EX stage (where the effective address is computed) and the MEM stage (where memory access is performed); for example, this can be seen in [42] chapter six. Microblaze does not do this because block RAMs are synchronous memory. It uses the registers within the FPGA block RAMs in place of the EX/MEM pipeline register. If it were possible to replace the MEM/WB pipeline register instead, then the SMMU could operate at full speed (Figure 11(c)). Future work could investigate this topic: in principle, the SMMU can be attached to any CPU, but some designs may be better suited than others.

Another way to reduce the length of the critical path is to simplify the SMMU design, e.g. by reducing the number of logical address bits that are actually used, or the number of table entries. The number of address bits and the number of table entries affect the maximum clock frequency as shown in Figure 12(a). 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. This delay could be reduced by additional pipelining at the cost of higher access latency.

4.2. Logic Area

The FPGA model allows some reasoning about the size of the SMMU device. Figure 12(b) shows how the size scales with the number of table entries. The size is given in *lookup tables* (LUTs), since these are the basic logic elements of a Virtex-5 FPGA. This scaling is roughly proportional to the table size. Again, fairly large tables (16, 32 entries) only occupy a small percentage of the total FPGA space (28,800



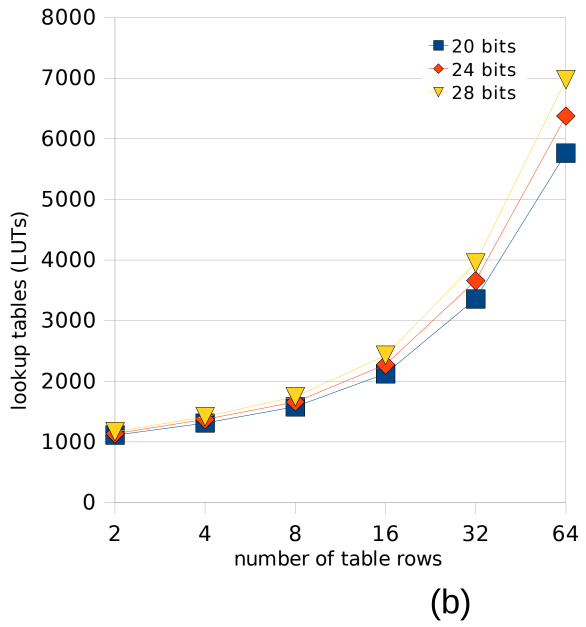


Figure 12. Characteristics of SMMU implementation when synthesised for the ML505 FPGA prototyping board [6]. (a) shows the maximum clock frequency of various versions as the number of table entries and the number of address bits is changed. and (b) shows the logic resources used.

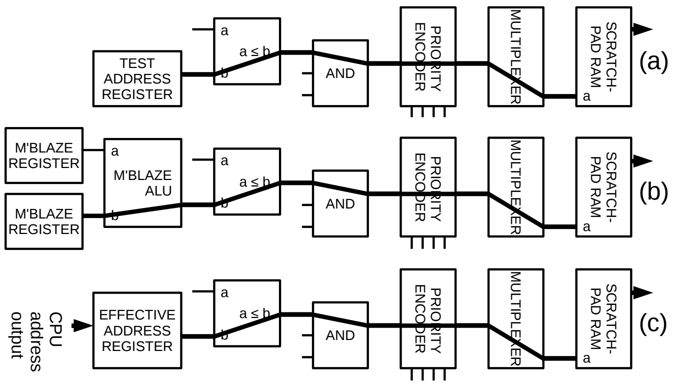


Figure 11. Critical paths: (a) the SMMU alone, (b) the SMMU and Microblaze ALU, (c) the SMMU within an ideal CPU.

LUTs). However, the scalability of the table is similar to that of a fully associative cache in that large numbers of entries are impractical. There will be speed, area and energy advantages to keeping the table size small.

The relationship between cache size and program performance is relatively well understood [7], but the relationship between SMMU table size and WCET is not clear. The size of a data cache limits the *amount of data* that can be stored, but the size of an SMMU table limits the *number of objects* that can be OPEN. The question is not about the *working set size* of a program, but about the number of objects in use at once. The design of register files (typically limited to 16 or 32 entries) suggests that the number of objects used simultaneously is usually small, justifying a 16 or 32 entry table, which has a relatively low cost. Further work will be needed to study the relationship between the number of objects and the WCET of typical programs.

4.3. Access Timings

Table 3 shows the timings for each SMMU operation, assuming default PLB settings and that the external memory is the static RAM present on the ML505 FPGA prototyping board [6]. The timings are provided by the SMMU status register (Figure 6). They demonstrate the importance of avoiding direct access to external memory where possible, because the majority of the transfer time is due to bus latency. Observe the difference between copying 8 words and copying 16 words: once a burst transaction becomes active, one word is transferred in a clock cycle. It is clearly better to copy n words in a single transaction instead of n separate transactions. The maximum PLB burst size is 16 words, so 32 and 64 word transfers require multiple transactions.

5. Evaluation: C Programming Case Study

In this section, the SMMU is applied to various C functions. The purpose is (1) to show how it could be used in a practical context, and (2) to evaluate its performance in relation to more conventional designs (e.g. a cache). As section 2.7 states, an automatic scratchpad allocation algorithm is required for large-scale use of the SMMU. However, on a small scale (e.g. single functions) manual use of OPEN and CLOSE is practical.

SMMU Op.	Params.	Clock cycles
STORE	(scratchpad)	1
STORE	(external)	19
LOAD	(scratchpad)	1
LOAD	(external)	31
OPEN	8 words	42
CLOSE	8 words	34
OPEN	16 words	50
CLOSE	16 words	42
OPEN	32 words	96
CLOSE	32 words	76
OPEN	64 words	188
CLOSE	64 words	144

TABLE 3. Timings for various SMMU operations on the ML505 prototyping board.

The study examines the reference JPEG library version 6b [43]. This is a good example of "legacy" C code. The newest parts of the code base date from 1998; some parts are at least ten years older than that. Despite this, libjpeg-6b carries out a task that is still very practical, since mobile embedded systems are often required to encode and decode JPEG images (e.g. digital cameras, embedded web browsers). The issues that turn up during the application of the SMMU to this code are likely to reappear for any C code that has not been specifically written for the SMMU or a hard real-time embedded system.

Two functions are chosen to be supported by the SMMU (with 16 table entries) and the associated scratchpad (size 16 kbytes). The relevant parts of the call tree are shown in Figure 13. The two functions account for most of the execution time of libjpeg as it decodes a reference image:

- *ycc_rgb_convert* (∼42% of execution time[4])
  Converts image data stored as luminance and colour data (YCC format) into RGB format for display on screen.
- *decompress_onepass* (∼40% of execution time)
  Carries out an inverse *discrete cosine transform* (DCT) operation to turn compressed image data into pixel data.

Importantly, both of these functions use input and output data via pointers. The JPEG library only uses globals to store constants, and these are rarely accessed. The functions are filled with data dependent accesses to the heap and stack that could not be handled efficiently by previous techniques as described in sections 1.1-1.3.

This study does not consider WCET, because the input data is a fixed reference image. However, conventional WCET analysis techniques could be applied because of the time-predictability of the CPU and memory subsystem.

5.1. Microblaze/SMMU Simulator

During the early stages of this case study, it was apparent that *manual* identification of base pointers is a non-trivial problem. The OPEN operation expects to be given the base

4. These timings are from the Microblaze/SMMU simulator with the SMMU disabled (section 5.1).

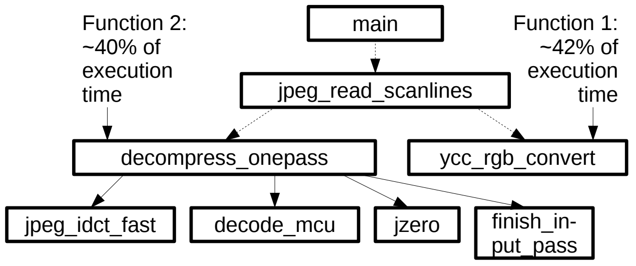


Figure 13. Part of the call tree for jpeg-6b, showing the two functions considered by the case study.

address and size of an area of memory that is about to be used. In principle, it is always possible to obtain the base address, but sufficiently complex C code may obfuscate it. Consequently, in early experiments, only a few of the base pointers were correctly identified.

One solution for this problem is analysis of source code or machine code, identifying all possible sources of an address used by a memory operation, and ensuring that an OPEN operation is correctly introduced. However, there is also a need to work out *which* objects should be OPENed, because each OPEN trades preloading time for access time and consumes scratchpad space. As the WCET is not considered in this study, a *memory access profile* is the most obvious way to do this. In general, this contains information about the memory resources used by a program. In this specific case, the memory profile contains an entry for *each instruction that creates a base pointer*. This entry has the fields listed in Figure 15. A hash table relates every instruction address (`pc`) to at most one record.

A profile (of any sort) is most easily obtained from a Microblaze simulator because of the high bandwidth required to download trace information from a real Microblaze CPU. A Microblaze-compatible simulator was refactored from the MCGREP project [37] for this purpose. It was extended with an SMMU simulator, matching the memory map of the real hardware platform considered in section 3. Within the simulator, a perfect instruction cache is assumed. Instruction execution times are as specified in the Microblaze reference manual assuming a speed-optimised design [38]. Data accesses from the SMMU take one clock cycle; data accesses from main memory take L clock cycles (set as 20, which is an optimistic figure according to Table 3).

Instructions that create base pointers can be detected by a very simple method. A program is a deterministic system; it will produce the same output given the same input, because the execution process will follow the same path through the code. The path does not change when the heap, stack and static data areas are *moved* from start address X to start address Y during the linking step. Moving the data areas from X to Y has only one effect: every pointer value changes by $Y - X$. This provides a simple way to detect any instruction that produces a pointer, and any register or stack location that contains a pointer (Figure 14). *Two* copies of the program are executed

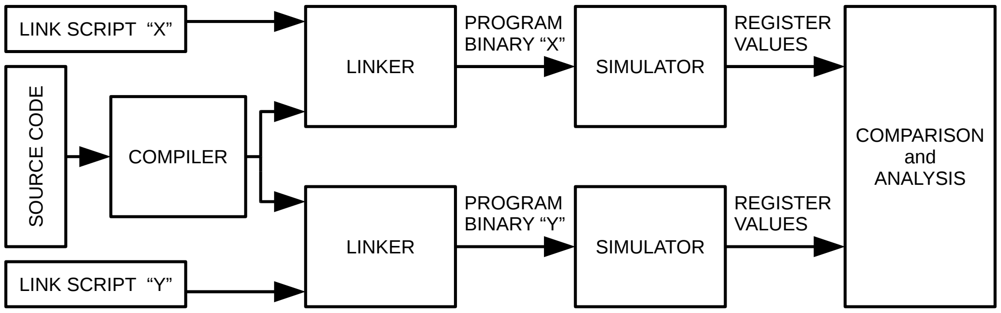


Figure 14. Two separate copies of the program are simulated at the same time. The data memory addresses are offset in the second copy, so differences in the register values reveal the creation and usage of pointers.

```
typedef struct Base_Pointer_struct
{
    /* where the base pointer is created: */
    unsigned    pc ;
    /* range of offsets used by accesses */
    int         min_offset , max_offset ;
    /* statistical data about the base pointer */
    unsigned    load_count , store_count ;
    unsigned    create_count ;
} Base_Pointer ;
```

Figure 15. The memory profiler creates a `Base_Pointer` record for each instruction that produces a base pointer.

simultaneously - differences in the output produced by any instruction indicate the presence of a pointer[5]. The program counters can be compared at each step to ensure that the two simulators are following the same path through the code.

In this way, instructions that merely *process* a pointer (e.g. adding an offset) can be distinguished from ones that *create* a pointer, e.g. from a static memory address, or by loading a value from memory. It is possible to track base pointers as they are created and used, and as they move through the register file and stack. This allows a record such as Figure 15 to be obtained for every instruction that creates a base pointer. Note that the record contains *all* the information needed by OPEN: the size of the object, and the offset of the region that will be accessed. It also indicates how often the object is accessed and how often the base pointer is created. This data is needed to decide which objects should be loaded into scratchpad.

5.2. Scratchpad Space Allocation Scheme

Each OPEN operation requires a scratchpad location to be specified for the copy. This location needs to be chosen so that the copy does not overwrite data that has been previously OPENed and is still in use. Conventional dynamic memory allocation algorithms can be used for this purpose, but this could be unwise: even if the algorithm has $O(1)$ time complexity, fragmentation could still occur within the scratchpad space, causing allocations to fail unexpectedly. In a time-predictable system, the best solution is to plan all memory allocations offline (section 2.7).

However, a simple, time-predictable and fragmentation-free algorithm does exist for memory allocation. Its disadvantage is that memory can only be deallocated in *last-in first-out* (LIFO) order, which is not optimal under all circumstances (e.g. Figure 16). This algorithm treats the scratchpad space as a stack (not to be confused with the stack used by C). There is a global copy location value. Every OPEN operation pushes a new item onto the stack, increasing the copy location by the size of the object. Every CLOSE operation pops the top item from the stack, decreasing the copy location to its previous value. Clearly, OPEN and CLOSE operations must be matched in LIFO order, and situations like the one illustrated in Figure 16 cannot be handled efficiently. This means that more scratchpad space will be required than is strictly necessary. A more advanced allocation algorithm would be able to make better use of the scratchpad resources.

The LIFO algorithm must store its data (e.g. pointers to free space) within scratchpad memory. Otherwise, the cost of allocating and freeing space will become significant. OPEN could be used for this purpose at the cost of one table entry. Alternatively, a second scratchpad could be added to the system: this option is used for the following experiments, as it has the advantage of isolating accesses carried out by memory management functions from all other sorts of access.

5.3. Function 1: ycc_rgb_convert

This section considers the jpeg-6b function that converts image data from the internal JPEG format (YCC, in which luminance and colour data are stored separately) into RGB format. The function converts one or more image rows at a time. Figure 17 shows the source code. A total of 860,000 memory accesses are carried out by ycc_rgb_convert as it processes the reference image (size 256 by 256). This is 13 memory accesses per pixel, plus a small overhead for the other memory accesses needed to set up the conversion (some items

5. This method does have the caveat that a fully deterministic memory subsystem is required (to ensure that the simulators operate in lock step). Some code *is* sensitive to the values of memory addresses, e.g. `malloc`, because alignment is important, but this can be avoided by ensuring that $Y - X$ is a large power of two.

Label	Counters			Size	Cost	Benefit	Score
	l	s	c	z	$cC(z)$	$B(l+s)$	
range_limit	196k	0	128	521	38.4k	3.74M	3.70M
outptr	256	196k	256	771	108k	3.74M	3.63M
inptr2	65.8k	0	256	259	43.0k	1.25M	1.21M
inptr0	65.8k	0	256	259	43.0k	1.25M	1.21M
inptr1	65.8k	0	256	259	43.0k	1.25M	1.21M
Crrtab	65.7k	0	128	680	48.6k	1.25M	1.20M
Crgtab	65.7k	0	128	680	48.6k	1.25M	1.20M
Cbgtab	65.7k	0	128	704	50.2k	1.25M	1.20M
Cbbtab	65.7k	0	128	704	50.2k	1.25M	1.20M
stack	4.10k	2.43k	1	1.76k	918	124k	123k
	791	0	113	112	10.8k	15.0k	4.18k
	105	0	15	112	1.44k	2.00k	555
	2	0	1	8	44	38	-6
	4	0	1	460	270	76	-194
	254	0	112	16	5.38k	4.83k	-550
	508	0	112	460	30.2k	9.65k	-20.6k

TABLE 4. Base pointers that exist in the register file or stack during execution of ycc_rgb_convert.

Loop	Accesses objects	LIFO memory allocation
1	X, A, B	X, B, A
2	X, B, C	X, B, C
3	X, A, C	X, B, C, A?
		X, A, C?

Figure 16. LIFO memory allocation only permits the removal of the topmost object in the LIFO queue. In this example, three loops are executed in sequence. Each require three objects from the set {A, B, C, X}. There is no efficient way to load object A for loop 3; either all four objects must be resident (a waste of scratchpad space) or object C must be unloaded and then reloaded (a waste of time).

are stored on the stack).

The memory profiler reveals the base pointers that are active during the function (Table 4). The table gives the total number of load (l) and store (s) operations on each base pointer, along with the number of times that base pointer is created (c). The total size (z) is given: this is the maximum offset minus the minimum offset. These are used to compute the cost C of loading the object into scratchpad whenever the base pointer is created. This cost is multiplied by 2, because the object must also be unloaded. C is defined as $C(z) = 2(L + z)$, where L is 20 (section 5.1). The data is also used to compute the benefit B, which is the time saved by loading the object into scratchpad instead of keeping it in external memory. B is defined as $B(l + s) = (L - 1)(l + s)$.

Score (the final column) is defined as $B - C$. This indicates the *approximate* value of moving the object into scratchpad. The value is approximate because the *actual* cost of the transfer will be higher than C due to (1) the overhead of managing the LIFO queue (section 5.2) and (2) the discrepancy between the true size of an object and the size measured during simulation. The table has been sorted in descending order of Score.

The contents of the table are input data (inptr1, etc.), lookup tables (Crgtab, range_limit), output data (outptr) and the stack. Other unlabelled items are minor forms of input data that

```
void ycc_rgb_convert (j_decompress_ptr cinfo,
     JSAMPIMAGE input_buf, JDIMENSION input_row,
     JSAMPARRAY output_buf, int num_rows)
{
  my_cconvert_ptr cconvert =
                 (my_cconvert_ptr) cinfo->cconvert;
  JDIMENSION num_cols = cinfo->output_width;
  register JSAMPLE * range_limit =
                 cinfo->sample_range_limit;
  register int * Crrtab = cconvert->Cr_r_tab;
  register int * Cbbtab = cconvert->Cb_b_tab;
  register INT32 * Crgtab = cconvert->Cr_g_tab;
  register INT32 * Cbgtab = cconvert->Cb_g_tab;

  while (--num_rows >= 0) {
    inptr0 = input_buf[0][input_row];
    inptr1 = input_buf[1][input_row];
    inptr2 = input_buf[2][input_row];
    input_row++;
    outptr = *output_buf++;
    for (col = 0; col < num_cols; col++) {
      y  = GETJSAMPLE(inptr0[col]);
      cb = GETJSAMPLE(inptr1[col]);
      cr = GETJSAMPLE(inptr2[col]);
      outptr[RGB_RED] =   range_limit[y + Crrtab[cr]];
      outptr[RGB_GREEN] = range_limit[y +
((int)RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr], SCALEBITS))];
      outptr[RGB_BLUE] =  range_limit[y + Cbbtab[cb]];
      outptr += RGB_PIXELSIZE;
    }
  }
}
```

Figure 17. Function 1, ycc_rgb_convert.

are accessed only as the procedure is called, e.g. cconvert. It is clear that range_limit and outptr are the best candidates for scratchpad allocation, because the cost of storing these in scratchpad is *much* smaller than the benefit of fast access. The input data and lookup tables are also good candidates. These represent the "low hanging fruit" for scratchpad allocation, enabling a 6.4× reduction in execution time in comparison to the use of external memory. This execution time reduction is just over half of the 12.2× reduction that would be expected if ycc_rgb_convert was executed with a perfect data cache. The modified source code appears in Figure 18.

A further marginal reduction (to 6.8×) is possible by storing the stack frame of ycc_rgb_convert within the scratchpad. This improvement is very small because the main cost of processing the image has already been reduced as far as possible: the 13 memory accesses per pixel all go to scratchpad. However, if scratchpad space is available, it is still worthwhile as the benefit outweighs the cost. The pointer to the current stack frame is stored in register `r1`, and the OPEN operation can obtain `r1` using inline assembly:

```
asm volatile ( "addik %0, r1, 0\n" : "=r"(sp) );
```

Together, these improvements allow 99.6% of the memory accesses in ycc_rgb_convert to be handled by the scratchpad. As Table 4 shows, attempting to OPEN any of the remainder will actually increase the execution time, because $C > B$.

More substantial improvements are possible, however. The JPEG decoder currently processes exactly one row at a time, i.e. the num_rows variable is always 1 on entry to ycc_rgb_convert. If more than one row was processed at once, the overall cost of OPENing and CLOSEing the lookup table data would be reduced. For example, processing 2 rows at a time would reduce the execution time by 8.2× (relative to the use of external memory). This change is an adjustment to a single parameter of the high-level function jpeg_read_scanlines.

This example demonstrates that making efficient use of memory can require understanding what the program is doing, then using that knowledge to reduce the number of times that OPEN and CLOSE need to be called. The necessary changes may not be trivial. Increasing the jpeg_read_scanlines parameter beyond 2 has no effect on ycc_rgb_convert, because other parts of the JPEG software are only able to produce two rows at once. Even greater improvements would require larger changes elsewhere in the software.

The 12.2× execution time reduction (for a perfect data cache) is not practical, but the program modifications described in this section are within a factor of 1.5 of that ideal case. More importantly, the scratchpad and SMMU will *always* produce the same execution time, unlike any practical data cache.

5.4. Function 2: decompress_onepass

The decompress_onepass function produces the YCC data that is processed by Figure 18. It is also a significant user of CPU time (and memory bandwidth) within the JPEG software, but scratchpad allocations are more difficult because the function is substantially larger and more complex. Four subroutines are called (Figure 13):

- *jpeg_idct_fast* (performs the inverse DCT operation)
- *decode_mcu* (decodes the raw JPEG data)
- *jzero_far* (zeroes a memory area)
- *finish_input_pass* (updates pointers after pass)

A total of 780,000 memory accesses are carried out by decompress_onepass as it operates on the reference image. A perfect data cache would reduce the execution time by 6.9× in relation to the use of external memory only (with L = 20).

```
void ycc_rgb_convert (j_decompress_ptr cinfo,
     JSAMPIMAGE input_buf, JDIMENSION input_row,
     JSAMPARRAY output_buf, int num_rows)
{
  my_cconvert_ptr cconvert =
              (my_cconvert_ptr) cinfo->cconvert;
  JDIMENSION num_cols = cinfo->output_width;
  register JSAMPLE * range_limit =
              cinfo->sample_range_limit;
  register int * Crrtab = cconvert->Cr_r_tab;
  register int * Cbbtab = cconvert->Cb_b_tab;
  register INT32 * Crgtab = cconvert->Cr_g_tab;
  register INT32 * Cbgtab = cconvert->Cb_g_tab;

  e1 = OPEN ( range_limit - 257 , 1408 , 0 ) ;
  e2 = OPEN ( Crrtab , 1020 , 1408 );
  e3 = OPEN ( Crgtab , 1020 , 1408 + 1020 );
  e4 = OPEN ( Cbgtab , 1020 , 1408 + 1020 * 2 );
  e5 = OPEN ( Cbbtab , 1020 , 1408 + 1020 * 3 );

  while (--num_rows >= 0) {
    inptr0 = input_buf[0][input_row];
    inptr1 = input_buf[1][input_row];
    inptr2 = input_buf[2][input_row];
    e6 = OPEN ( inptr0 , MAX_NUM_COLS ,
                    1408 + 1020 * 4 );
    e7 = OPEN ( inptr1 , MAX_NUM_COLS ,
                    1408 + 1020 * 4 + MAX_NUM_COLS );
    e8 = OPEN ( inptr2 , MAX_NUM_COLS ,
                1408 + 1020 * 4 + MAX_NUM_COLS * 2 );
    input_row++;
    outptr = *output_buf++;
    e9 = OPEN ( outptr , MAX_NUM_COLS * 3 ,
                1408 + 1020 * 4 + MAX_NUM_COLS * 3 );
    for (col = 0; col < num_cols; col++) {
      y  = GETJSAMPLE(inptr0[col]);
      cb = GETJSAMPLE(inptr1[col]);
      cr = GETJSAMPLE(inptr2[col]);
      outptr[RGB_RED] =   range_limit[y + Crrtab[cr]];
      outptr[RGB_GREEN] = range_limit[y +
((int)RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr], SCALEBITS))];
      outptr[RGB_BLUE] =  range_limit[y + Cbbtab[cb]];
      outptr += RGB_PIXELSIZE;
    }
    CLOSE ( e9 ) ; CLOSE ( e8 ) ;
    CLOSE ( e7 ) ; CLOSE ( e6 ) ;
  }
  CLOSE ( e5 ) ; CLOSE ( e4 ) ; CLOSE ( e3 ) ;
  CLOSE ( e2 ) ; CLOSE ( e1 ) ;
}
```

Figure 18. Function 1, ycc_rgb_convert, with OPEN and CLOSE commands. Note that the size and copy location parameters of each OPEN are given in bytes, and derived from constants (e.g. MAX_NUM_COLS) and the true sizes of the data elements rather than the range of offsets used during simulation (the JPEG library uses larger tables than strictly necessary to account for errors in the input data). The total space usage is 7kb for the 256 column reference image.

The operation of decompress_onepass is split into two phases which occur within an enclosing loop. The first phase calls decode_mcu to obtain new JPEG data (accounting for 18% of the memory accesses). The second phase calls jpeg_idct_fast to produce YCC data (accounting for 73% of the memory accesses). Although both phases share some memory elements (e.g. an input buffer), their usage of the scratchpad will be quite different.

Table 5 shows the base pointers that are active during jpeg_idct_fast. As before, these can be divided into input data, output data, lookup tables and the stack. The input data (e.g. coef->MCU_buffer), lookup tables (e.g. compptr->dct_table)

Label	Counters			Size	Cost	Benefit	Score
	l	s	c	z	$cC(z)$	$B(l+s)$	
stack	172k	159k	1	1.76k	918	6.30M	6.30M
coef->MCU_buffer	98.9k	0	1.54k	130	159k	1.88M	1.72M
output_ptr	21.2k	0	768	64	55.3k	403k	348k
IDCT_range_limit(cinfo)	64.4k	0	1.54k	1.16k	946k	1.22M	277k
compptr->dct_table	27.8k	0	1.54k	248	251k	528k	276k
cinfo	12.0k	1	113	460	30.5k	228k	198k
compptr	13.1k	0	768	84	63.0k	249k	186k
cinfo->coef	3.41k	16	16	56	1.09k	65.1k	64.0k
output_ptr	768	0	16	12	736	14.6k	13.9k
output	0	98.3k	12.3k	259	2.06M	1.87M	-196k

TABLE 5. Base pointers that exist in the register file or stack during execution of jpeg_idct_fast.

and stack can be handled as described in the previous section. It is best to OPEN these objects in decompress_onepass because they are shared by multiple calls to jpeg_idct_fast.

This covers the majority (82%) of the accesses performed by jpeg_idct_fast, making this part of decompress_onepass execute 3.5× faster than it would using external memory only. However, a perfect data cache would be 7.5× faster. The discrepancy is mostly due to the remaining 18% of the accesses. These account for 81% of the execution time spent in accessing memory: the external memory access latency is $L\times$ greater than the latency of a scratchpad access. These accesses form the bottom three rows of Table 5.

The data shows that most of the remaining external memory accesses are store operations to the output buffer. Output presents a problem because of its access pattern. jpeg_idct_ifast fills square blocks of 8 by 8 pixels, so the longest contiguous memory space is 8 bytes. This does not respond well to OPEN and CLOSE when used naïvely, as the figures for B and C demonstrate. Each access to a row of 8 pixels uses a new base pointer, so B and C assume an OPEN and a CLOSE for each row. This is much less efficient than simply writing directly to external memory, because of the overhead incurred by each OPEN and CLOSE.

Like the improvement gained in section 5.3 by processing more than one row of data at once, a change is necessary to improve the efficiency of memory usage. One solution would be to OPEN all of the memory that might be written to, but this would be a large area (the entire image). Another solution would OPEN 8 complete rows of the output buffer; decompress_onepass scans the image from left to right. However, the memory occupied by rows is also not contiguous, because the output buffer is actually a separate buffer for each row. The rows could only be contiguous if part of the JPEG software was rewritten. Therefore, the simplest solution is to OPEN all 8 rows separately.

In this arrangement, 99.7% of the accesses are handled by scratchpad. The unhandled accesses still account for 5% of the execution time, but jpeg_idct_fast now executes 5.3× faster than it would using external memory only. This is within a factor of 1.5 of the performance of a perfect data cache.

However, the overall execution time of decompress_onepass has only been reduced by 3.6×. 8.0% of the memory access instructions are handled by external memory, and 93% of these occur in decode_mcu. Table 6 shows the base pointers that are active within decode_mcu.

As in Table 5, the base pointers in Table 6 can be classed as input, output, stack or lookup table. Some of these also appear in Table 5 and are already allocated to scratchpad. Others can be allocated in decode_mcu, although next_input_byte is a little difficult due to the unbounded nature of the input data stream. When entropy and cinfo->coef are allocated to the scratchpad, only 6.6% of the memory access instructions in decompress_onepass are handled by external memory.

Further improvements require a solution to the problem posed by actbl and dctbl (bottom of Table 6). The naïve solution loads the lookup tables as soon as the base pointers are created, which results in a cost that is significantly greater than the benefit. A better solution would involve examination of the source code, which would reveal that actbl can only have only one of five values: NULL and one of four tables. The same is true for dctbl, which has another set of four tables. The tables could be preloaded at the beginning of decompress_onepass, or earlier. This is similar to the solution applied for the output buffer.

However, the solution is not easy in this case. Firstly, the tables are large (11392 bytes in total). Secondly, the cost of the OPEN operations at the start of decode_mcu is not enough to offset the cost of the subsequent load operations. This is why the execution time of decode_mcu actually *increases* by 3.1% when actbl is OPENed. The tables could be opened within decompress_onepass, and kept open throughout the image, but this would reduce the space available for all the items in Table 5 which have a greater significance on execution time. Therefore, actbl and dctbl must remain in external memory unless a larger scratchpad and larger SMMU table are available.

5.5. Summary

This case study has demonstrated that the SMMU and scratchpad can be used with a real C program. In the simulator, the scratchpad allocations discussed in sections 5.3 and 5.4 reduce the execution time of jpeg-6b by 3.0× in relation to external memory only. This has been achieved using a 16kbyte scratchpad which is only used within two functions (decompress_onepass and ycc_rgb_convert). Within those functions, the reduction is 5.0×. 8.9× would be achieved using a perfect data cache. Therefore, the performance of the SMMU is within a factor of 1.8 of a perfect data cache for those functions.

Across the whole program, a perfect data cache could achieve a 7.8× reduction in relation to external memory only. A real data cache would approach that figure in good conditions, but would be much slower than the SMMU implementation in some circumstances.

On real hardware, the SMMU implementation described in section 3 achieves a 2.7× reduction in relation to the use of external memory only. The real memory latency is higher and the real cost function C is only approximately linear (Table 3). Additionally, real hardware cannot use a perfect instruction cache, so a conventional direct-mapped instruction cache is used.

The case study demonstrates that some memory objects are much harder to allocate to scratchpad than others, because they are large or infrequently accessed (like "actbl" in Table 6) or fragmented (like "output" in Table 5). The significance of these objects increases when others are allocated to scratchpad, and they can dominate the execution time of a function, as in jpeg_idct_fast, where one SMMU allocation arrangement led to 18% of the accesses accounting for 81% of the execution time. Unfortunately, it can be difficult to find ways to allocate these objects. An understanding of the code will be needed in some cases.

The case study provides good evidence for the claim made in section 2.7: automatic scratchpad allocation is needed for programs of any practical size to avoid the time-consuming process of manually identifying objects. Automatic approaches could improve the execution time further, because variables would be allocated to scratchpad in other parts of the program. However, automatic allocation would not be able to apply the transformations used to improve access to some objects. At best, an automatic tool would simply be able to tell a programmer where optimisations are most needed.

6. Conclusion

This report has presented the scratchpad memory management unit (SMMU) as a replacement for a data cache for hard real-time systems. The SMMU can be interfaced to CPUs for embedded systems such as Microblaze (section 3), and this has been tested in an FPGA and found to have reasonable implications for clock frequency and logic area (section 4). The usage of the SMMU within a C program has also been examined through a case study (section 5). The case study used a memory profiler to identify the properties of each memory object being used by a particular function.

The efficiency of the SMMU is less than that of a perfect data cache by a factor of 1.8 for the functions considered in the case study, and by a factor of 2.7 for the whole program. Not all variables can be loaded into scratchpad due to its limited size. The scratchpad loading process also takes time, and the reduction in access time must compensate for this time. Sometimes the required data can be loaded earlier, reducing the number of OPEN operations required (as for the "output" object in Table 5). However, this increases space requirements.

The advantage of the SMMU is its time predictability combined with its low latency. It allows the latency of every load or store operation in a program to be bounded or known precisely. It is not a perfect data cache, but it behaves as one for a known subset of memory operations.

The existence of the SMMU provides a practical way to implement memory accesses with deterministic latency, which is assumed by some previous work [34], [35]. It could be combined with related approaches, such as a cache for stack data and instructions [17].

Topics for future investigation include (1) an allocation algorithm to add OPEN and CLOSE operations to a program in order to reduce the WCET, (2) the use of multiple levels of scratchpad (analogous to multi-level caching; this could allow larger scratchpads to be used), and (3) improvements to OPEN and CLOSE to handle the special cases of read-only and write-only objects.

References

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

[2] P. Puschner and A. Burns, "Guest editorial: A review of worst-case execution-time analysis," *Real-Time Syst.*, vol. 18, no. 2-3, pp. 115–128, 2000.

[3] ARM, "Platform Baseboard for ARM11 MPCore," http://www.arm.com/products/DevTools/PB11MPCore.html.

[4] Simtec, "EB110ATX (codename CATS)," http://www.simtec.co.uk/products/EB110ATX/.

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

[6] Xilinx, "ML505 User Guide," Manual UG347, 2008.

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

[8] P. J. Denning, "Virtual memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, 1970.

[9] P. Puschner and A. Schedl, "Computing maximum task execution times - a graph-based approach," *Real-Time Syst.*, vol. 13, no. 1, pp. 67–91, 1997.

[10] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *Trans. on Embedded Computing Sys.*, vol. 7, no. 1, pp. 1–38, 2007.

[11] S.-K. Kim, S. L. Min, and R. Ha, "Efficient worst case timing analysis of data caching," in *Proc. RTAS*, 1996, p. 230.

[12] T. Lundqvist and P. Stenström, "A method to improve the estimated worst-case performance of data caching," in *Proc. RTCSA*, 1999, p. 255.

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

[14] 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*, 2008, pp. 137–146.

[15] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54/1–2, pp. 265–286, 2008. [Online]. Available: http://www.jopdesign.com/doc/rtarch.pdf

[16] S. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-149, Nov 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-149.html

[17] M. Schoeberl, "Time-predictable cache organization," in *Proc. STFSSD*, March 2009.

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

Label	Counters			Size	Cost	Benefit	Score
	l	s	c	z	$cC(z)$	$B(l+s)$	
stack	170k	158k	1	1.76k	918	6.25M	6.25M
output_ptr	21.2k	0	768	64	55.3k	403k	348k
cinfo	12.0k	1	113	460	30.5k	228k	198k
compptr	13.1k	0	768	84	63.0k	249k	186k
entropy	8.45k	1.79k	256	216	37.9k	194k	156k
next_input_byte	6.15k	0	256	64	18.4k	116k	98.4k
cinfo->coef	3.66k	16	16	56	1.09k	69.9k	68.8k
output_buffer	768	0	16	12	736	14.6k	13.9k
coef->MCU_buffer	0	8.22k	1.54k	116	150k	156k	5.75k
actbl	18.1k	0	1.54k	1.42k	1.15M	344k	-807k
dctbl	3.07k	0	1.54k	1.42k	1.15M	58.4k	-1.09M

TABLE 6. Base pointers that exist in the register file or stack during execution of decode_mcu.

[19] R. Kirner and P. Puschner, “Discussion of Misconceptions about WCET,” in *Proc. WCET*, 2003, pp. 61–64.

[20] F. Mueller, “Compiler support for software-based cache partitioning,” in *Proc. LCTES*. New York, NY, USA: ACM Press, 1995, pp. 125–133.

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

[22] J.-F. Deverge and I. Puaut, “WCET-Directed Dynamic Scratchpad Memory Allocation of Data,” in *Proc. ECRTS*, 2007, pp. 179–190.

[23] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, “WCET Centric Data Allocation to Scratchpad Memory,” in *Proc. RTSS*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 223–232.

[24] 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*. New York, NY, USA: ACM Press, 2002, pp. 213–218.

[25] R. J. Pankhurst, “Operating systems: Program overlay techniques,” *Commun. ACM*, vol. 11, no. 2, pp. 119–125, 1968.

[26] I. Puaut and D. Hardy, “Predictable paging in real-time systems: A compiler approach,” in *Proc. ECRTS*, 2007, pp. 169–178.

[27] S. Udayakumaran, A. Dominguez, and R. Barua, “Dynamic allocation for scratch-pad memory using compile-time decisions,” *Trans. on Embedded Computing Sys.*, vol. 5, no. 2, pp. 472–511, 2006.

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

[29] D. R. Chase, M. Wegman, and F. K. Zadeck, “Analysis of pointers and structures,” in *Proc. PLDI*, 1990, pp. 296–310.

[30] A. Moshovos, “Exploiting load/store parallelism via memory dependence prediction,” in *Speculative Execution in High Performance Computer Architectures*. CRC Press, 2005, pp. 355–392.

[31] E. H. Gornish and A. Veidenbaum, “An integrated hardware/software data prefetching scheme for shared-memory multiprocessors,” *Int. J. Parallel Program.*, vol. 27, no. 1, pp. 35–70, 1999.

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

[33] A. Ermedahl and J. Gustafsson, “Deriving annotations for tight calculation of execution time,” *LNCS*, vol. 1300, pp. 1298–1307, 1997.

[34] P. Puschner, “Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures,” in *Proc. ECRTS*, ser. Technical Report, Jun. 2002.

[35] J. Whitham and N. Audsley, “Predictable Out-of-order Execution Using Virtual Traces,” in *Proc. RTSS*, 2008, pp. 445–455.

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

[37] J. Whitham, “SMMU Web Page,” http://www.jwhitham.org.uk/c/smmu.html.

[38] Xilinx, “Microblaze processor reference guide,” http://www.xilinx.com/bvdocs/userguides/ug081.pdf, Xilinx Corporation, Manual UG081, 2005.

[39] PetaLogix, “Linux Solutions (accessed 23 January 08),” http://www.petalogix.com/, 2007.

[40] Xilinx Corporation, “Microblaze Structural VHDL Source Code licence,” http://www.xilinx.com/ipcenter/doc/microblaze_click_core_source_license.pdf.

[41] IBM, “CoreConnect PLB4 Bus Cores,” http://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_PLB4_Bus_Cores.

[42] D. A. Patterson and J. L. Hennessy, *Computer organization & design: the hardware/software interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[43] Independent JPEG Group, http://www.ijg.org/.