Studying the Applicability of the Scratchpad Memory Management Unit

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

A combination of a scratchpad and scratchpad memory management unit (SMMU) has been proposed as a way to implement fast and time-predictable memory access operations in programs that use dynamic data structures. A memory access operation is time-predictable if its execution time is known or bounded – this is important within a hard real-time task so that the worst-case execution time (WCET) can be determined. However, the requirement for time-predictability does not remove the conventional requirement for efficiency: operations must be serviced as quickly as possible under worst-case conditions.

This paper studies the capabilities of the SMMU when applied to a number of benchmark programs. A new allocation algorithm is proposed to dynamically manage the scratchpad space. In many cases, the SMMU vastly reduces the number of accesses to dynamic data structures stored in external memory along the worst-case execution path (WCEP). Across all the benchmarks, an average of 47% of accesses are rerouted to scratchpad, with nearly 100% for some programs. In previous scratchpad-based work, time-predictability could only be assured for these operations using external memory. The paper also examines situations in which the SMMU does not perform so well, and discusses how these could be addressed.

1 Introduction

Time-predictable execution of memory access operations (*load* and *store*) is an important feature for execution of *hard real-time tasks* [4]. The *worst case execution time* (WCET) of each task partly depends on the latency of its memory accesses [18]. The use of a *scratchpad* has been proposed as a way to implement fast time-predictable memory accesses in embedded hard real-time systems [8, 16, 22, 23].

The *scratchpad memory management unit* (SMMU) extends scratchpad capabilities [27] to allow programs to store the *working set* of dynamic data structures in fast on-chip memory without the problem of *conflict misses* created by locked caches [11] nor the *pointer aliasing* and *invalidation* problems created by some dynamic scratchpad management techniques [26]. Unlike previous approaches [8, 24], the SMMU provides a program with a single logical address space. When the working set changes, data may be relocated between scratchpad and external memory; logical addresses are unchanged.

This paper characterizes the WCET improvement when the SMMU is used in addition to previous time-predictable scratchpad management techniques within a series of programs. Previous work is suitable for static data structures, i.e. global and local variables [8, 23], but not dynamic data structures. Accesses to these must use external memory, which is relatively slow. A substantial WCET reduction is possible when most accesses are routed to scratchpad.

The approach taken is as follows. First, an algorithm is described for reducing the WCET of a program by adding the SMMU operations "OPEN" and "CLOSE" at appropriate points. These copy data between external memory and scratchpad, and update a remapping table in the SMMU so that the copied objects retain the same logical address and program semantics are unchanged [27].

Secondly, the effects of the algorithm on the WCET of various benchmark programs is examined. This provides real examples of the SMMU's strengths and weaknesses, and shows the reduction in WCET that can be expected through the use of the SMMU. The algorithm is not the subject of the evaluation; it is merely a part of the mechanism by which the SMMU is evaluated. In particular, suboptimal decisions made by the algorithm will simply lead to poorer WCET reductions.

The structure of this paper is as follows. Section 2 has background detail on related work and the SMMU. Section 3 describes an allocation algorithm and section 4 describes the experimental environment used for evaluation of the SMMU. Section 5 gives the results of applying the algorithm to the benchmarks, and section 6 develops the investigation further with a discussion of how to improve the WCET in the cases where it was not greatly reduced. Section 7 concludes.

2 Background

Within a real-time system, it is often necessary to estimate the WCET of a real-time task in order to be certain that the task will meet its deadline [4]. Estimates must be *safe* (greater than the true WCET), but also *tight* (as close as possible to the true WCET) [18]. Time-predictable computer architectures aim to enable WCET estimation by specifying timing characteristics at the architectural level [2] and/or limiting the range of possible timing behaviors [21]. Time-predictable architecture research has often focused on the CPU [14, 17] or the instruction memory subsystem [16]. These issues are orthogonal to the work in this paper, which focuses on data: specifically memory access instructions.

External memory accesses have a high *latency* (execution time) in relation to the latency of other instructions [21], perhaps of the order of 100 CPU clock cycles [27]. This would limit the speed of program execution if it applied to every memory access.

The conventional solution is a data cache [11]. Data caches store recently-used data in fast on-chip memory, approximating the working set. They are not time-predictable because the latency of every memory access is dependent on the addresses used by earlier accesses (known as the *reference string* [7]). Obtaining tight but safe WCET estimates is a major challenge that has previously been addressed as follows:

1. Ensuring that only one reference string is possible for the program – accesses with non-predictable addresses bypass the cache [25]. This limits the cache benefit to static data structures, allows the exact number of cache misses to be calculated for each path through the code.

2. Allocating memory to eliminate conflict misses between objects that might be accessed simultaneously [13]. This technique uses *shape analysis* to determine which data structures may be used simultaneously [20], and uses this data to ensure that they will not conflict in cache. This provides a global bound on the number of cache misses.

3. Locally bounding the number of cache misses within loops based on the relationships between the addresses used in each iteration [19].

Bounding the number of cache misses is useful but does not provide enough information for WCET analysis of complex CPUs because of the interaction between the CPU pipeline and the cache [14]. Analysis is useful for systems that *must* use caches, but it is also valid to ask whether caches are actually a good technology for time-predictable systems as their behavior must be worked around to achieve time-predictable operation.

Scratchpads are a possible cache alternative. It is a low latency memory that is tightly coupled to the CPU [22]. Access times are independent of the reference string, simplifying WCET analysis and making scratchpads ideal for use in hard real-time systems [23]. Scratchpad allocation may be static, i.e. fixed throughout program execution [23], but *dynamic scratchpad management* techniques are more effective in general because they may keep the working set in scratchpad. This is done by copying objects at predetermined points in the program in response to execution [8, 24]. Dynamic scratchpad management requires a dynamic scratchpad allocation algorithm to decide where copy operations should be carried out.

A time-predictable dynamic scratchpad allocation algorithm has been described by Deverge and Puaut [8]. A program is divided into regions, each with a different set of objects loaded into scratchpad. It supports only static data structures, i.e. global and local variables. This restriction ensures that every program instruction can be trivially linked to the variables it might use. This is important (1) so that no attempt is made to access a variable in scratchpad while it is in external memory or vice versa, and (2) so that the access time can be accurately computed.

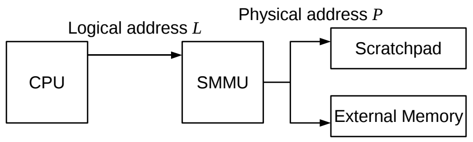

Figure 1: A computer architecture with SMMU and scratchpad.

Currently, the only dynamic scratchpad allocation algorithm that supports dynamic data structures is described by Udayakumaran, Dominguez and Barua [24]. It uses a form of shape analysis [20] to determine which instructions can access which data structures, and thus ensures that accesses to any particular object type can only occur during the regions where that object type is loaded into scratchpad. However, the technique is not time-predictable, because objects are *spilled* into external memory when insufficient scratchpad space is available.

In general, supporting dynamic data structures in scratchpads creates two problems. The first is *pointer aliasing* [1], where two or more pointers reference a single memory location: this becomes a problem when the object at that location is relocated to scratchpad, but not all of the pointers are updated to reflect the change. The remaining pointers reference a stale copy of the data. The second is *pointer invalidation* [24], where a pointer becomes invalid due to the relocation of an object (e.g. from external memory to scratchpad). The pointer is left "dangling", referring to uninitialized memory. In [24], these problems are addressed by keeping each object at a fixed physical memory location, so that pointers are never invalidated and aliases always reference the same object. This limits the usefulness of scratchpad space and impacts time-predictability whenever objects may be spilled into external memory.

2.1 The SMMU

The *scratchpad memory management unit* (SMMU) [26] is based on a simple idea: use a scratchpad to store the working set, but maintain a consistent *logical* address space as objects are moved between scratchpad and external memory. The *physical* address of an object may change, but the logical address used by the CPU and program is constant.

In this way, the SMMU combines the *address transparency* of a cache with the *time-predictability* property of a scratchpad. Each access to memory is independent of all others, there is no dependence on the reference string, and consequently the memory subsystem is extremely easy to model accurately for WCET analysis. Additionally, there is no need to perform shape analysis on the programs used, because there is no possibility of conflict between the objects mapped into scratchpad.

Figure 1 shows a simple computer architecture making use of an SMMU, scratchpad and external memory. The SMMU introduces two new operations, OPEN and CLOSE. OPEN maps an area of the logical address space to the scratchpad memory. The contents of the area are copied from external memory to scratchpad, and any future memory accesses that use the area are redirected to the scratchpad. CLOSE reverses the process. The hardware mechanism that implements OPEN and CLOSE is designed to operate correctly even if the memory areas being OPENed and CLOSEd overlap [27].

2.2 Formal SMMU Model

Let an abstract computer system include an external memory, a scratchpad and an SMMU (Figure 1). The two memories exist at different physical addresses (common practice for scratchpads [22]). Let l be a logical address, p be a physical address, and $f : l \mapsto p$ be a function that maps a logical address onto a physical address. The initial mapping f is $p = f_0(l) = l$.

OPEN takes three parameters: b_i, the lowest logical address of the area to be mapped, s_i, the size of the area, and t_i, the physical address of the destination, which is in scratchpad memory. During OPEN, the external memory range $[b_i, b_i + s_i]$ is copied to the scratchpad area $[t_i, t_i + s_i]$. After OPEN(b_i, s_i, t_i), the remapping f is:

$$p = f_i(l) = \begin{cases} l + t_i - b_i & \text{if } b_i \leq l < b_i + s_i \\ l & \text{otherwise} \end{cases} \quad (1)$$

This reroutes all accesses to memory range $[b_i, b_i + s_i]$ to the copy in the scratchpad; this is invisible to the CPU, because all of the data in that range retains the same logical address. The CLOSE operation reverses the OPEN process by copying scratchpad area $[t_i, t_i + s_i]$ to external memory and removing the remapping for i from f. CLOSE accepts a single parameter (i).

While i is OPEN, the copy of $[b_i, b_i + s_i]$ in scratchpad is considered to be more recent than the copy in external memory. More than one area of memory can be OPEN at the same time, up to a limit defined by the SMMU implementation. These areas can overlap in the logical address space. If this happens (e.g. due to pointer aliasing [1]) then more than one copy of the overlapping area exists in scratchpad and a priority ordering defines the copy that is considered most recent. This ordering is encoded in the expanded definition of f:

$$f(l) = \begin{cases} l + t_i - b_i & \text{if } \exists i, (b_i \leq l < b_i + s_i) \wedge \\ & \quad \forall j, (j \leq i \\ & \quad \quad \vee \neg[b_j \leq l < b_j + s_j]) \\ l & \text{otherwise} \end{cases} \quad (2)$$

f requires $2n$ comparisons in the worst case, where n is the maximum number of memory areas that can be OPEN simultaneously. However, these comparisons do not require $O(n)$ clock cycles as they can be performed in parallel by dedicated hardware. Adding an SMMU to a system does increase the delay for memory access, potentially reducing the maximum frequency of the CPU if memory accesses are not pipelined, but by much less than $O(n)$. The critical path is through one comparator and one multiplexer, not n [27].

2.3 When should the SMMU be used?

OPEN enables time-predictable low-latency access to any memory area $[b_i, b_i + s_i]$, which is copied to $[t_i, t_i + s_i]$ in scratchpad. There is a copying cost $C(s_i)$ for s_i words of memory. Using a bus architecture that supports *burst transactions* [16], $C(s_i) < s_i C(1)$ for $s_i > 1$; i.e. less time is required to copy s_i words in one burst than s_i words in s_i separate transactions. The definition of $C(s_i)$ is hardware-dependent, but typically includes a constant component L (the time taken to set up a burst transaction), giving an equation similar to:

$$C(n) = L + n \quad (3)$$

Values for L may vary from around 10 CPU clock cycles [16] to over 70 for some embedded systems [27]. n may be multiplied by some factor to reflect the bus width, such as $4n$ for a 128-bit bus transferring 32-bit words; this factor is omitted for simplicity.

C is used to determine if data area i should be OPENed. Let $N(i)$ be the worst-case number of accesses to memory area $[b_i, b_i + s_i]$ within a subprogram that defines i once. $B(i)$ is the time saved by OPENing i:

$$B(i) = N(i)C(1) - 2C(s_i) - N(i) \quad (4)$$

Where $N(i)C(1)$ is the total execution time related to i if it is not OPENed, $2C(s_i)$ is the cost of OPENing and CLOSEing i; and $N(i)$ is the worst-case cost of accessing i if it is in scratchpad (the latency of scratchpad accesses is 1 [27]). Clearly, if $B(i) > 0$, then OPENing is worthwhile: the execution time is reduced.

2.4 How is the SMMU used?

The two examples given in this section illustrate how OPEN and CLOSE can be used within a program.

Example 1, vector multiplication. Suppose that some code multiplies a vector (V) by a constant (K):

```
FOR Index := 0 TO 99 DO
    V[ Index ] := K * V[ Index ] ;
END;
```

V is represented by an array stored in logical address range $[b_V, b_V + s_V]$. V is accessed 200 times (100 loads, 100 stores), so $N(v) = 200$. The size $s_V = 100$.

There could be a reduction in WCET if V was copied to scratchpad. OPEN could be used to (1) copy logical address range $[b_V, b_V + s_V]$ to scratchpad physical address range $[t_V, t_V + s_V]$, and (2) remap any access $l \in [b_V, b_V + s_V]$ to $[t_V, t_V + s_V]$. This would reduce the latency of every access to V. The new code would appear as:

```
V_ref := OPEN(V, 100, 0);
FOR Index := 0 TO 99 DO
```

```
        V[ Index ] := K * V[ Index ] ;
    END;
    CLOSE( V_ref );
```

Equation 4 defines $B(\text{v})$, the WCET reduction from OPENing V. Substituting the definition of C (equation 3) into equation 4:

$$B(\text{v}) = N(\text{v})(L+1) - 2(L + s_{\text{V}}) - N(\text{v})$$
$$B(\text{v}) = (N(\text{v}) - 2)L - 2s_{\text{V}}$$
$$B(\text{v}) = 198L - 200 \qquad (5)$$

In this form, it is clear that V should be OPENed if $L > 1$; the WCET reduction will be proportional to L. Observe that this might change if s_{V} and $N(\text{v})$ had different values.

Example 2, linked list traversal. Xt is a list element:

```
    TYPE Xt = RECORD
            Data : INTEGER ;
            Next : POINTER TO Xt ;
        END ;
```

Let X be a list of Xt. Some code multiplies the Data member of each element of X by a constant K:

```
    J := X ;
    WHILE J <> NIL DO
        J.Data := K * J.Data ;
        J := J.Next ;
    END;
```

It is not possible to OPEN all of list X because X does not occupy a contiguous range of logical addresses. However, in any dynamic data structure, each element J occupies a contiguous range $[b_{\text{J}}, b_{\text{J}} + s_{\text{J}}]$, so it is possible to OPEN each element in turn:

```
    J := X ;
    WHILE J <> NIL DO
        J_ref := OPEN(J, SIZEOF(Xt), 0);
        J.Data := K * J.Data ;
        J := J.Next ;
        CLOSE( J_ref );
    END;
```

This is not as effective as opening the entire list. Suppose that s_{J} = SIZEOF(Xt) = 2 words. $N(\text{J}) = 3$, since Next is loaded and Data is loaded and stored. Adapting equation 5, $B(\text{J})$ is defined as:

$$B(\text{J}) = (N(\text{J}) - 2)L - 2s_{\text{J}} = L - 4 \qquad (6)$$

In this form, J should be OPENed if $L > 4$. OPEN and CLOSE take approximately the same time as each access: the execution time is only reduced when $N(\text{J}) > 2$.

Currently, the SMMU provides no mechanism for handling read-only, write-only and temporary objects efficiently, because of the possibility that the object might overlap a read-write object due to aliasing. This will be considered in future work.

3 Algorithm for SMMU Evaluation

The SMMU allows a program to store dynamic data structures in scratchpad (section 2.1), but this new capability cannot be applied to all data structures in all programs (section 2.3). This motivates a study of how often the SMMU is useful in real programs.

Such a study requires an algorithm to automatically add OPEN and CLOSE operations to a program. This will enable a large number of programs to be examined, and thus provide a more realistic picture of the effectiveness of the SMMU. It is not important that the algorithm be optimal; it is merely a means to the end of studying the SMMU.

This section describes the requirements for such an algorithm (section 3.1), its parameters (section 3.2) and previous work (section 3.3). Lastly, it specifies a suitable algorithm (section 3.4).

3.1 Problem Description

The “SMMU allocation algorithm” must decide where OPEN and CLOSE operations should be placed, and what their parameters should be.

The problem to be solved is a form of WCET-directed scratchpad allocation algorithm, similar to those given in [8, 16]. The scratchpad is dynamically managed; algorithms that consider only a static allocation are not suitable [23] because they could never support dynamic data structures. There is an additional constraint, as the number of objects mapped into scratchpad is limited by the size of the comparator network implementing equation 2, and there is a new difficulty, as the definition, liveness and usage of pointers must be considered on a local basis (i.e. at the function level, *not* shape analysis).

All such algorithms rely on WCET analysis to determine which data should be stored in scratchpad. WCET may be estimated using a variety of techniques [4, 18]; the chosen method is not important. Estimating the WCET reveals the *worst-case execution path* (WCEP), the path through the code that maximizes the execution time. This indicates which parts of the code are most costly.

3.2 Defining the Parameters

Let a program be represented by a *control-flow graph* (CFG) $G = (V, E)$. E is the set of all basic blocks; V is the set of vertexes linking them together.

Let P be the set of all pointers referencing dynamic data structures that are used within the program. Each $p \in P$ is a temporary reference to an object in memory occupying logical address range $[b_p, b_p + s_p]$. It is possible to distinguish between a pointer $p \in P$ referencing dynamic data and any other sort of pointer by examining the instruction that defines p. p only references a dynamic data structure if it is created by a load operation, or by arithmetic on the result of a load operation.

Define-use analysis [1] may be used to determine which basic block $e \in E$ defines a pointer $p \in P$, and to determine which basic blocks may subsequently use p. Let $U(e)$ be the multiset of $p \in P$ used by basic block e.

Liveness analysis [1] is also useful. A pointer $p \in P$ is live at basic block e if e is on a path between the definition of p and some usage of p not following any other definition. Let $I(e)$ be the subset of P that is live during e.

All of the above information can be captured from a program binary and source code. Two further types of information are needed from elsewhere. Firstly, the maximum number of iterations of each loop in the program must be bounded for WCET analysis [18]. The bounds are known as *absolute* and *relative capacity constraints*. They bound the maximum execution frequencies of basic blocks. In some cases, loop bounds can be determined automatically by analysis of the code [9], but in general they must be specified by the programmer.

Secondly, the maximum size of the data accessed via each pointer $p \in P$ must also be known. The maximum size for p is s_p. The problem of determining each s_p is similar to the issue of loop bounds. s_p is sometimes implicitly encoded by the programming language, for example if p refers to a data structure of fixed size, such as a C++ `class`, C `struct` or Ada `record` or `array`. In other circumstances, s_p may be computed automatically from the loop bounds, e.g. when a loop iterates through all elements of p. However, in the most general case, it must be specified by the programmer.

3.3 Previous Work

Scratchpad allocation algorithms are typically greedy and directed by WCET analysis [8, 16, 23]. If dynamic scratchpad management is to be used, the program is partitioned into non-overlapping regions, typically containing a single loop or function [16]. Each region has its own memory map. When program execution moves from one region to another, data may be moved between external memory and scratchpad. Creating region boundaries around loops is a good idea because the contents of a loop are likely to be executed more often than the surrounding code. Therefore, it may be worth changing the scratchpad contents to suit the code within the loop. Each point in the CFG requires a fixed set of objects to be present scratchpad: this simplifies the algorithms that are used at the cost of a lack of sensitivity to the execution context of each basic block.

Having determined such regions, scratchpad allocation algorithms operate in two phases: (1) WCET analysis is performed to find the most frequently executed code/most commonly accessed variables, then (2) the chosen code/variables are moved into scratchpad. This process repeats until scratchpad space or possible candidates are exhausted. Sometimes the WCET is estimated after every decision [23]; other algorithms carry out a fixed number of steps before re-evaluating it [16].

These ideas are sound but not entirely suitable for the "SMMU allocation algorithm" because it operates on temporary references to objects. Each p does not exist until defined by the code. Consequently, regions need to be smaller. It is good to form regions around loops [16], but for the SMMU, it is also good to break these regions down further by creating region boundaries whenever the liveness set $I(e)$ changes. This is because any change in the liveness set may be caused by the creation of a new pointer which should be OPENed, or the destruction of an old pointer which can be CLOSEd. The allocation algorithm will also be limited by the implementation of equation 2.

3.4 SMMU Allocation Algorithm

Inputs: (1) a CFG $G = (V, E)$, (2) the set of pointers P; (3) the liveness set $I(e) \subset P$; (4) the usage multiset $U(e)$; (5) relative and absolute capacity constraints for WCET estimation; (6) size information $\forall p \in P, s_p$; (7) the scratchpad size S_s; and (8) the comparator network size limit S_l.

Outputs: (1) Non-overlapping regions $R \subset E \cup V$; (2) the set of pointers that are OPEN during each region $O(R) \in P$; and (3) a scratchpad location $t_{p,R}$ for each pointer and each region.

Algorithm Step A. Partitioning: The CFG is partitioned at every *loop preheader* [16], and then at every change in the liveness set. The liveness set is deemed to change at every vertex v_1 where $\exists v_0, \exists v_2, I((v_0, v_1)) \neq I((v_1, v_2))$.

Step B. Set $\forall R, O(R) := \emptyset, \forall R, \forall p, t_{p,R} :=$ undefined.

Step C. WCET analysis. Let $c(e)$ be the execution cost of basic block e according to a conventional CPU model, which should assume that all memory accesses are serviced as quickly as possible. Let $c'(e)$ be the execution cost of e taking external memory accesses into account. $c'(e) = x(e) + y(e) + c(e)$, where $x(e)$ is the execution cost of instructions that use external memory:

$$x(e) = C(1)|p \in U(e) \wedge e \in R \wedge p \notin O(R)| \quad (7)$$

and $y(e)$ is the execution cost of OPEN and CLOSE:

$$((v_1, v_2) = e) \wedge (v_1 \in R_1) \wedge (v_2 \in R_2) \Rightarrow$$
$$y(e) = \sum_{p \in O(R_1) \neq p \in O(R_2)} C(s_p) \quad (8)$$

Then, $c'(e)$ is substituted for $c(e)$ as the cost of basic block e during WCET analysis.

Step D. Determine the operation that makes the greatest contribution to the WCET. This will be one of the terms in equation 7 or 8 multiplied by $f(e)$, the worst-case frequency of execution for basic block e.

Step E. Attempt to reduce the cost of this operation. If it is a term from equation 7, the cost is reduced by opening p within region R. An entry in the comparator network must be available: $|O(R)| < S_l$, and $t_{p,R}$ must be found such that there is no conflict with any other scratchpad space:

$$\begin{aligned} \forall p' \in O(R), \neg(t_{p',R} \leq t_{p,R} < t_{p',R} + s_{p'}) \\ \wedge\neg(t_{p',R} \leq t_{p,R} + s_p < t_{p',R} + s_{p'}) \\ \wedge(0 \leq t_{p,R} \wedge t_{p,R} + s_p \leq S_s) \end{aligned} \quad (9)$$

If it is a term from equation 8, attempt to increase the number of regions in which p is OPEN. p must retain the same location $t_{p,R}$ in any adjacent region R'; therefore, equation 9 applies. This can reduce the cost of OPEN and CLOSE operations by making them less frequent.

Step F. Re-evaluate the WCET as in step C. If the WCET has decreased relative to the best known WCET, then firmly accept all changes made to date, and go to step D. Otherwise, *tentatively* accept the changes made by step E. If the number of changes that have been tentatively accepted in this way exceeds a preset threshold, then backtracking takes place. Tentative changes are undone.

4 Experimental Environment

This section describes the benchmark programs and experimental environment used to study the SMMU. It is divided into three parts. The first justifies the selection of benchmark software (section 4.1). Section 4.2 discusses the problem of deriving loop bounds and object size data. Lastly, section 4.3 describes the results of the initial analysis of the benchmark programs.

4.1 Benchmark Selection

Most WCET-related research makes use of a set of benchmarks from the Mälardalen Real-time Technology Center (MRTC) [15], which feature the loop bound annotations required for WCET analysis [5]. However, these are not suitable for this paper because they do not use dynamic data structures.

A new set of benchmarks is needed to exercise the capabilities of the SMMU. These do not have to be hard real-time if they can provide information about the interaction between the SMMU and real code. To evaluate the SMMU, the benchmark programs need to be representative of typical programs, not specifically hard real-time programs. Therefore, benchmark suites such as Mibench [10] and SPEC CPU2000 [12] are suitable (Figure 2). The C programs included in these suites are intended to be representative of typical computational loads (SPEC) and typical embedded systems code (Mibench). They make heavy use of dynamic data structures: exactly the type of operations needed to study the SMMU.

The SPEC and Mibench programs that are not included in Figure 2 are those that do not use dynamic data structures at all (e.g. bitcount, sha) and those written in Fortran. This is because the experimental environment currently only supports C and C++ code.

4.2 Deriving Loop Bounds and Object Size Data

The disadvantage of CPU2000 and Mibench is the lack of loop or object size annotations, which prevents WCET analysis. It is not feasible to manually add such annotations to all CPU2000 and Mibench programs due to their size. One could assume that the input data is fixed, simplifying the programs to being *single-path* [17], but this is not realistic – single-path code is rare and this simplifying assumption could distort the results.

In this paper, we eliminate the need for annotations by obtaining the loop bounds and object sizes by running the programs with test data. Then, this information is applied during conventional WCET analysis using the algorithm described in [18]. This is still not entirely realistic, since it is assumed that the test data produces the maximum number of loop iterations and uses the largest possible objects. However, it is far better than the single-path assumption, because (1) conditional statements are not constrained at all, (2) loops may exit early, and (3) when an instruction may use two or more different pointers, all possibilities are considered by WCET analysis.

Benchmark	Description
adpcm^M, gsm^M, lame^M	Audio compression
ammp^F	Molecular modeling
art^F	Image recognition
basicmath^M	Math functions
bf^M, rijndael^M	Encryption algorithm
bzip2^I, gzip^I	Lossless data compression
cjpeg^M, djpeg^M	Image codec
crafty^I	Chess-playing program
crc32^M	Checksum
dijkstra^M	Shortest path algorithm
equake^F	Earthquake simulator
fft^M	Fast Fourier transform
gap^I	Group algorithms
ispell^I	Spell checker
mcf^I	Combinatorial optimization
mesa^F	Software OpenGL renderer
patricia^M	Routing table manipulation
qsort^M	Quicksort algorithm
rsynth^M	Text to speech synthesis
stringsearch^M	Finds substrings in a list
susan^M	Image edge detection
vpr^I	FPGA placement algorithm

Figure 2: Benchmark programs are taken from SPEC [12] CINT2000 (I), CFP2000 (F) and Mibench [10] (M).

4.3 Initial Investigation

The benchmarks were compiled for the Alpha instruction set [6]. Then, each was executed using the M5 simulator [3] for 100 million instructions or until completion (whichever came first). Analysis of the execution trace revealed the loop bound and object size information required for WCET analysis. It also provided the liveness and usage information for the pointers used within the program ($I(e)$ and $U(e)$). The WCET estimation model assumed an execution time cost of 1 clock cycle per instruction, except for instructions accessing external memory, which are accounted for as in equations 7 and 8. Accesses to static data structures (global and local variables) always take 1 clock cycle. Accesses to dynamic data structures take $C(1)$ cycles unless the data is in scratchpad. In that case, they also take 1 clock cycle. For the definition of C, $L = 50$ clock cycles. This access latency is close to that found on real embedded systems [27].

Figure 3 shows some of the information gathered about each benchmark after WCET analysis. The columns labeled "Proportion of WCEP" give statistics about the instructions that would be found on the WCEP, if it were to be executed. R is the proportion of instructions that only access registers, S is the proportion that only access static data structures, and D is the proportion that access dynamic data structures. This final group of instructions is the one affected by the SMMU allocation algorithm. The data shows that (on average) 70% of the instructions

Benchmark	Proportion of WCEP			WCET	
	R	S	D	Code	DE
adpcm	92%	3.1%	4.7%	47	100
ammp	72%	20%	8.1%	220	800
art	76%	7.8%	16%	140	990
basicmath	65%	34%	0.47%	230	49
bf	64%	23%	14%	610	3700
bzip2	60%	19%	21%	120	1200
cjpeg	65%	25%	9.4%	45	200
crafty	62%	24%	14%	580	3700
crc32	64%	22%	13%	620	3600
dijkstra	63%	27%	9.4%	67	280
djpeg	79%	5.3%	16%	7.8	60
fft	74%	25%	0.32%	200	30
gap	61%	20%	19%	140	1200
gsm	68%	23%	8.5%	45	180
gzip	68%	12%	20%	130	1200
ispell	70%	22%	7.2%	18	56
lame	78%	20%	1.4%	220	150
mcf	67%	29%	4.6%	120	260
mesa	73%	25%	2.3%	140	150
patricia	74%	24%	1.6%	140	96
qsort	62%	19%	20%	55	490
rijndael	69%	25%	6.7%	120	380
rsynth	65%	33%	2.2%	110	110
stringsearch	75%	24%	1.3%	0.34	0.20
susan	83%	0.57%	16%	32	250
twolf	65%	26%	9.4%	140	570
vpr	77%	18%	4.8%	130	290

Figure 3: Information about each benchmark program. Each value is rounded to 2 significant digits. Throughout this paper, execution counts, frequencies and proportions are determined along the WCEP, that is, by taking the worst-case frequency of execution for each basic block into account.

Benchmark	Prop. DS	Max OPEN	Max size	E	$\frac{\text{E}}{\text{DE}}$
adpcm	100%	2	2512	1.00	0.01
ammp	84%	2	6696	150	0.18
art	22%	1	176	780	0.79
basicmath	60%	1	40	30	0.61
bf	97%	2	1247	94	0.03
bzip2	19%	3	300	1000	0.90
cjpeg	47%	3	16176	120	0.61
crafty	12%	1	224	3300	0.89
crc32	97%	1	616	110	0.03
dijkstra	10%	2	2036	260	0.91
djpeg	80%	8	4824	16	0.26
fft	5.4%	1	16	30	0.98
gap	36%	2	672	1000	0.84
gsm	96%	2	1360	18	0.10
gzip	0.78%	1	72	1200	0.99
ispell	7.7%	1	24	55	0.99
lame	9.7%	1	64	150	0.96
mcf	55%	2	1232	170	0.66
mesa	0.03%	2	408	150	1.00
patricia	61%	2	3080	75	0.79
qsort	14%	2	1863	450	0.92
rijndael	96%	4	2304	140	0.36
rsynth	62%	2	880	43	0.38
stringsearch	97%	1	55	0.08	0.39
susan	33%	1	608	160	0.67
twolf	2.1%	1	616	560	0.98
vpr	55%	3	222	170	0.59

Figure 4: Initial results of the SMMU allocation algorithm.

in a benchmark program access only registers. The remaining 30% access either static or dynamic data structures. However, the proportion of each type of access varies widely: from programs that almost exclusively use dynamic data structures (susan) to those that almost exclusively use static data structures (fft). Clearly, the benchmarks represent many different ways of using memory.

The columns labeled "WCET" give the estimated WCET of each program when dynamic memory accesses are routed to external memory. The WCET estimate is split into two parts: the WCET of the program (Code) and the WCET of accesses to dynamic data structures in external memory (DE). (The WCET estimate for the program is the sum of both of these values.)

DE is greater than Code unless D is very much smaller than S and R. This demonstrates that the cost of dynamic data accesses can be highly significant even if only a small proportion of the instructions on the WCEP actually access dynamic data. This is because the cost of external memory accesses is so much higher than accesses to on-chip memory. Therefore, the SMMU may be worthwhile even for programs that mostly use static data structures.

5 Applying the Algorithm

To apply the "SMMU allocation algorithm" to the benchmarks, it is necessary to make three further assumptions: (1) the size of the scratchpad memory, (2) the maximum number of OPEN objects, and (3) the tolerance threshold of the algorithm (section 3.4, step F).

The comparator network that implements equation 2 limits the number of objects to a small number: in hardware, a size of 8 or 16 entries appears realistic [27]. Typical scratchpad sizes range between 1kb and 64kb [16].

The results of an initial execution of the algorithm are shown in Figure 4. A scratchpad size $S_s = 16$kb was used, along with a maximum of $S_l = 16$ OPEN objects. A tolerance threshold of 20 was chosen to produce higher quality results at the expense of increased computation time.

Figure 4 shows (from left to right) the proportion of dynamic data structure accesses on the WCEP that were routed to scratchpad (DS), the maximum number of objects that were OPEN simultaneously, and the combined size of those objects in scratchpad (given in bytes).

The WCET of accesses to external memory is given as E: this is the the combined execution cost of the accesses that could not be routed to scratchpad plus the execution cost of the OPEN and CLOSE operations (equation 3). Finally, the ratio $\frac{\text{E}}{\text{DE}}$ is shown. This shows how the WCET

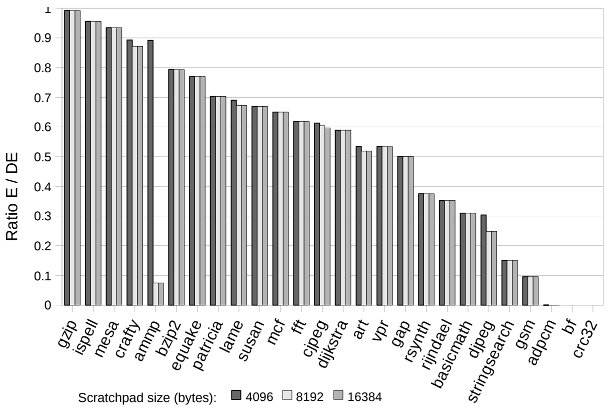

Figure 5: Ratio $\frac{\text{E}}{\text{DE}}$ using different scratchpad sizes.

of the instructions that accessed external memory was affected by the algorithm. A ratio of 1 is poor: no change has been made. A ratio close to 0 is a good result: almost all external memory access costs are eliminated.

Figure 5 gives an alternative view of Figure 4, sorted by $\frac{\text{E}}{\text{DE}}$. The bar chart shows data for scratchpads of size 4kb and 8kb as well as 16kb.

Figures 4 and 5 demonstrate that SMMU allocation has a real benefit. The column labeled E is always less than DE in Figure 3: even when the execution time of OPEN and CLOSE is taken into account, the SMMU has reduced the WCET. Sometimes, the reduction is substantial. Over 95% of all of the dynamic memory access operations in adpcm, crc32, bf, gsm, rijndael and stringsearch are routed to the scratchpad via the SMMU. The effects of accessing external memory is vastly reduced. OPEN and CLOSE still incur a cost in each case, which can be quite high (39% of DE in the case of stringsearch and 36% of DE for rijndael) but negligible for adpcm, bf and crc32.

Figures 4 also shows that the SMMU's comparator network does not need to be very large to obtain good results, as the maximum number of OPEN objects is actually 8. A full 16kb scratchpad is also unnecessary in many cases. It has a large benefit only for djpeg and ammp.

The mean value of DS is 47%. If the benchmarks are assumed to be representative of real code, then it is possible to say that on average roughly half of the dynamic memory access operations within a program can be routed to scratchpad by the SMMU. (Assuming a 16kb scratchpad, 8-entry comparator network and $L = 50$.) This is not as good as a similarly-sized data cache operating under best-case conditions, because such an arrangement may handle 99% of accesses or more. However, 47% is a worst-case. In worst-case conditions, data cache behavior is both difficult to analyze and extremely poor [26].

6 Further Investigation

Although Figures 4 and 5 include cases where the SMMU is very successful, there is also a group of programs that do not greatly benefit from the SMMU. The worst of these are ispell, gzip and mesa, where the WCET of external memory operations is reduced by less than 1%.

This section gives a study of why the SMMU is not so effective in some cases (section 6.1), and a discussion of how this could be improved (sections 6.2 and 6.3).

6.1 SMMU Limitations

Two factors reduce the effectiveness of the SMMU. Firstly, objects may be too large to be stored in the scratchpad. Figure 6 provides statistics about the sizes of the dynamic objects accessed along the WCEP in each program.

5% of the accesses use an object of size $\leq$ that given in the first column, 25% use an object of size $\leq$ the second column, and so on.

Figure 6 reveals that the lack of benefit is partly explained by very large objects which are frequently accessed, such as the 59kb objects used by mesa, the 74kb objects used by rsynth, the 1Mb objects used by qsort or the 60Mb objects used by gzip and bzip2. The bzip2 benchmark has a better $\frac{\text{E}}{\text{DE}}$ than gzip because some smaller objects are also used within dynamic data structures. These small objects can be allocated to the SMMU; however, the bulk of dynamic memory accesses still have to go to external memory.

The second problem is demonstrated by Figure 7. This provides statistics about $B(p)$, the benefit from OPENing the object referenced by p. To obtain the data in Figure 7, $B(p)$ was computed for every access to p on the WCEP. 25% of accesses to some p have $B(p) \leq$ the number in the first column, 50% have $B(p) \leq$ the number in the second column, and so on.

Figure 7 shows that $B(p) \leq 0$ occurs in real programs. The WCET will be increased by OPENing p if $B(p) \leq 0$. This happens in equake, where over 75% of P should not be OPENed, in dijkstra and lame (where over 50% should not be OPENed), and to a lesser extent in art, crafty, gap, vpr and several others. This typically occurs if a pointer is only used once per definition, as in a double dereference.

6.2 Making Objects Smaller

The "SMMU allocation algorithm" is not able to split large objects into smaller sections. For example, the gzip and bzip2 benchmarks both include a SPEC test harness which defines very large buffers (60Mb) for input and output. Accesses to these buffers only make use of small fixed-sized blocks, but because the block addresses are derived from one pointer (the beginning of the buffer), the allocation algorithm considers all 60Mb as a single object. It naïvely assumes that the whole object should be OPENed, when in fact it would be far more efficient and practical to OPEN a block-sized area for each access. In effect, this is how a cache operates: storing recently-used data rather than entire objects. However, the SMMU cannot be used in this way unless explicitly directed to do so, e.g. by a code transformation to split large objects into smaller parts.

For example, consider the following code from the rsynth benchmark:

Benchmark	s_p Is Greater Than				
	5%	25%	50%	75%	95%
adpcm	504b	504b	504b	1kb	1kb
ammp	8b	2kb	2kb	2kb	2kb
art	40b	78kb	624kb	624kb	624kb
basicmath	15b	15b	25b	25b	40b
bf	616b	616b	616b	616b	616b
bzip2	20b	349kb	58Mb	58Mb	58Mb
cjpeg	128b	2kb	6kb	15kb	15kb
crafty	8b	24b	112b	228b	121kb
crc32	616b	616b	616b	616b	616b
dijkstra	8b	8b	804b	39kb	39kb
djpeg	128b	260b	15kb	15kb	15kb
fft	15b	15b	80kb	80kb	80kb
gap	24b	40b	101b	5kb	5kb
gsm	224b	224b	680b	680b	680b
gzip	63kb	64kb	60Mb	60Mb	60Mb
ispell	16b	24b	544b	2kb	794kb
lame	8b	92b	768b	1kb	16kb
mcf	30b	616b	616b	4Mb	4Mb
mesa	59kb	59kb	59kb	59kb	59kb
patricia	15b	45b	616b	616b	751kb
qsort	616b	1Mb	1Mb	1Mb	1Mb
rijndael	16b	524b	524b	616b	616b
rsynth	48b	48b	48b	74kb	74kb
stringsearch	33b	33b	33b	33b	55b
susan	9kb	9kb	9kb	9kb	9kb
twolf	16b	23kb	23kb	23kb	23kb
vpr	31b	31b	31b	31kb	31kb

Figure 6: Values of s_p used within benchmark programs.

```
FOR I := 0 TO N DO
      P[ I ] := short2ulaw(Data[ I ]);
END;
```

P and Data could be stored in scratchpad if the loop was *tiled* into two nested loops:

```
FOR I := 0 TO N BY M DO
      P_ref := OPEN(& P[ I ], M, 0);
      Data_ref := OPEN(& Data[ I ], M, M);
      FOR J := 0 TO M − 1 DO
            P[ I + J ] := short2ulaw(Data[ I + J ]);
      END;
      CLOSE(P_ref); CLOSE(Data_ref);
END;
```

This code transformation is known as *loop tiling* and it is applied automatically by some compilers in order to improve the efficiency of a cache [28]. However, it is not suitable for all access patterns. Within bzip2's "sortIt" function, a 256kbyte variable is used by a bucket sort. Some loops act on this variable sequentially, but others have a "random" access pattern that is dependent on the data being sorted. Random access is not a problem when the entire object can be loaded into scratchpad, but when the object is too large to fit, there is no way to be sure which part of the object will be needed next. These accesses need to be directed to external memory; the only way to avoid this is to choose a different sort algorithm, which might have other disadvantages.

Benchmark	$B(p)$ Is Greater Than			
	25%	50%	75%	95%
adpcm	4.8×10^4	9.8×10^4	9.8×10^4	9.8×10^4
ammp	2.8×10^3	3.8×10^5	3.8×10^5	3.8×10^5
art	-7.0×10^0	4.9×10^6	5.4×10^6	5.4×10^6
basicmath	4.5×10^1	4.6×10^2	4.6×10^2	3.0×10^6
bf	1.7×10^9	1.7×10^9	1.7×10^9	1.7×10^9
bzip2	1.4×10^2	3.9×10^7	3.9×10^7	3.9×10^7
cjpeg	3.2×10^3	7.5×10^4	1.3×10^6	1.3×10^6
crafty	-3.0×10^0	4.2×10^1	1.9×10^2	6.7×10^3
crc32	3.4×10^9	3.4×10^9	3.4×10^9	3.4×10^9
dijkstra	-2.0×10^2	-3.0×10^0	6.0×10^3	7.6×10^4
djpeg	4.7×10^3	1.1×10^4	6.7×10^4	2.1×10^5
fft	4.5×10^1	3.1×10^6	3.1×10^6	3.1×10^6
gap	-3.0×10^0	4.6×10^1	1.9×10^3	1.0×10^8
gsm	3.6×10^3	3.9×10^7	3.9×10^7	3.9×10^7
gzip	3.3×10^7	3.8×10^7	1.3×10^8	3.3×10^8
ispell	3.0×10^1	1.6×10^2	4.6×10^4	6.4×10^6
lame	-1.1×10^2	-3.0×10^0	1.1×10^3	2.7×10^5
mcf	5.6×10^2	1.2×10^7	2.2×10^7	2.2×10^7
mesa	1.9×10^6	1.9×10^6	1.9×10^6	1.9×10^6
patricia	4.5×10^1	1.6×10^3	1.6×10^3	5.0×10^3
qsort	3.7×10^7	7.9×10^7	7.9×10^7	7.9×10^7
rijndael	2.9×10^3	7.8×10^7	1.8×10^8	1.8×10^8
rsynth	5.3×10^4	5.3×10^4	3.0×10^5	3.0×10^5
stringsearch	3.0×10^2	3.0×10^2	3.0×10^2	8.8×10^2
susan	1.1×10^7	1.1×10^7	1.1×10^7	1.1×10^7
twolf	3.5×10^4	3.5×10^4	4.2×10^4	4.2×10^4
vpr	-3.0×10^0	4.7×10^2	9.3×10^2	9.3×10^2

Figure 7: Values of $B(p)$ within benchmark programs.

6.3 Increasing the Benefit of OPEN

This section examines several cases in which $B(p) \leq 0$, and proposes solutions. The first example is found within the dijkstra benchmark, where over 50% of all $p \in P$ have $B(p) \leq 0$. This is primarily because of an operation in the "enqueue" function which finds the final element in a linked list:

```
qLast := qHead;
WHILE (qLast.qNext <> NIL) DO
      qLast := qLast.qNext;
END;
```

This is a misuse of the linked list data structure, since a pointer to the final element could simply be stored. This would improve the efficiency of the "enqueue" algorithm (constant time rather than linear time) and vastly reduce the number of dynamic memory accesses performed by the benchmark. A similar problem causes inefficiency in the ammp benchmark. The "a_number" function counts the elements of a linked list by iterating through it; using a counter variable would halve the number of dynamic memory accesses performed by ammp.

In addition to a number of cases where objects are too large, the art benchmark also contains a poorly-selected data structure. "bus" is a 2D array of floating-point numbers which would be more efficiently used if its rows and columns were transposed. This is because users of this structure iterate through the rows of the array rather than the columns, e.g.:

```
    FOR ti := 0 TO numfls-1 DO
        Y[tj].y := Y[tj].y +
            fl_layer[ti].P * bus[ti][tj];
    END;
```

Transposing rows and columns for "bus" would allow the memory range bus[tj][0] to bus[tj][numfls-1] to be OPENed outside the loop.

7 Conclusion

This paper has evaluated the effects of a combination of a scratchpad and SMMU on the dynamic memory accesses carried out by a set of benchmark programs. A scratchpad enables time-predictable memory access operations. The SMMU extends this to include the address transparency of a cache, enabling time-predictable support for dynamic data structures. Hard real-time programs can make use of the SMMU to implement time-predictable and low-latency access to dynamic data without complicating safe and tight WCET analysis.

Experiments show that most memory access operations using dynamic data structures can benefit from the SMMU and scratchpad, even with a modest scratchpad size and a low limit on the maximum number of OPEN objects. This is partly a natural consequence of program designs that are cache efficient. The SMMU approach is limited in similar cases those where data cache behavior would be less than ideal, such as memory accesses at random locations within a large object. Suboptimal use of data structures is also a significant problem in some benchmark programs; better data structure choices would vastly improve performance. However, the approach is also limited when objects are too large, meaning that code transformations similar to loop tiling should be applied by the compiler or programmer so that large objects are processed in small sections where possible. Future work should investigate this.

Acknowledgments

This work was supported by the EU ICT projects eMuCo and JEOPARD, nos. 216378 and 216682. Thanks to Rob Davis and the reviewers for their helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. 1986.

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

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

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

[5] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Syst.*, 11(2):145–171, 1996.

[6] Compaq. *The Alpha Architecture Handbook*. 1998.

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

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

[9] J. Gustaffson, A. Ermedahl, and B. Lisper. Algorithms for infeasible path calculation. In *Proc. WCET*, pages 1–6, 2006.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. IISWC*, 2001.

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

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

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

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

[15] MRTC. WCET Benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

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

[17] P. Puschner. Is wcet analysis a non-problem? – towards new software and hardware architectures. In *Proc. WCET*, 2002.

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

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

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

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

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

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

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

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

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

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

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