WCET-Aware Function-Level Dynamic Code Management on Scratchpad Memory

YOOSEONG KIM, Arizona State University
DAVID BROMAN, KTH Royal Institute of Technology
AVIRAL SHRIVASTAVA, Arizona State University

Scratchpad memory (SPM) is a promising on-chip memory choice in real-time and cyber-physical systems where timing is of the utmost importance. SPM has time-predictable characteristics since its data movement between the SPM and the main memory is entirely managed by software. One way of such management is dynamic management. In dynamic management of instruction SPMs, code blocks are dynamically copied from the main memory to the SPM at runtime by executing direct memory access (DMA) instructions. Code management techniques try to minimize the overhead of DMA operations by finding an allocation scheme that leads to efficient utilization. In this article, we present three function-level code management techniques. These techniques perform allocation at the granularity of functions, with the objective of minimizing the impact of DMA overhead to the worst-case execution time (WCET) of a given program. The first technique finds an optimal mapping of each function to a region using integer linear programming (ILP), whereas the second technique is a polynomial-time heuristic that is suboptimal. The third technique maps functions directly to SPM addresses, not using regions, which can further reduce the WCET. Based on ILP, it can also find an optimal mapping. We evaluate our techniques using the Mälardalen WCET suite, MiBench suite, and proprietary automotive applications from industry. The results show that our techniques can significantly reduce the WCET estimates compared to caches with the state-of-the-art cache analysis.

CCS Concepts: • **Computer systems organization** → **Embedded software**; **Real-time system architecture**; *Special purpose systems*; • **Software and its engineering** → *Compilers*; Software post-development issues;

Additional Key Words and Phrases: Scratchpad memory (SPM), worst-case execution time (WCET), hard real-time systems, code management

ACM Reference Format:
Yooseong Kim, David Broman, and Aviral Shrivastava. 2017. WCET-aware function-level dynamic code management on scratchpad memory. ACM Trans. Embed. Comput. Syst. 16, 4, Article 112 (May 2017), 26 pages.
DOI: http://dx.doi.org/10.1145/3063383

This work is an extended version of a paper, ©2014 IEEE. Reprinted, with permission, from Y. Kim et al., “WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores,” in Proceedings of Real-Time and Embedded Technology and Applications Symposium. The main differences are refined definitions and analyses (Section 3), region-free mapping (Section 4.3), and more extensive evaluation (Section 5).

This work was supported in part by the Swedish Research Council (#623-2013-8591, #623-2011-955), the National Science Foundation (CCF 1055094 (CAREER) and CNS 1525855), and the National Institute of Standards and Technology (60NANB16D305).

Authors' addresses: Y. Kim and A. Shrivastava, Arizona State University; emails: {yooseong.kim, aviral.shrivastava}@asu.edu; D. Broman, School of Information and Communication Technology, KTH Royal Institute of Technology, Sweden; email: dbro@kth.se.

DOI: http://dx.doi.org/10.1145/3063383

1. INTRODUCTION

In real-time [Buttazzo 2011] and cyber-physical [Lee 2008] systems, timing is a correctness criterion, not just a performance factor. Execution of program tasks must be completed within certain timing constraints, often referred to as *deadlines*. When real-time systems are used in safety-critical applications, such as automobiles or aircraft, missing a deadline can cause devastating, life-threatening consequences.

To ensure the absence of missed deadlines, safe upper bounds of tasks' *worst-case execution times* (WCETs) must be calculated by WCET analysis [Wilhelm et al. 2008]. The tightness and the timeliness of the analysis results are largely affected by the time predictability of a given system. Traditional approaches for improving the average-case performance (e.g., caches and speculative execution) are often disastrous to time predictability [Axer et al. 2014].

Scratchpad memories (SPMs) are a promising alternative to caches in real-time and cyber-physical systems, for their time-predictable characteristics. SPM is a raw memory controlled only by executing explicit direct memory access (DMA) instructions. Compared to caches whose data movement is controlled implicitly by the addresses of memory accesses and a given replacement policy, SPMs are more time predictable thanks to the explicit management [Suhendra et al. 2005; Liu et al. 2012].

In this article, we present techniques that allocate code blocks to SPM, called code management techniques, and the objective of the techniques is to reduce the WCET of a given program. There are several previous SPM management techniques that focus on reducing WCET by allocating either program variables [Suhendra et al. 2005; Deverge and Puaut 2007; Wan et al. 2012] or code blocks [Falk and Kleinsorge 2009; Prakash and Patel 2012; Puaut and Pais 2007; Wu et al. 2010] to the SPM. As code management techniques, our techniques are related to the latter group of work but have these major differences.

First, our techniques perform dynamic management in which the SPM contents are updated at runtime by executing DMA instructions in order to exploit the locality of large applications. Many previous code management techniques are static [Falk and Kleinsorge 2009; Prakash and Patel 2012; Suhendra et al. 2010], where selected instructions are loaded into the SPM only at loading time and not at runtime.

The second difference is at the granularity of management. Code management techniques perform management at various levels of granularity, such as basic blocks [Steinke et al. 2002a; Janapsatya et al. 2006; Puaut and Pais 2007; Wu et al. 2010], groups of basic blocks on a straight-line path [Verma et al. 2004], or fixed-size pages [Egger et al. 2006]. In this article, we focus on function-level code management techniques [Baker et al. 2010; Pabalkar et al. 2008; Jung et al. 2010; Bai et al. 2013], which load code blocks at the granularity of functions. Also, in function-level management, instructions are always fetched from the SPM, not the main memory. Other management schemes, on the other hand, allocate only part of the instructions to the SPM and leave the rest in the main memory. The accesses for the instructions left in the main memory are assumed to be uncached and slow or to be cached, which can be less time predictable.

All previous function-level code management techniques aim to optimize the *average-case execution time* (ACET) by reducing overall DMA operation overhead, but none of them considers WCET. We present the first and only function-level code management techniques that optimize the WCET of a given program. Also, all previous techniques use *function-to-region mappings* [Pabalkar et al. 2008] to allocate SPM space to functions. Our techniques can not only find an optimal function-to-region mapping for WCET but also find an optimal *region-free mapping* that maps functions directly to SPM addresses, not regions, which can lead to a lower WCET than the optimal function-to-region mapping. We evaluate our approach using several benchmarks from the Mälardalen suite [Gustafsson et al. 2010], MiBench suite [Guthaus et al. 2001], and proprietary automotive control applications from industry. The results show that our techniques can effectively reduce the WCET. The following are our major contributions:

—**WCET analysis:** We present algorithms together with an integer linear programming (ILP) formulation for computing a safe upper bound of the WCET of a program for a given SPM space allocation to functions (Section 3).

—**Optimal function-to-region mapping:** We present an ILP formulation that finds an optimal function-to-region mapping for a given SPM size (Section 4.1).

—**A heuristic for function-to-region mapping:** We present a polynomial-time heuristic algorithm for finding a suboptimal function-to-region mapping in a more scalable way than ILP (Section 4.2).

—**Optimal region-free mapping:** We present an ILP formulation that finds an optimal region-free mapping of functions to SPM addresses for a given SPM size, without using the notion of regions (Section 4.3).

—**Extensive evaluation:** We evaluate our approach in comparison with three previous function-level techniques and also with four-way set-associative caches using the state-of-the-art cache analysis technique [Cullmann 2013] (Section 5).

2. BACKGROUND AND MOTIVATION

In this section, we briefly introduce how function-level dynamic code management works. Then, we use a simple motivating example to demonstrate the difference between the mapping optimized for ACET and the mapping optimized for WCET. Lastly, we show another motivating example to explain the benefit of mapping functions directly to addresses, instead of regions.

2.1. Function-Level Dynamic Code Management

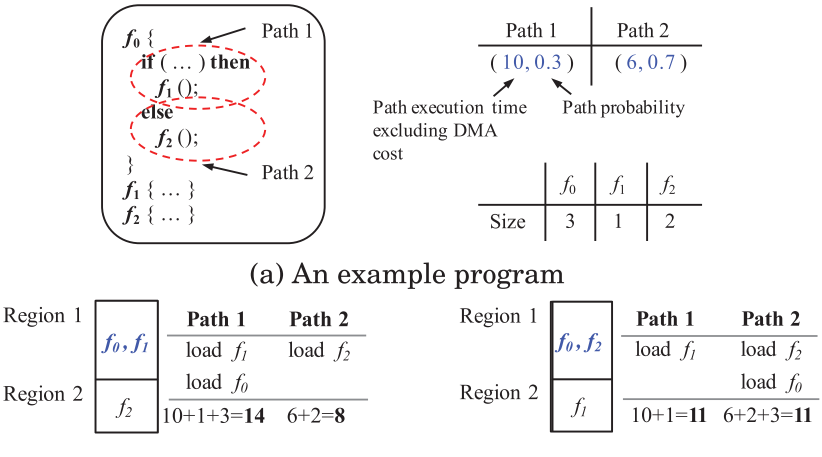
Function-level code management [Pabalkar et al. 2008] loads instructions at the granularity of functions around each call site. Since it is assumed that the core fetches instructions only from the SPM, a whole function must be loaded in the SPM before executing the function.[1] Where to load each function is decided at compile time, and in all previous approaches, such decisions are represented by function-to-region mappings. A function-to-region mapping is a surjective map from all functions in the program to all regions in the SPM.

Code management using function-to-region mappings is analogous to a direct-mapped cache. A region corresponds to a cache line. As memory addresses are mapped to cache lines, functions are mapped to regions. A function is always loaded to the starting address of its region, so loading a function always replaces any previously loaded function in the region. At a call (return), the compiler-inserted code looks up the state of the region to check if the callee (caller) function is loaded in the region. If not, the function is loaded by a DMA operation, and the core waits until it finishes before proceeding to execute the function. This process is analogous to tag comparison and cache miss handling in caches.

2.2. Why Do We Need a New Technique for Optimizing WCET?

Figure 1(a) shows our example program with three functions: f_0, f_1, and f_2. The main function f_0 has two paths, calling functions f_1 on Path 1 and f_2 on Path 2. The probability of the program to take each path is determined by the branch probability

[1] This imposes a limitation that the largest function in a program must fit in the SPM in order to be executable using function-level code management techniques.



(a) An example program

(b) Mapping A

(c) Mapping B

	Mapping size	ACET	WCET
A	$max(3,1) + 2 = 5$	$14 * 0.3 + 8 * 0.7 = 9.8$	$max(14,8) = 14$
B	$max(3,2) + 1 = 4$	$11 * 0.3 + 11 * 0.7 = 11$	$max(11,11) = 11$

(d) ACET and WCET comparison

Fig. 1. A mapping that is good for the average case is not necessarily good for the worst case. ©IEEE.

of the if-statement in f_0. The execution times of each path excluding the waiting time for DMA operations and path probabilities are also shown in the figure. The cost for loading each function is assumed to be the same as the size of the function.

Let us assume the size of the SPM is 5. Since the sum of all function sizes is larger than the SPM size, not all functions can have a private region. Here, we consider two feasible mapping solutions: mapping f_0 and f_1 to the same region (mapping A) and mapping f_0 and f_2 to the same region (mapping B). Figures 1(b) and 1(c) compare the sequence of DMA operations on each path for each mapping choice. For instance, with mapping A, f_0 must be loaded again when f_1 returns because f_0 was evicted by f_1.

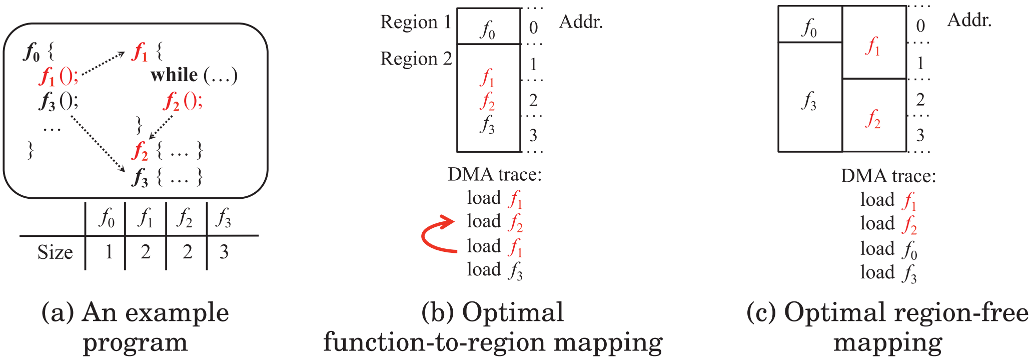
Figure 1(d) shows the ACET and the WCET for each mapping. Considering path probabilities, mapping A achieves a better ACET than mapping B. The overall amount of DMA transfers is less with mapping A because it can avoid evicting the largest function, f_0, on the more frequently executed path, Path 2. The WCET of the program is, however, better with mapping B.[2]

This example shows that optimizing for the ACET may not always result in a good WCET. Previous mapping techniques only try to optimize the ACET and are therefore not suitable for systems with timing constraints. In this article, we present mapping techniques that explicitly optimize the WCET of a program.

2.3. Why Do We Need Region-Free Mapping?

Figure 2(a) shows a motivating example with four functions, f_0, f_1, f_2, and f_4. f_0 first calls f_1, and then f_1 calls f_2 in a loop. After f_1 returns, f_0 calls f_3. The execution sequence of the functions is $f_0 f_1 (f_2 f_1)^n f_0 f_3 f_0$, where n is the number of iterations of the loop in which f_2 is called. We assume f_0 is preloaded before execution.

[2] In fact, the best mapping for both the ACET and the WCET would be mapping f_1 and f_2 into the same region and leaving f_0 in a private region. Here, we only consider mapping A and B for illustrative purposes.



(a) An example program

(b) Optimal function-to-region mapping

(c) Optimal region-free mapping

Fig. 2. Even with the optimal function-to-region mapping, f_1 and f_2 replace each other repetitively in the loop. Region-free mapping load them to disjoint address ranges, keeping them loaded after initial loadings.

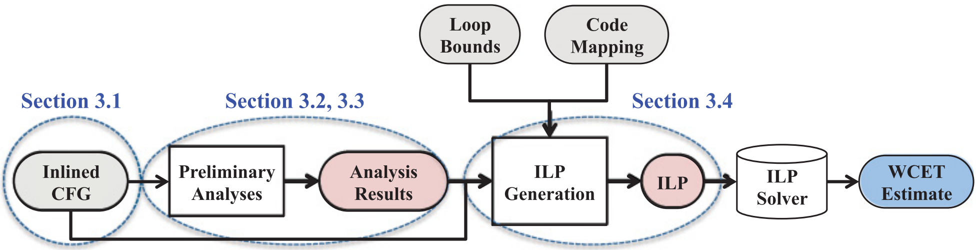


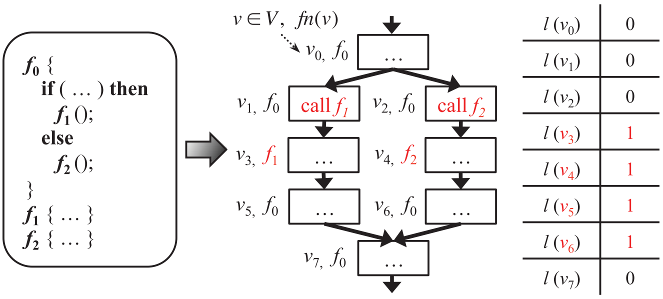
Fig. 3. An overview of our WCET analysis.

Let us assume the SPM size is 4. When a function-to-region mapping is used, it is not possible to assign separate regions to f_1 and f_2. This is because the size of the largest function, f_3, is 3, so at least one region has to be as large as 3. The remaining SPM space is only 1, and the only function that can fit in a region whose size is 1 is f_0. Thus, the optimal function-to-region mapping, shown in Figure 2(b), is to map f_0 in one region of size 1, and all the rest to the other region of size 3. With this mapping, f_0 is kept loaded in a separate region, so it is not reloaded again when other functions return. This mapping, however, causes f_1 and f_2 to replace each other repetitively in the loop, causing DMA operations in every iteration. This is a significant overhead and can increase the WCET greatly.

If we can map each function directly to an address range, not a region, this problem can be solved. As shown in Figure 2(c), f_1 and f_2 can be mapped to disjoint address ranges, from 0 to 1 and from 2 to 3, respectively. This can greatly improve the WCET because B and C can stay loaded after their initial loadings. This mapping causes f_1 to be reloaded when f_1 returns back to f_0 because their allocated SPM spaces overlap, but it happens only once. When f_3 returns, f_0 does not need to be reloaded. In this article, we find an optimal *region-free* mapping (mapping of functions to SPM addresses) for minimizing the WCET of a given program.

3. WCET ANALYSIS FOR DYNAMIC CODE MANAGEMENT

In order to find a mapping that can optimize the WCET of a program, we first need to be able to estimate the WCET of the program for a mapping—which can be either a function-to-region mapping or a region-free mapping. Figure 3 shows an overview of our WCET analysis framework. Given a graph representation of the program, we need to perform two analyses to obtain necessary information about the program. Using this



```
f0 {
  if ( … ) then
    f1 ();
  else
    f2 ();
}
f1 { … }
f2 { … }
```

$l\,(v_0)$	0
$l\,(v_1)$	0
$l\,(v_2)$	0
$l\,(v_3)$	1
$l\,(v_4)$	1
$l\,(v_5)$	1
$l\,(v_6)$	1
$l\,(v_7)$	0

Fig. 4. Inlined CFG represents the global call sequence and the control flow by inlining the CFG of the callee function at each function call.

information, along with a mapping and loop bounds, we formulate an ILP to compute a safe upper bound of the WCET.

3.1. Inlined Control Flow Graph

We use a variant of control flow graphs (CFGs), called *inlined* CFGs, to represent a given program. An inlined CFG is a CFG of a whole program, not just one function, whose edges represent not only control flows within a function but also function calls and returns. An example program is depicted in Figure 4. In this example, like the example from Figure 1(a), the main function, f_0, has one branch and calls f_1 and f_2. We assume that both f_1 and f_2 consist of a single basic block. When f_0 calls f_1 at v_1, the CFG of f_1 is inlined as v_3, and similarly the CFG of f_2 is inlined as v_4. The notable benefit with this representation is that context information or call stack trace is explicit at any node in a graph, which avoids pessimism regarding uncertainties with call history in static analysis. One limitation is that recursive functions cannot be represented, which can be acceptable in the context of real-time embedded applications. Note that this is only a representation of a program for analysis and we do not actually inline all function calls.

Let $G = (V, E, v_s, v_t, F, fn)$ be an inlined CFG. V is the set of vertices, each of which is a basic block. The set of edges is defined as $E = \{(v, w)|$ there is a direct path from v to w due to control flow, a function call, or a return, where $v, w \in V.\}$. Unlike basic blocks in conventional CFGs, function call instructions are always at the end of a basic block and cannot be in the middle of a basic block. Vertices v_s and v_t represent the starting basic block and the terminal basic block. F is the set of functions in the program, and $fn : V \to F$ is a mapping stating that $fn(v)$ is the function that v belongs to.

A mapping $l : V \to \{0, 1\}$ identifies *loading points* of functions. For a vertex v, $l(v)$ is 1 only when there is an immediate predecessor u such that $fn(u) \neq fn(v)$, which means there is an incoming edge from another function. Figure 4 illustrates $fn(v)$ and $l(v)$.

A path is a finite sequence of vertices $p = p_1, p_2, \ldots, p_k$ such that $\forall 1 \leq i \leq k,\ p_i \in V$ and $\forall 1 \leq i < k, \exists(v_i, v_{i+1}) \in E$. The ith vertex on p is denoted by p_i, and the length of a path p is denoted by $len(p)$. A vertex can appear multiple times on a path for the presence of loops. Given a vertex v, $P(v)$ denotes the set of all paths that start from v_s and end with an immediate predecessor of v. For a path p and a function f, $last(p, f) \in V \cup \{\perp\}$ denotes the last occurrence of f on p. Thus, if we let $last(p, f) = p_i$, then $fn(p_i) = f$ and $fn(p_j) \neq f,\ i < j \leq len(p)$. When f does not appear on p, $last(p, f) = \perp$.

3.2. Finding Initial Loading Points

A function needs to be loaded at least once when it is called for the first time, which is analogous to cold misses in caches.

We define a binary mapping $il : V \rightarrow \{0, 1\}$ to identify initial loading points of functions. For a vertex $v \in V$, $il(v)$ is 1 only when v is an initial loading point of $fn(v)$, which is determined using traditional dominance analysis [Khedker et al. 2009] as follows:

$$il(v) = \begin{cases} 0 & \exists d \in SDOM(v),\ fn(d) = fn(v) \\ 1 & \text{otherwise,} \end{cases} \tag{1}$$

where $SDOM(v)$ denotes the set of strict dominators of v. If there is any strict dominator d whose $fn(d)$ value is the same as $fn(v)$, function $fn(v)$ can be safely assumed to have been loaded before executing v. Otherwise, v is an initial loading point. For example, in the example program in Figure 4, v_3 is a potential loading point of f_1, and its strict dominators are v_0 and v_1. Since both v_0 and v_1 belong to f_0, not f_1, v_3 is an initial loading point of f_1, and thus, $il(v_3) = 1$.

3.3. Interference Analysis: Finding the Interference Among Functions

At a loading point v that is not an initial loading point, $fn(v)$ is guaranteed to have been loaded before control reaches v. To determine if $fn(v)$ is still loaded at v, we make a conservative assumption as follows. If there exists a function $g \neq fn(v)$ that satisfies the following two conditions, we assume that $fn(v)$ has been evicted from the SPM:

(1) g and $fn(v)$ share SPM space (their allocated SPM spaces overlap).
(2) There exists a path $p \in P(v)$ on which g is executed between $last(p, fn(v))$ and v.

Satisfying two conditions means, in other words, that $fn(v)$ could have been evicted by g on a path from $last(p, fn(v))$ to v. The first condition cannot be checked because the SPM addresses of functions are not decided before the code mapping stage (Section 4). The second condition, however, can be checked by analyzing the given CFG.

If the second condition is satisfied, we say that at loading point v, $fn(v)$ and g have *interference*. Interference analysis[3] finds the set of all functions that potentially interfere with f at all loading points v, namely, *interference set*, defined next.

Definition 3.1 (*Interference Set*). Let $G = (V, E, v_s, v_t, F, fn)$ be an inlined CFG. For a vertex $v \in V$ and a function $f \in F$, the interference set $IS[v, f] \subseteq F \backslash \{f\}$ is the set of all functions that appear between the path between $last(p, f)$ and v, excluding $last(p, f)$ and v, for all paths $p \in P(v)$.

When $last(p, f)$ is $\perp$ for all paths $p \in P(v)$, $IS[v, f] = \emptyset$. The following equation restates the previous definition:

$$\forall v \in V, f \in F,\ IS[v, f] = \bigcup_{\forall p \in P(v)} \{fn(p_j) \mid i < j \leq len(p),\ p_i = last(p, f)\}. \tag{2}$$

Table I shows interference sets for the example in Figure 4. To help follow how interference sets are calculated at each vertex v, the table also shows the set of $last(p, f)$ for all paths $p \in P(v)$ in the right three columns.

In other words, interference set $IS[v, f]$ is the set of functions that *could* evict f from the SPM before f is executed at v. The eviction can actually occur if any function in $IS[v, f]$ is assigned an SPM space that overlaps with the SPM space assigned for f. Since a loading point v loads $fn(v)$, only $IS[v, fn(v)]$ is meaningful in estimating DMA

[3] The term "interference analysis" has been used in the context of compiler optimization, such as in register allocation or in optimizing parallel programs. Our interference analysis is different from any of those, but similar in the sense that the results are used to predict any side effect of compiler decisions. For example, allocating a register to a variable may cause additional spills of other interfering variables, and mapping a function to an SPM address may cause additional DMA overhead for loading other interfering functions.

Table I. Interference Sets for the Example Program in Figure 4

	$IS[v, f_0]$	$IS[v, f_1]$	$IS[v, f_2]$	$\bigcup_{\forall p \in P(v)}\{last(p, f)\}$		
				f_0	f_1	f_2
v_0, v_1, v_2, v_3, v_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$ or has only immediate predecessor of v		
v_5	$\{f_1\}$	$\emptyset$	$\emptyset$	$\{v_1\}$	$\{v_3\}$	$\emptyset$
v_6	$\{f_2\}$	$\emptyset$	$\emptyset$	$\{v_2\}$	$\emptyset$	$\{v_4\}$
v_7	$\emptyset$	$\{f_0\}$	$\{f_0\}$	$\{v_5, v_6\}$	$\{v_3\}$	$\{v_4\}$

ALGORITHM 1: Interference Analysis

```
  Input: Inlined CFG ($G$)      Output: Interference sets ($IS$)
1 foreach $(v, f) \in V \times F$ do $\mathrm{IN}[v, f] \leftarrow \emptyset$
2 repeat
3 |  foreach $(v, f) \in V \times F$ do Evaluate Equations (3) and (4)
  until $IN[v, f]$ and $OUT[v, f]$ stay unchanged for all $v \in V$ and $f \in F$
4 foreach $(v, f) \in V \times F$ do $IS[v, f] = \mathrm{IN}[v, f] - \{f\}$
```

costs. Nevertheless, the interference sets are calculated for all functions at each vertex to pass down the information to successor vertices.

Interference sets can be calculated by a form of forward data-flow analysis, using the following data-flow equations, called from Algorithm 1. Let IN$[v, f]$ and OUT$[v, f]$ be the interference sets $IS[v, f]$ before and after executing v, respectively:

$$\mathrm{IN}[v, f] = \bigcup_{(u,v)\in E} \mathrm{OUT}[u, f] \tag{3}$$

$$\mathrm{OUT}[v, f] = \begin{cases} \emptyset & \text{if } f \neq fn(v) \wedge \mathrm{IN}[v, f] = \emptyset \\ \{fn(v)\} & \text{if } f = fn(v) \\ \mathrm{IN}[v, f] \cup \{fn(v)\} & \text{otherwise.} \end{cases} \tag{4}$$

Input value IN$[v, f]$ is the union of output values from all predecessors, and there are three different cases regarding how output value OUT$[v, f]$ is updated. First, when f is not $fn(v)$, OUT$[v, f]$ remains empty unless IN$[v, f]$ has any function in it. IN$[v, f]$ can become a nonempty set only when f has been executed previously, which is done by the second condition. The second condition says that when f is $fn(v)$, any collected execution history in IN$[v, f]$ is reset and the output value contains only $fn(v)$. Once this happens, starting from the successors u of v, IN$[u, f]$ will not be an empty set, and the function execution history can be recorded by taking a union of the input value and $fn(u)$, as seen in the third condition. Notice that Algorithm 1 sets $IS[v, f]$ to be IN$[v, f] - \{f\}$ at line 4, after data-flow values converge to a final fixed point, to comply to the definition that $IS[v, f]$ does not contain f.

3.4. ILP Formulation for WCET Analysis

We formulate an ILP to find a safe upper bound of the WCET of a given program. Variables in the ILP are written in capital letters, and constants are in small letters. The formulation requires the input inlined CFG G to be acyclic, so we require G to be reducible and remove all back edges first.

The high-level structure of our formulation is similar to the one from the previous work [Suhendra et al. 2005; Falk and Kleinsorge 2009] in two aspects: (1) a WCET estimate is obtained by accumulating the cost of each basic block backward from the end to the start of the program (Equation (6)), and (2) the objective is to minimize the WCET (Equation (5)). There are, however, significant differences in the rest of the formulation as we model the function loading cost at each vertex (Equation (13)).

Let W_v be a WCET estimate from v to the end of the program. Thus, W_{v_s} is a WCET estimate for the whole program. The objective is to get a safe-yet-tight estimate of the WCET of the program as follows:

$$minimize \quad W_{v_s}. \tag{5}$$

Each vertex v can contribute to the WCET with the sum of its computation cost C_v and its loading cost L_v. C_v is the time to execute all instructions in v, except the time to execute DMA instructions, which is L_v. For each successor w of v, W_v is greater than or equal to the sum of the cost of v and the cost of w. This makes W_v a safe upper bound of the WCET from v to the end of the program. The terminal basic block does not have any successor, so W_{v_t} is the cost of itself:

$$\begin{aligned} \forall(v, w) \in E, \quad W_v &\geq W_w + C_v + L_v \\ W_{v_t} &= C_{v_t} + L_{v_t}. \end{aligned} \tag{6}$$

The computation cost C_v is a product of the number of times v is executed in the worst case (n_v) and the worst-case estimation of the time it takes to execute the instructions in v for once (c_v):

$$C_v = n_v \cdot c_v. \tag{7}$$

For loading cost L_v to exist, v must be a loading point, that is, $l(v) = 1$. To employ the value of $l(v)$ in the formulation, $l(v)$ is imported as a constant l_v as follows. Similarly, the information regarding initial loading point $il(v)$ is imported as a constant il_v:

$$l_v = l(v) \tag{8}$$

$$il_v = il(v). \tag{9}$$

The mapping information is taken into account as follows. For a pair of functions, f and g, a binary constant $o_{f,g}$ is only one when their allocated SPM address ranges overlap. When the mapping is function-to-region mapping, this means that both functions are mapped to the same region. With a region-free mapping, this is calculated using the mapped address and the size of each function:

$$o_{f,g} = \begin{cases} 1 & \text{if the allocated SPM spaces for } f \text{ and } g \text{ overlap} \\ o & \text{otherwise.} \end{cases} \tag{10}$$

Let d_f denote the time it takes to load function f by a DMA operation plus the overhead of executing DMA instructions. The loading cost L_v is modeled as follows. Table II shows different scenarios when loading cost L_v exists. If there exists any interfering function whose allocated SPM space overlaps with that of $fn(v)$, $fn(v)$ needs to be reloaded every time v is executed. AM_v (Always-Miss) models the loading cost in this case:

$$\forall f \in IS[v, fn(v)], \quad AM_v \geq n_v \cdot d_{fn(v)} \cdot o_{fn(v),f}. \tag{11}$$

Consider an initial loading point v that is executed more than once in a loop. If there is no interfering function or none of the interfering function shares SPM space with $fn(v)$, $fn(v)$ needs to be loaded only once. FM_v (First-Miss) models the loading cost in this case. The value of FM_v is $d_{fn(v)}$ as $fn(v)$ is loaded only once. If, however, any interfering function shares SPM space with $fn(v)$, it becomes Always-Miss, and the value of FM_v should be the same as AM_v. The difference in AM_v and $d_{fn(v)}$ is compensated by adding $(n_v - 1) \cdot d_{fn(v)}$ to $d_{fn(v)}$ as follows:

$$\forall f \in IS[v, fn(v)], \quad FM_v \geq d_{fn(v)} + (n_v - 1) \cdot d_{fn(v)} \cdot o_{fn(v),f}. \tag{12}$$

Table II. Categorization of Function Loading at Vertex *v*

	Initial Loading Point ($il_v = 1$)	Noninitial Loading Point ($il_v = 0$)
$fn(v)$ shares SPM space with an interfering function ($\exists f \in IS[v, fn(v)]$, $o_{fn(v),f} = 1$)	Always-Miss	
Otherwise ($\forall f \in IS[v, fn(v)]$, $o_{fn(v),f} = 0$)	First-Miss	No DMA Cost

Finally, since the loading cost is present only when l_v is 1, and its value is either FM_v or AM_v, it is modeled by the following constraint:

$$L_v = l_v \cdot (il_v \cdot FM_v + (1 - il_v) \cdot AM_v). \tag{13}$$

After solving the previous ILP, the objective value W_{v_s} is a safe WCET estimate of the given program running on an SPM-based architecture with dynamic code management using the given code mapping.

4. OPTIMIZING WCET IN CODE MANAGEMENT

In this section, we present our three techniques of finding a code mapping for WCET.

The ILP formulations in this section are extensions to the formulation in Section 3.4, to *explore* all mapping solutions instead of taking a fixed mapping solution as input. The extensions have no similarities with previous work [Suhendra et al. 2005; Falk and Kleinsorge 2009] and are completely new.

4.1. ILP Formulation for Optimal Function-to-Region Mapping

A function-to-region mapping solution is represented by the following binary variables:

$$\forall f \in F, r \in R, \quad M_{f,r} = \begin{cases} 1 & \text{if } f \text{ is mapped to } r \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

The number of regions used in a mapping solution can vary for different solutions. For example, all functions can be mapped to one region (there is only one region), or each function can be mapped to a unique region (the number of regions is the same as the number of functions). To handle various numbers of regions, we let the set of all regions, R, be a set of integers ranging from 1 to $|F|$, each of which represents a unique region. If a mapping solution uses only $n < |F|$ regions, there will be $(|F| - n)$ regions that do not have any functions mapped to them.

The following constraints ensure the feasibility of mapping solutions that the solver will explore. First, every function is mapped to exactly one region:

$$\forall f \in F, \quad \sum_{r \in R} M_{f,r} = 1. \tag{15}$$

Second, the sum of the region sizes is not greater than the SPM size:

$$\begin{gathered} \forall f \in F, r \in R, \quad S_r \geq M_{f,r} \cdot s_f \\ \text{SPMSIZE} \geq \sum_{r \in R} S_r, \end{gathered} \tag{16}$$

where SPMSIZE is the size of the SPM, and s_f is the size of function f. S_r is a variable that represents the size of the largest function mapped to r.

For a pair of functions f and g and a region r, we use a binary variable $M_{f,g,r}$ that is 1 only when f and g are both mapped to r. This is represented by the following logical condition between variables. If both f and g are mapped to r, only the constraints in

Set 1 should be satisfied but not the constraints in Set 2, and vice versa:

$$\text{Set 1: } M_{f,r} + M_{g,r} > 1 \qquad \text{Set 2: } M_{f,r} + M_{g,r} \leq 1$$
$$M_{f,g,r} = 1 \qquad M_{f,g,r} = 0.$$

The previous logical constraints can be integer programmed using the standard way of formulating logical constraints [Bradley et al. 1977] as follows:

$$\forall f, g \in F, r \in R, \quad \begin{aligned} & M_{f,r} + M_{g,r} + B \cdot (1 - M_{f,g,r}) > 1 \\ & M_{f,r} + M_{g,r} \leq 1 + B \cdot M_{f,g,r}, \end{aligned} \tag{17}$$

where B is a constant chosen to be large enough so that regardless of the value of $M_{f,g,r}$, both constraints should be satisfied at the same time. In this case, B should be at least 2 to make $M_{f,r} + M_{g,r} + B \cdot (1 - M_{f,g,r}) > 1$ satisfiable when $M_{f,g,r}$ is 0.

Then, the constraints for FM_v and AM_v from the constraints in Equation (11) and Equation (12) need to be rewritten using $M_{fn(v),f,r}$ as follows:

$$\forall f \in IS[v, fn(v)], r \in R, \quad AM_v \geq n_v \cdot d_{fn(v)} \cdot M_{fn(v),f,r} \tag{18}$$

$$FM_v \geq d_{fn(v)} + (n_v - 1) \cdot d_{fn(v)} \cdot M_{fn(v),f,r}. \tag{19}$$

The solution of this ILP formulation is an optimal function-to-region mapping represented by the set of variables $M_{f,r}$ for all functions $f \in F$ and regions $r \in R$. The final objective value W_{v_s} is the WCET estimate for the found mapping solution.

4.2. WMP: A Heuristic Alternative to the ILP Formulation

The ILP-based technique from the previous section can find an optimal solution, but it can take a long time for an ILP solver to find one. As each function needs to be mapped to each region, and the number of regions can be as many as the number of functions, the solution space of the ILP grows exponentially with the number of functions.

To solve this problem, we present *WCET-aware Merging and Partitioning* (WMP), a polynomial-time heuristic technique that builds upon the ways of searching the solution space of our previous techniques, *function mapping by updating and merging* (FMUM) and *function mapping by updating and partitioning* (FMUP) [Jung et al. 2010]. As the name suggests, FMUM starts with assigning a separate region to every function and tries to merge regions so that the mapping can fit in the SPM and the cost of the mapping decreases, whereas FMUP starts with having only one region and iteratively partitions a region into two regions. While the cost function in these techniques estimates the overall amount of DMA transfers, we introduce a new cost function that estimates the WCET of the program.

Before discussing the details of the WMP algorithm, we would like to point out that the ILP-based technique can also be used as a heuristic with a time limit set to the solver, as we do in Section 5. This makes the solver output the best solution found by the time limit, which may not be optimal. In our experiments with a 3-hour time limit, this ILP-based technique could always find solutions that are better (meaning that the resulting WCET is smaller) or at least as good as the solutions found by WMP. This, however, brings up another problem of choosing a time limit that is long enough to find good solutions. For example, in our experiments with benchmark "1REG," the ILP solver needed at least 50 seconds to find a solution as good as the solution found by WMP and at least 20 minutes to find a solution better than WMP's solution. On the other hand, WMP could find a solution within a second for all benchmarks, and the increase in WCETs compared to the ILP with a 3-hour time limit is not greater than 6.5%. In this sense, WMP is still a reasonable and scalable alternative to the ILP.

Algorithm 2 shows the pseudocode. Given an inlined CFG, the interference sets, and the size of SPM, it returns a function-to-region mapping M. A mapping solution, M,

ALGORITHM 2: WMP: A Heuristic to Find a Function-to-Region Mapping for WCET. ©IEEE.

```
   Input: Inlined CFG ($G$), SPM size ($S$)
   Precondition: $S \geq \max_{f \in F} s_f$
   Output: A feasible function-to-region mapping ($M$) (Size($M$) $\leq S$)
   function WMP($G$,$S$)
1      remove all back edges in $G$
2      $T \leftarrow$ Topologically sorted vertices of $G$
3      $M_m \leftarrow$ Merge($G$, $T$, $S$)
4      $M_p \leftarrow$ Partition($G$, $T$, $S$)
5      if Cost($M_m$, $T$) < Cost($M_p$, $T$) then
6          $M \leftarrow M_m$
       else
7          $M \leftarrow M_p$
8      return $M$
```

```
    Input: Function-to-region mapping ($M$),
           Topological-sorted vertex list ($T$)
    Output: The WCET estimate ($c[v_t]$)
    function Cost($M$, $T$)
9       initialize $c[v]$ to 0 for all $v \in V$
10      for $v$ in $T$ from head to tail do
11          $c \leftarrow n_v \cdot c_v$
12          if $l(v) = 1$ then
13              if $\exists f \in IS[v, fn(v)]$ such that
                $M[fn(v)] = M[f]$ then
14                  $c \leftarrow c + n_v \cdot d_{fn(v)}$
                else
15                  if $il(v) = 1$ then
16                      $c \leftarrow c + d_{fn(v)}$
17          $c[v] \leftarrow c + \max_{(u,v) \in E} c[u]$
18      return $c[v_t]$
```

is represented by an integer array whose size is the same as the number of functions. The ID of a function is represented by an array index, and its mapped region is the value of the array element. For example, if function 1 is mapped to region 2, $M[1] = 2$.

We use the longest path in the input inlined CFG as the cost of a mapping. To find the longest path, we first remove all back edges from the graph and topologically sort the vertices (lines 2–3). The cost function, Cost, visits each vertex in topological order and calculates the computation cost (line 11) and the loading cost (lines 12–15) for the given mapping. At each vertex v, the final cost of the vertex $c[v]$ is the sum of its own cost and the maximum cost among the costs of all predecessors (line 16). Thus, the cost of the terminal vertex $c[v_t]$ becomes the longest path length.

We find two mapping solutions by merging and partitioning (lines 3–4), whose algorithms are shown in Algorithm 3. The one with a lesser cost is selected (lines 5–7).

Algorithm 3 shows two heuristics, Merge and Partition. Merge starts with mapping each function to a separate region (line 1). In every iteration of the while loop at lines 2 to 14, we take every pair of two regions (lines 4–5) to merge and create a temporary mapping M' where two regions are merged (lines 6–7). We check the cost of M' and keep a record of the best pair of regions to be merged and its cost (lines 8–11). After trying all combinations, we change the original mapping M by merging the best pair of regions (line 12). The loop repeats until the mapping can fit in the SPM; that is, the sum of the sizes of regions is smaller than *SPMSIZE* (line 2). Partition starts with mapping all functions to one region (line 14). Variable *nr* represents the current number of regions. Again, we create a duplicate of M and move each function f to a different region r, creating another region $nr + 1$ (lines 18–20). We keep a record of the best combination of f and r and its cost (lines 21–24). After trying all functions, we move function bf to region br (line 25). The loop repeats until the number of regions is not greater than $|F|$ (line 15) or until the number of regions stops increasing (line 26).

Function Size is defined in Algorithm 4. It calculates the memory size requirement of a given mapping M by summing up the size of the largest function in each region.

The while loop in Merge takes at most $|F| - 1$ times because the number of regions decreases by one at every iteration. The for-loop nest at lines 5 to 7 takes $|F|^2$ times at most. Merging two regions requires checking every array element at least once to

ALGORITHM 3: Search Feasible Mapping Solutions by Merging and Partitioning. ©IEEE.

```
Input: Inlined CFG ($G$), Topologically sorted vertex list ($T$), SPM size ($S$)
Precondition: $S \geq \max_{f \in F} s_f$
Output: A feasible function-to-region mapping ($M$) (Size($M$) $\leq S$)

function Merge($G$, $T$, $S$)
1     initialize $M[f]$ to $f$ for $1 \leq f \leq |F|$
2     while Size($M$) $> S$ do
3         $bc \leftarrow \infty$
4         for $r_1 = 1$ to $|F| - 1$ do
5             for $r_2 = r_1 + 1$ to $|F|$ do
6                 $M' \leftarrow$ a duplicate of $M$
7                 merge $r_1$ and $r_2$ in $M'$
8                 $nc \leftarrow$ Cost($M'$, $T$)
9                 if $nc < bc$ then
10                    $br_1 \leftarrow r_1, br_2 \leftarrow r_2$
11                    $bc \leftarrow nc$
12        merge $br_1$ and $br_2$ in $M$
13    return $M$

function Partition($G$, $T$, $S$)
14    $M[f] \leftarrow 1$ for $1 \leq f \leq |F|$, $nr \leftarrow 1$
15    while $nr \leq |F|$ do
16        $bc \leftarrow$ Cost($M$, $T$)
17        for $f = 1$ to $|F|$ do
18            $M' \leftarrow$ a duplicate of $M$
19            for $r = 1$ to $\min(nr + 1, |F|)$ do
20                $M'[f] \leftarrow r$
21                $nc \leftarrow$ Cost($M'$, $T$)
22                if $nc < bc \wedge$ Size($M'$) $\leq S$ then
23                    $bf \leftarrow f, br \leftarrow r$
24                    $bc \leftarrow nc$
25        if $bc <$ Cost($M$, $T$) then $M[bf] \leftarrow br$
26        if $br = nr + 1$ then $nr \leftarrow nr + 1$
          else break
27    return $M$
```

ALGORITHM 4: Find the SPM Size Requirement of a Given Function-to-Region Mapping

```
Input: Inlined CFG($G$), function-to-region Mapping ($M$), Function sizes ($s_f, \forall f \in F$)
Output: The SPM size that the given mapping $M$ requires.

function Size($M$)
1     initialize $S[r]$ to 0 for $1 \leq r \leq |F|$
2     for each $f \in F$ do
3         if $S[M[f]] < s_f$ then $S[M[f]] \leftarrow s_f$
4     return $\sum_{r \in R} S[r]$
```

find all functions mapped to one region and move it to another region, which takes $O(|F|)$ time complexity. The time complexity of Cost is $O(|V| \cdot |F - 1|)$ since it visits every vertex only once, and the number of functions in the interference set $IS[v, fn(v)]$ can be at most $|F| - 1$ because $fn(v)$ is excluded in the set. Thus, the time complexity of function Merge is $O(|F|^4 \cdot |V|)$. Similarly, Partition has the same time complexity because the while loop and for loops at lines 16, 18, and 20 iterate at most $|F|$ times.

The WMP algorithm always terminates. In Merge, the SPM size S is greater than or equal to the size of the largest function by assertion at the beginning, and the size of mapping, Size(M), is reduced after every iteration of the while loop (line 2) by merging two regions (line 12). All for loops in Merge (lines 4–5) have finite loop bound $|F|$ too. Function Cost finishes in a finite number of iterations because the vertex list has finite length $|V|$ (line 10) and the interference sets can have at most $|F| - 1$ vertices (line 13). Thus, Merge terminates in a finite number of steps. Similarly, Partition also finishes in a finite number of steps because at the end of every iteration, either the number of regions nr increases or the loop terminates (line 26).

The WMP algorithm is sound and complete in that it always finds a solution that is a feasible mapping that can fit in the SPM. Merge always returns a feasible mapping because in every iteration, two regions are merged in line 12 and the loop termination condition in line 2 ensures the feasibility of the mapping. The initial solution of Partition is mapping all functions into one region, which is certainly feasible because of the precondition $S \geq \max_{f \in F} s_f$. During the execution of the algorithm, the mapping changes only in a way that the resulting mapping fits in the SPM (line 22).

The WMP algorithm is, however, not optimal because it does not explore the entire solution space. As a heuristic, WMP trades optimality for speed. For example, `Merge` only considers merging two regions at time in a greedy fashion. Once two regions are merged, the functions in the regions have to be mapped to the same region until the end of the algorithm.

4.3. Optimizing WCET Using Region-Free Mappings

Variable A_f represents region-free mapping of *f*, the address at which function *f* will be loaded, and it should be in the following range:

$$0 \leq A_f \leq \text{SPMSIZE} - s_f, \tag{20}$$

where s_f denotes the size of function *f*. Then, the following constraints compare the mapped addresses of two functions and represent their relations. For a pair of functions *f* and *g*, binary variable $G_{f,g}$ is 1 if A_f is greater than A_g, and 0 otherwise:

$$\begin{gathered} \forall f, g \in F \text{ such that } f \neq g, \quad -\mathcal{M}(1 - G_{f,g}) \leq A_f - A_g \leq \mathcal{M} \cdot G_{f,g} \\ G_{f,g} + G_{g,f} = 1, \end{gathered} \tag{21}$$

where $\mathcal{M}$ is a sufficiently large integer, used to linearize the *if* conditions. In our formulation, *SPMSIZE* can be safely used as $\mathcal{M}$. For example, if $G_{f,g}$ is 1, the previous constraints become $0 \leq A_f - A_g \leq \mathcal{M}$ and $G_{g,f} = 0$, so A_f has to be greater than or equal to A_g. If $G_{f,g}$ is 0, the previous constraints become $-\mathcal{M} \leq A_f - A_g \leq 0$ and $G_{g,f} = 1$, so A_f has to be less than or equal to A_g.

The address range that is allocated to function *f* is $[A_f + 0, A_f + 1, \ldots, A_f + s_f - 1]$, where s_f is the size of *f*. For a pair of functions *f* and *g*, to make sure that their addresses do not overlap, either one of the two constraints should be satisfied: $A_f + s_f - 1 < A_g$ when $A_f < A_g$ ($G_{f,g} = 1$), or $A_g + s_g - 1 < A_f$ when $A_g > A_f$ ($G_{g,f} = 1$). Built on this idea, the following constraints make binary variable $O_{f,g}$ to be 1 if the address ranges of *f* and *g* overlap, and 0 otherwise:

$$\begin{aligned} \forall f, g \in F \text{ such that } f \neq g, \quad A_f + s_f &< A_g + 1 + \mathcal{M} \cdot G_{f,g} + \mathcal{M} \cdot O_{f,g} \\ \mathcal{M} \cdot (1 - O_{f,g}) + A_f + s_f &\geq A_g + 1 + \mathcal{M} \cdot O_{f,g} \\ A_g + s_g &< A_f + 1 + \mathcal{M} \cdot G_{g,f} + \mathcal{M} \cdot O_{f,g} \\ \mathcal{M} \cdot (1 - O_{f,g}) + A_g + s_g &\geq A_f + 1 + \mathcal{M} \cdot O_{f,g} \\ O_{f,g} &= O_{g,f}. \end{aligned} \tag{22}$$

For example, if $A_f > A_g$ ($G_{f,g} = 1$), the first four lines in the previous equation become as follows:

$$\begin{aligned} A_f + s_f &< A_g + 1 + \mathcal{M} + \mathcal{M} \cdot O_{f,g} \\ \mathcal{M} \cdot (1 - O_{f,g}) + A_f + s_f &\geq A_g + 1 + \mathcal{M} \cdot O_{f,g} \\ A_g + s_g &< A_f + 1 + \mathcal{M} \cdot O_{f,g} \\ \mathcal{M} \cdot (1 - O_{f,g}) + A_g + s_g &\geq A_f + 1 + \mathcal{M} \cdot O_{f,g}. \end{aligned}$$

The first line becomes meaningless because of the third term, $\mathcal{M}$, on the right-hand side, and the second line also becomes meaningless regardless of the value of $O_{f,g}$ because A_f is greater than A_g. When the address ranges of *f* and *g* do overlap ($A_g + s_g \geq A_f + 1$), the third line ensures that $O_{f,g}$ becomes 1, and the fourth line becomes meaningless—it satisfies regardless of the value of $O_{f,g}$. When the address ranges do not overlap

Table III. Benchmarks Used in Our Evaluation

	Total Code Size	Largest Function Size (B)	Number of Functions	Source
cnt	948	332	6	Mälardalen
matmult	1,064	304	7	Mälardalen
dijkstra	1,644	744	6	MiBench
compress	2,892	872	9	Mälardalen
sha	2,420	1,040	7	MiBench
fft1	3,404	1,984	6	Mälardalen
lms	3,804	980	8	Mälardalen
edn	4,624	1,924	9	Mälardalen
adpcm	8,468	2,272	17	Mälardalen
rijndael	9,448	3,128	7	MiBench
statemate	10,580	3,520	8	Mälardalen
1REG	27,736	7,748	28	Proprietary
DAP1	36,400	27,860	17	Proprietary
susan	51,672	10,504	19	MiBench
DAP3	56,748	43,004	28	Proprietary

($A_g + s_g < A_f + 1$), the third line becomes meaningless, but the fourth line ensures that $O_{f,g}$ becomes 0.

We rewrite Equation (11) and Equation (12) with $O_{f,g}$ variables as follows:

$$\forall f \in IS[v, fn(v)], \quad AM_v \geq n_v \cdot d_{fn(v)} \cdot O_{fn(v),f} \tag{23}$$

$$FM_v \geq d_{fn(v)} + (n_v - 1) \cdot d_{fn(v)} \cdot O_{fn(v),f} \tag{24}$$

The final objective value W_{v_s} after solving this ILP is a WCET estimate, and the A_f variables represent the optimal mapping of functions to SPM addresses.

5. EXPERIMENTAL RESULTS

We use various benchmarks from the Mälardalen WCET benchmark suite [Gustafsson et al. 2010] and MiBench suite [Guthaus et al. 2001], together with three real-world proprietary automotive powertrain control applications from industry. Among 31 benchmarks in the Mälardalen suite and 29 benchmarks in the MiBench suite, we exclude the ones with recursion or that have fewer than six functions. From the Mälardalen suite, we use all eight benchmarks that have at least six functions and do not have recursion. The MiBench suite is in general much larger in size and more complicated, and we were not able to generate the inlined CFGs for six benchmarks due to the presence of recursion or function pointers, and 19 due to the complexity of compiled binaries.[4] We use all of the remaining four benchmarks from the MiBench suite.[5]

Table III shows the benchmarks used in the evaluation. The sizes shown in the table are the sizes after management code is inserted into the code. Only functions in the user code are considered, and library function calls are considered to take the same cycles as normal arithmetic instructions. We compile benchmarks for ARM v4 ISA and generate inlined CFGs from the disassemblies.

We assume the cost of loading x bytes into SPM by DMA to be $(L - B/W) + (\lceil x/W \rceil)$ cycles, where L is the cache miss latency, B is the cache block size of the system in comparison, and W is the word size, as it is modeled by Whitham and Audsley [2009].

[4] We could not generate inlined CFGs for these benchmarks because of the complexity of makefile build systems. This is only a technical difficulty in our implementation and not a fundamental limitation.

[5] We commented out a recursive function call for printing in “dijkstra” without changing the core algorithm.

The first term is the setup time that takes in every transfer regardless of the transfer size, and the second is the transfer time that corresponds to the transfer size. As in Whitham's work, we use 50, 16, and 4 for L, B, and W, respectively. We observed that over a large set of different parameters, there was no significant difference in results in terms of relative performance comparison.

To simplify computing WCET estimates, we assume that every instruction takes one cycle as it is on processors designed for timing predictability, such as PRET [Liu et al. 2012; Zimmer et al. 2014]. Thus, the worst-case execution time of a basic block in number of cycles is assumed to be the same as the number of instructions in it, unless it has DMA instructions. All data accesses are from a separate data SPM, without any contention with the accesses to the main memory or the instruction SPM. These assumptions are only for simplifying the evaluation and not limitations of our approach. We can extend our work by combining any microarchitecture analysis work to consider other timing effects such as pipeline hazards, but it is outside the scope of this article.

Loop bounds are found by profiling, except for the powertrain control applications that have infinite loops in the main scheduler. For such benchmarks, loop bounds are set to be a power of 10 according to the level of nesting. We use the Gurobi optimizer 6.0[6] to solve ILPs. All experiments are run on a 2.2Ghz Core i7 processor with 16GB of RAM. We set a time limit of 3 hours for the ILP solver so that if it cannot find an optimal solution within 3 hours, we use the best solution found up to that point.

The correctness of our WCET estimation is verified by running selected benchmarks on the gem5 simulator [Binkert et al. 2011]. We modified the simulator so that it maintains a state machine of an SPM that is updated by DMA operations. Every instruction takes one cycle, and at function calls and returns, additional cycles are taken according to the state of the SPM, for executing management code and DMA operations. The number of cycles each benchmark took on the simulator was always less than or equal to the WCET estimates we obtained by analysis. The whole evaluation setup—the tools for generating inlined CFGs, performing cache analysis, finding mappings, and the simulator—is publicly available for download.[7]

We use two SPM sizes, A and B, for each benchmark. A and B are $l + (t - l) * 0.1$ and $l + (t - 1) * 0.3$, respectively, where l and t denote the size of the largest function and the total code size. We picked these two values, 0.1 and 0.3, to stress test the mapping techniques' capabilities. With too large SPM sizes (values closer to 1), any mapping techniques are likely to allocate separate regions to functions, which will generally be beneficial for both ACET and WCET. Likewise, too small SPM sizes (values closer to 0) can be too restrictive to compare mapping techniques effectively.

5.1. Function-to-Region Mappings: ILP Versus WMP Heuristic

WMP is a greedy heuristic that may not always be able to find an optimal solution. Figure 5 shows the increase in the WCET estimates when the mappings found by WMP are used, compared to the case in which optimal mappings found by the ILP are used. On the x-axis, there are two cases for each benchmark, showing two different SPM sizes. WMP can find the same optimal solutions as the ILP-based technique does in most cases, and the maximum increase in the WCET is 6.5%.

Table IV shows the algorithm execution times of ILP-based mapping techniques (rbILP and rfILP denote the region-based ILP for finding function-to-region mappings and for region-free mappings, respectively) and the heuristic. The algorithm execution times include the times for running all analyses. The ILP-based technique can find an

[6] Gurobi Optimization: http://www.gurobi.com.

[7] https://github.com/yooseongkim/SPMCodeManagement.

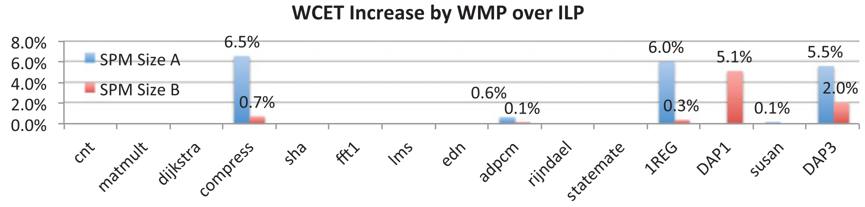


Fig. 5. The increase in WCET estimates by using WMP over ILP is limited within 6.5%.

Table IV. Execution Time Comparison of ILP and WMP Heuristic

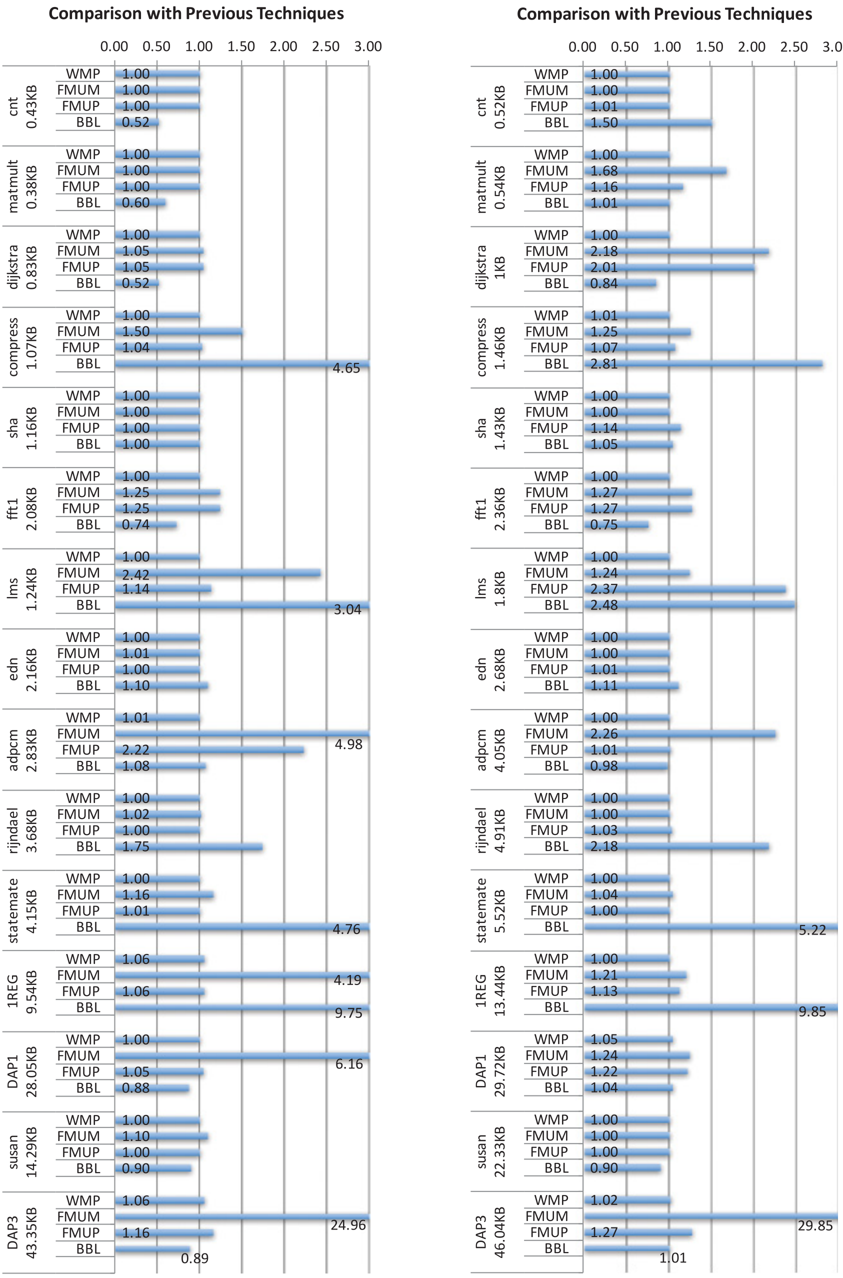
	SPM	Execution Time (sec.)				SPM	Execution Time (sec.)		
	Size	rbILP	rfILP	WMP		Size	rbILP	rfILP	WMP
cnt	432	0.01	0.001	$<10^{-3}$	adpcm	2,896	19.41	11.23	$<10^{-3}$
	528	0.07	0.001	$<10^{-3}$		4,144	8890.3	235.5	0.05
matmult	384	0.01	0.01	$<10^{-3}$	rijndael	3,760	0.02	0.001	$<10^{-3}$
	544	0.1	0.001	$<10^{-3}$		5,024	0.46	0.14	$<10^{-3}$
dijkstra	848	0.05	0.01	$<10^{-3}$	statemate	4,240	0.06	0.02	0.002
	1,024	0.16	0.01	$<10^{-3}$		5,648	0.34	0.18	0.003
compress	1,088	0.3	0.09	0.001	1REG	9,760	>3 hrs	10223.9	0.29
	1,488	0.5	0.18	0.001		13,760	>3 hrs	>3 hrs	0.35
sha	1,184	0.04	0.01	0.001	DAP1	28,720	12.37	5.27	0.11
	1,456	0.55	0.17	0.001		30,432	8.4	3.81	0.10
fft1	2,128	0.18	0.05	$<10^{-3}$	susan	14,624	1.08	0.68	0.05
	2,416	0.2	0.09	0.002		22,864	0.34	0.27	0.05
lms	1,264	0.14	0.08	0.003	DAP3	44,384	>3 hrs	9938.1	0.83
	1,840	0.35	0.09	0.003		47,136	54.3	39.22	0.98
edn	2,208	0.02	0.001	0.001					
	2,736	0.04	0.001	0.001					

optimal solution within seconds for most cases, but for larger benchmarks like "1REG" and "DAP3," it cannot finish within the time limit. In contrast, WMP can finish under a second for all benchmarks. A point worth noting here is that as Figure 5 shows, the WCETs resulting from using the ILP-based technique with the 3-hour time limit are always lower than or at least the same as the WCETs resulting from using the heuristic. In our experiments, the qualities of all solutions found by the time limit are within 3% of optimality. This means that solving the ILP within a reasonable time limit or an optimality range can be a good heuristic itself.

5.2. Function-to-Region Mappings: Comparison with Previous Techniques

We evaluate our mapping techniques in comparison with two function-level techniques [Jung et al. 2010] and one basic-block-level technique [Puaut and Pais 2007]. The two function-level techniques, namely, FMUM and FMUP, take iterative approaches like WMP. They are designed to optimize ACET as their cost function estimates the overall amount of DMA transfers. The basic-block-level technique, denoted as BBL in this section, loads the basic blocks in loops to the SPM at loop preheaders and is optimized for reducing the WCET. This is a dynamic management technique as the SPM contents change at loop preheaders. We assume that fetching an instruction directly from the main memory takes L (50) cycles (see the beginning of Section 5).

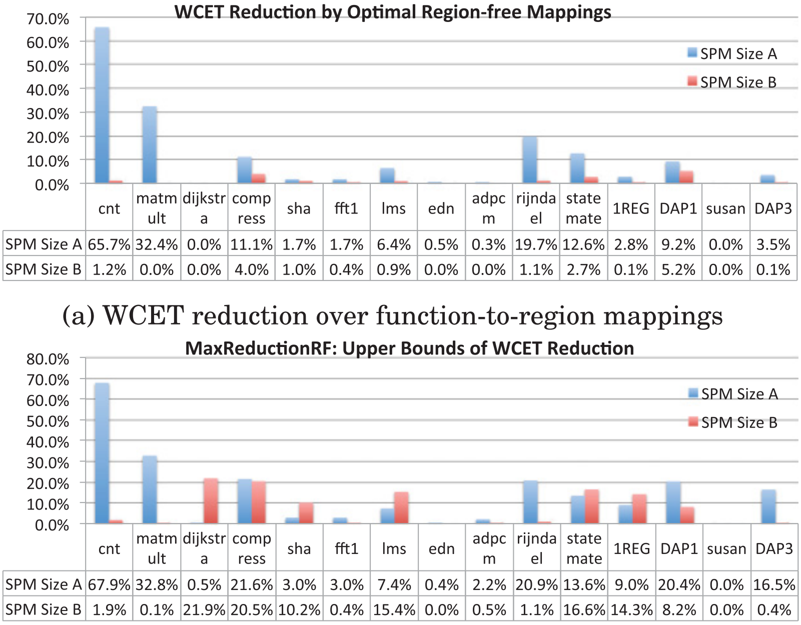
Figure 6 compares the WCET estimates for WMP, FMUM, FMUP, and BBL. These WCET estimates are normalized to the WCET estimates obtained with optimal mappings found by our ILP-based technique. WMP never underperforms FMUM and



(a) SPM size A

(b) SPM size B

Fig. 6. FMUM and FMUP are ACET-oriented function-level techniques, and BBL is a basic-block-level WCET-optimizing technique.



	cnt	matmult	dijkstra	compress	sha	fft1	lms	edn	adpcm	rijndael	statemate	1REG	DAP1	susan	DAP3
SPM Size A	65.7%	32.4%	0.0%	11.1%	1.7%	1.7%	6.4%	0.5%	0.3%	19.7%	12.6%	2.8%	9.2%	0.0%	3.5%
SPM Size B	1.2%	0.0%	0.0%	4.0%	1.0%	0.4%	0.9%	0.0%	0.0%	1.1%	2.7%	0.1%	5.2%	0.0%	0.1%

(a) WCET reduction over function-to-region mappings

	cnt	matmult	dijkstra	compress	sha	fft1	lms	edn	adpcm	rijndael	statemate	1REG	DAP1	susan	DAP3
SPM Size A	67.9%	32.8%	0.5%	21.6%	3.0%	3.0%	7.4%	0.4%	2.2%	20.9%	13.6%	9.0%	20.4%	0.0%	16.5%
SPM Size B	1.9%	0.1%	21.9%	20.5%	10.2%	0.4%	15.4%	0.0%	0.5%	1.1%	16.6%	14.3%	8.2%	0.0%	0.4%

(b) Upper bounds of WCET reduction

Fig. 7. Region-free mapping can reduce the WCET, but the amount of the reduction is limited.

FMUP, in all cases. As they are not optimized for WCETs, their performance is inconsistent; their WCET estimates range from 1.0 to 29.85. On the other hand, BBL outperforms even the ILP-based technique in most cases with smaller SPM size A, especially with smaller benchmarks. This is caused by the fundamental differences between function-level code management and basic-block-level code management. In function-level approaches, a function is loaded as a whole after checking whether the function is loaded in the SPM. When a loop contains function calls, there can be an overhead of checking the SPM state and then performing DMA operations in a loop. In all basic-block-level approaches including BBL, however, the loading never takes place in a loop but only at loop preheaders. In case of nested loops, the preheaders of the outermost loop are selected to avoid repetitive loading in a loop. This makes it very efficient when all loops are small enough to fit entirely in the SPM and most of the code is not in a loop, which is the case when BBL outperforms our approaches greatly. BBL struggles when many basic blocks in a loop have to be left in the main memory due to the SPM size restriction, for example, “1REG.” An instruction cache, if present, can reduce the overhead of accessing the main memory, but we leave such extensive comparison as future work. Overall, WMP is more consistent in reducing the WCET as its WCET estimates are kept under 1.06, compared to BBL, whose WCET estimates range from 0.52 to 9.85.

5.3. Function-to-Region Mappings Versus Region-Free Mappings

Figure 7(a) shows the reduction in WCET estimates by using region-free mappings compared to the function-to-region mappings found by the ILP.

In many cases, the reduction is less than 1%. The reason is as follows. For region-free mapping to be able to successfully avoid the reloading at a loading point v, the sum of the size of $fn(v)$ and the size of the largest function in the interference set $IS[v, fn(v)]$

should not be greater than the size of the SPM. If not, the largest function has to share SPM space with $fn(v)$, so region-free mapping cannot do anything. If at least these two functions can be assigned disjoint address ranges, all the inference can be removed by letting all functions in $IS[v, fn(v)]$ share SPM space with each other, but not with $fn(v)$.

Based on this insight, we calculate the upper bound of WCET reduction achievable by region-free mapping as follows. We first find the WCEP when the optimal function-to-region mapping is used. Then, we find all loading points v on the WCEP that satisfy these two conditions: (1) v is classified as Always-Miss, and (2) the sum of the size of $fn(v)$ and the size of the largest function in $IS[v, fn(v)]$ is smaller than the SPM size. These are the only loading points where region-free mapping can remove interference and reduce the WCET. We introduce a new value, **MaxReductionRF**, that is the sum of the loading costs at these vertices, which is a rough upper bound of WCET reduction possible by region-free mapping. Figure 7(b) shows MaxReductionRF normalized to the WCET in percentage. We can see that the WCET reductions in Figure 7(a) have a strong resemblance to the upper bounds in Figure 7(b) in many cases such as "cnt" or "matmult." For "edn" and "susan," there is no room to reduce the WCET with region-free mappings.

Even though region-free mapping has a larger solution space compared to function-to-region mapping, the ILP for region-free mapping has much fewer constraints than that for function-to-region mapping, which is shown in Table IV. We observed that linearizing the concept of binary variables in function-to-region mapping using Equations (15) and (17) causes the number of constraints to increase exponentially as the number of functions (and regions) increases. This difference translates to a great reduction in the ILP solving times with large benchmarks.

5.4. WCET Reduction over Caches

We evaluate our techniques in comparison with four-way set-associative caches with LRU replacement policy of the same size as the SPMs. We set the associativity to 4 like in many processors in embedded application, such as Renesas V850 or various ARM Cortex series, but we did not observe a significant difference in results with different associativity numbers. Although the LRU replacement policy is not commercially popular, it is considered to be the most predictable replacement policy [Guan et al. 2012].

The cache or SPM sizes for each benchmark are chosen as $2^{\lceil \log_2(l) \rceil}$ and $2^{\lceil \log_2(l) \rceil + 1}$, where l is the size of the largest function. The first is the smallest power of 2, greater than the largest function size, and the second is the next power of 2. A cache miss latency takes L (50) cycles (see the beginning of Section 5). Although the cache sizes for small benchmarks are much smaller than the instruction cache sizes in modern processors, this models the real-world situation where code sizes are usually much larger than the instruction cache size.

We implemented the cache analysis algorithm by Cullmann [2013], which is currently the state of the art and fixes an error of the traditional cache analysis used in the industry-leading aiT tool [Ferdinand 2004]. We run the must and may analyses [Ferdinand and Wilhelm 1999], and new persistence analysis, based on abstract interpretation, on our generated inlined CFGs. We do not perform virtual loop unrolling [Ferdinand and Wilhelm 1999], so the first iterations of loops are treated the same as the rest of the iterations. Although as Huber et al. discuss [2014], considering local execution scopes (e.g., a function or a loop) may help identify more first misses, we consider only the global execution scope (the whole program) as the persistence analysis algorithm itself does not discuss how to set persistence scopes. We use the same inlined CFGs for both caches and SPMs for fair comparison.

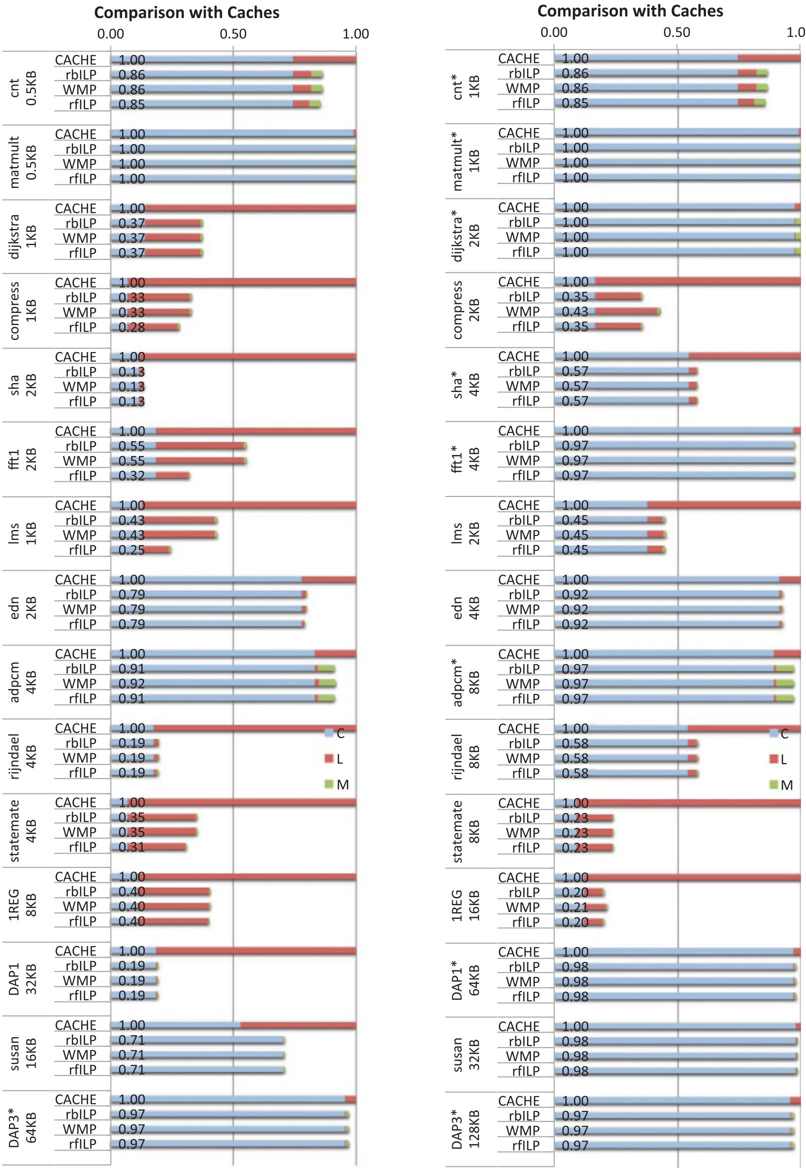


Fig. 8. C (blue), L (red), and M (green) denote computation time, L cache miss handling time or DMA time, and SPM management code execution time, respectively.

Figure 8 compares the WCET estimates. rbILP represents the region-based ILP from Section 4.1, and rfILP is the region-free ILP from Section 4.3. The cache/SPM size is shown after the name of each benchmark, and data values represent the WCET estimates normalized to the WCET estimates for caches (thus, it is always 1 for caches).

The WCET reduction is significant for most of the benchmarks where cache miss handling overhead (L) is very large. One main reason is the lack of link-time optimizations for caches. Instruction addresses are determined in the linking stage. Unless a WCET-aware code positioning technique [Falk and Kotthaus 2011; Um and Kim 2003;

Li et al. 2015] is used, the linker is generally not aware of the impact on the WCET of its decisions or the cache configuration in the target system. For this reason, function calls may cause many conflict misses in caches, whereas such side effects are actively avoided by code mapping in SPMs. When nested function calls exist in loops, the effect of cache conflict misses can be pronounced. Also, DMA operations for large functions take advantage of burst transfers [Huber et al. 2014]. In SPMs, a whole function is loaded at once with only one setup cost, whereas in caches, a cache miss penalty that includes the setup cost is incurred at every cache block boundary. In our experiments with larger cache block sizes such as 32 bytes and 64 bytes, we did not observe significant differences in the overall trends of the results.

Note that the cache size is larger than the total code size for many cases; for example, 65KB is larger than the total code size of "DAP3."[8] Such cases are marked with * after the name of the benchmark. In these cases, caches show very little cache miss penalties, so the WCET reduction by our techniques is not significant. Nevertheless, WCET estimates with our techniques are always less than or equal to the WCET estimates with caches.

6. RELATED WORK

SPM first gained its popularity in embedded processors mostly due to its advantages over caches in terms of energy consumption and die area [Banakar et al. 2002]. To utilize such benefits of SPM, many researchers initially proposed static management techniques, in which selected data or code is loaded into the SPM once before execution and remains in the SPM during the entire execution. Avissar et al. [2002] present an algorithm that considers global and stack data. Steinke et al. [2002b] consider global data along with instructions in functions or basic blocks. Both techniques are designed to reduce the overall energy consumption.

Compared to static management techniques, dynamic techniques can better exploit the localities in different parts of a large program by updating the SPM contents during runtime. This leads to overall improved performance [Udayakumaran et al. 2006]. Kandemir and Choudhary [2002] propose a dynamic data management scheme based on loop transformation and data placement to maximize the data reuse. Many techniques have focused on reducing energy consumption by dynamically copying instructions and/or global variables into the SPM [Steinke et al. 2002a; Verma et al. 2004].

Our approaches are also a type of dynamic management technique that focuses on program code. Several dynamic code management techniques have been proposed [Egger et al. 2006; Janapsatya et al. 2006], all of which aim to reduce the ACET or energy consumption. Unlike these techniques, our objective is to reduce the WCET.

Focusing on the time-predictable characteristics of SPMs, researchers proposed various management techniques that aim to reduce the WCET of a given program. Several techniques statically select variables [Suhendra et al. 2005] or instructions [Falk and Kleinsorge 2009; Prakash and Patel 2012]. There are also dynamic techniques that select variables [Deverge and Puaut 2007; Wan et al. 2012] or basic blocks [Puaut and Pais 2007; Wu et al. 2010] to be loaded into the SPMs during runtime. In this article, we present dynamic code management techniques, so this article is more closely related to the latter than the former. The main difference is at the granularity of the management; those techniques work in basic block level, whereas ours work in function level.

[8] For caches, we use the original binaries without inserting management code. In "matmult," 1KB is not larger than the total code size after inserting management code, but is larger than the original binary. Similarly, in "adpcm," 8KB is larger than the total code size only for caches.

Function-level dynamic code management techniques [Baker et al. 2010; Pabalkar et al. 2008; Jung et al. 2010; Bai et al. 2013] were originally proposed for software-managed memory architecture such as the Cell processor [Kahle et al. 2005]. Here, cores can only access their local SPMs, so every executed instruction must be copied from the main memory to the SPM. Basic-block-level approaches are not applicable in this architecture; they load only selected basic blocks to the SPM and leave the rest of the basic blocks in the main memory. Function-level approaches let every instruction be fetched from the SPM by loading a function before it is executed. This larger granularity can benefit from the characteristics of the burst mode DMA operations, as each DMA operation has a setup overhead. On the other hand, the function-level management can have drawbacks such as worse memory utilization due to fragmentation or fetching unnecessary code altogether, compared to the basic-block-level management. We leave the detailed performance comparison between different granularities as future work and focus on developing WCET-aware function-level code management techniques.

Compared to the previous function-level code management techniques, our approaches have mainly two differences. First, all previous approaches aim to reduce the ACET, not WCET. They calculate the overall overhead of mapping multiple functions into one region considering function sizes and calling patterns, so mappings that incur high overhead scenarios such as reloading functions in a loop are avoided. They, however, fail to consider the worst-case execution scenario of each function and the control flow within a function that does not have any function call. Our techniques find optimal mappings for reducing the WCET using inlined CFGs that comprehensively contain all information necessary to calculate the WCET. Second, all previous approaches use function-to-region mappings [Pabalkar et al. 2008] in problem formulation, so the solution quality is limited by the abstraction of SPM addresses with regions. We present a technique to map functions directly to addresses, which can further reduce the WCET.

In function-level code management, the largest function must fit in the SPM, which can limit its applicability. Kim et al. [2016] present a function-splitting technique to overcome this limitation. Splitting a function can not only enable using smaller SPM sizes but also improve performance by reducing memory footprints of functions.

Gracioli et al. [2015] present an extensive survey of worst-case related cache optimization techniques and cache analysis techniques. Cache locking [Plazar et al. 2012; Ding et al. 2014] and partitioning [Liu et al. 2010; Suhendra and Mitra 2008] can be used to lower WCETs by reducing conflict misses, but the granularity of these techniques is limited by blocks, lines, or ways, which may cause a waste of cache space [Whitham and Audsley 2009]. Just like our code mapping techniques, code positioning techniques can be used to avoid conflict misses among functions to reduce WCETs [Falk and Kotthaus 2011; Um and Kim 2003; Li et al. 2015].

Our interference analysis works similarly to the traditional may analysis based on abstract interpretation [Ferdinand and Wilhelm 1999] with the use of a union operation in join function. The semantics of the results are, however, the same as the must analysis in the sense that the interference sets are used to conservatively determine whether a function is guaranteed to be loaded (always-hit) when the interferences on all paths are considered. Although we can find first misses using initial loading points (see Table II), this is more pessimistic than the persistence analysis [Cullmann 2013] in terms of identifying first misses. For example, consider this code: $main()\{f_1(); f_2();$ `while(...)`$f_1(); \}$. In this example, *main* calls f_1 and f_2 in a row and then calls f_1 in a while loop. Assume that f_1 and f_2 do not call any other function. The call to f_1 in the while loop is not an initial loading point, but it can still be a first miss when f_1 does not share SPM space with *main* or f_2. Persistence analysis, on the other hand, can categorize the call to f_1 in the loop as first miss. Developing a more advanced analysis for tighter WCET bounds is part of our future work.

Lastly, predictable DRAM controllers [Kim et al. 2015] can help with bounding memory latencies and making DMA operations timing predictable.

7. CONCLUSION

SPM is a promising on-chip memory choice for real-time systems but needs explicit management. This article proposes three code management techniques that allocate SPM space for functions, with a goal of minimizing the WCET by avoiding DMA operations' overhead on the worst-case execution path. Two techniques are based on traditional function-to-region mappings, and the third technique maps functions directly to SPM addresses. Experimental results with several benchmarks including automotive control applications from industry show that our techniques are highly effective in reducing the WCET. The heuristic algorithm can find a mapping solution within a second for all benchmarks without increasing the WCET more than 6.5% compared to the solution found by the ILP. Results show that the region-free mapping technique can, in a few cases, further reduce the WCET compared to the optimal function-to-region mapping. Interestingly, although region-free mapping introduces a larger solution space, the actual ILP solving times are reduced significantly. Overall, the reduction in the WCET estimates ranges from 0% to 97% compared to previous approaches, and from 0% to 87% compared to the static analysis results of caches.

REFERENCES

Oren Avissar, Rajeev Barua, and Dave Stewart. 2002. An optimal memory allocation scheme for scratchpad-based embedded systems. *ACM Trans. Embed. Comput. Syst.* 1, 1 (Nov. 2002), 6–26.

Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. 2014. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.* 13, 4, Article 82 (March 2014), 37 pages.

Ke Bai, Jing Lu, Aviral Shrivastava, and Bryce Holton. 2013. CMSM: An efficient and effective code management for software managed multicores. In *Proc. of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–9.

Michael A. Baker, Amrit Panda, Nikhil Ghadge, Aniruddha Kadne, and Karam S. Chatha. 2010. A performance model and code overlay generator for scratchpad enhanced embedded processors. In *Proc. of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 287–296.

Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proc. of International Symposium on Hardware/Software Codesign (CODES)*. 73–78.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.

Stephen P. Bradley, Arnoldo C. Hax, and Thomas L. Magnanti. 1977. *Applied Mathematical Programming*. Addison-Wesley Publishing Company.

Giorgio C. Buttazzo. 2011. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* (3rd ed.). Springer Publishing Company.

Christoph Cullmann. 2013. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.* 12, 1s, Article 40 (March 2013), 25 pages.

Jean-Francois Deverge and Isabelle Puaut. 2007. WCET-directed dynamic scratchpad memory allocation of data. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*. 179–190.

Huping Ding, Yun Liang, and T. Mitra. 2014. WCET-centric dynamic instruction cache locking. In *Proc. of Design, Automation & Test in Europe (DATE)*. 1–6.

Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. 2006. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proc. of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 223–233.

Heiko Falk and Jan C. Kleinsorge. 2009. Optimal static WCET-aware scratchpad allocation of program code. In *Proc. of Design Automation Conference (DAC)*. 732–737.

Heiko Falk and Helena Kotthaus. 2011. WCET-driven cache-aware code positioning. In *Proc. of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 145–154.

Christian Ferdinand. 2004. Worst-case execution time prediction by static program analysis. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*. 125–127.

Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst.* 17, 2 (Nov. 1999), 131–181.

Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.* 48, 2, Article 32 (Nov. 2015), 36 pages.

Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. 2012. WCET analysis with MRU caches: Challenging LRU for predictability. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 55–64.

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks - Past, present and future. In *Proc. of International Workshop on Worst-Case Execution time Analysis (WCET)*. 136–146.

Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of International Workshop on Workload Characterization (WWC)*. 3–14.

Benedikt Huber, Stefan Hepp, and Martin Schoeberl. 2014. Scope-based method cache analysis. In *Proc. of International Workshop on Worst-Case Execution time Analysis (WCET)*, Vol. 39. 73–82.

Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. 2006. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proc. of Asia and South Pacific Design Automation Conference (ASPDAC)*. 612–617.

Seung Chul Jung, Aviral Shrivastava, and Ke Bai. 2010. Dynamic code mapping for limited local memory systems. In *Proc. of International Conference on Application-specific Systems (ASAP)*. 13–20.

James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. 2005. Introduction to the cell multiprocessor. *IBM J. Res. Dev.* 49, 4/5 (July 2005), 589–604.

Mahmut Kandemir and Alok Choudhary. 2002. Compiler-directed scratch pad memory hierarchy design and management. In *Proc. of Design Automation Conference (DAC)*. 628–633.

Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. 2009. *Data Flow Analysis: Theory and Practice*. CRC Press.

Hokeun Kim, David Broman, Edward A. Lee, Michael Zimmer, Aviral Shrivastava, and Junkwang Oh. 2015. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 317–326.

Youngbin Kim, Jian Cai, Yooseong Kim, Kyoungwoo Lee, and Aviral Shrivastava. 2016. Splitting functions in code management on scratchpad memories. In *Proc. of International Conference on Computer-Aided Design (ICCAD)*. 1–8.

Edward A. Lee. 2008. Cyber physical systems: Design challenges. In *Proc. of International Symposium on Real-Time Computing (ISORC)*. 363–369.

Fuyang Li, Mengying Zhao, and C. J. Xue. 2015. C3: Cooperative code positioning and cache locking for WCET minimization. In *Proc. of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 51–59.

Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. 2012. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proc. of International Conference on Computer Design (ICCD)*. 87–93.

Tiantian Liu, Yingchao Zhao, Minming Li, and C. J. Xue. 2010. Task assignment with cache partitioning and locking for WCET minimization on MPSoC. In *Proc. of International Conference on Parallel Processing (ICPP)*. 573–582.

Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. 2008. SDRM: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Proc. of the IEEE Conference on High Performance Computing (HiPC)*. 569–582.

Sascha Plazar, Jan C. Kleinsorge, Peter Marwedel, and Heiko Falk. 2012. WCET-aware static locking of instruction caches. In *Proc. of International Symposium on Code Generation and Optimization (CGO)*. 44–52.

Aayush Prakash and Hiren D. Patel. 2012. An instruction scratchpad memory allocation for the precision timed architecture. In *Proc. of Design, Automation & Test in Europe (DATE)*. 659–664.

Isabelle Puaut and Christophe Pais. 2007. Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison. In *Proc. of Design, Automation & Test in Europe (DATE)*. 1484–1489.

Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter Marwedel. 2002a. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proc. of International Symposium on System Synthesis (ISSS)*. 213–218.

Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. 2002b. Assigning program and data objects to scratchpad for energy reduction. In *Proc. of Design, Automation & Test in Europe (DATE)*. 409–415.

Vivy Suhendra and Tulika Mitra. 2008. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proc. of Design Automation Conference (DAC)*. 300–303.

Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. 2005. WCET centric data allocation to scratchpad memory. In *Proc. of Real-Time Systems Symposium (RTSS)*. 223–232.

Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. 2010. Scratchpad allocation for concurrent embedded software. *ACM Trans. Program. Lang. Syst.* 32, 4, Article 13 (April 2010), 47 pages.

Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.* 5, 2 (May 2006), 472–511.

Junhyung Um and Taewhan Kim. 2003. Code placement with selective cache activity minimization for embedded real-time software design. In *Proc. of International Conference on Computer-Aided Design (ICCAD)*. 197–200.

Manish Verma, Lars Wehmeyer, and Peter Marwedel. 2004. Dynamic overlay of scratchpad memory for energy minimization. In *Proc. of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 104–109.

Qing Wan, Hui Wu, and Jingling Xue. 2012. WCET-aware data selection and allocation for scratchpad memory. In *Proc. of Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 41–50.

Jack Whitham and Neil Audsley. 2009. Implementing time-predictable load and store operations. In *Proc. of International Conference on Embedded Software (EMSOFT)*. 265–274.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages.

Hui Wu, Jingling Xue, and Sri Parameswaran. 2010. Optimal WCET-aware code selection for scratchpad memory. In *Proc. of International Conference on Embedded Software (EMSOFT)*. 59–68.

Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. 2014. FlexPRET: A processor platform for mixed-criticality systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 101–110.

Received April 2016; revised February 2017; accepted February 2017