WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores

Yooseong Kim*†, David Broman*‡, Jian Cai†, and Aviral Shrivastaval*†
* University of California, Berkeley, {yooseongkim, davbr, aviral}@berkeley.edu
† Arizona State University, {yooseong.kim, jian.cai, aviral.shrivastava}@asu.edu
‡ Linköping University, david.broman@liu.se

***Abstract*—Software Managed Multicore (SMM) architectures have advantageous scalability, power efficiency, and predictability characteristics, making SMM particularly promising for real-time systems. In SMM architectures, each core can only access its scratchpad memory (SPM); any access to main memory is done explicitly by DMA instructions. As a consequence, dynamic code management techniques are essential for loading program code from the main memory to SPM. Current state-of-the-art dynamic code management techniques for SMM architectures are, however, optimized for average-case execution time, not worst-case execution time (WCET), which is vital for hard real-time systems. In this paper, we present two novel WCET-aware dynamic SPM code management techniques for SMM architectures. The first technique is optimal and based on integer linear programming (ILP), whereas the second technique is a heuristic that is sub-optimal, but scalable. Experimental results with benchmarks from Mälardalen WCET suite and MiBench suite show that our ILP solution can reduce the WCET estimates up to 80% compared to previous techniques. Furthermore, our heuristic can, for most benchmarks, find the same optimal mappings within one second on a 2GHz dual core machine.**

I. Introduction

In real-time [1] and cyber-physical [2] 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. Computing safe upper bounds of a task's *worst-case execution time* (WCET) is essential to guarantee the absence of missed deadlines.

Real-time systems are becoming more and more complex with increasing performance demands. Performance improvements in recent processor designs have mainly been driven by the multicore paradigm because of power and temperature limitations with single-core designs [3]. Some recent real-time systems architectures are moving towards multicore [4] or multithreaded [5], [6] designs. However, coherent caches, which are popular in traditional multicore platforms, are not a good fit for real-time systems. Coherent caches make WCET analysis difficult and result in pessimistic WCET estimates [7].

This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), the Swedish Research Council (#623-2011-955), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: Medium: Timing Centric Software), and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota).

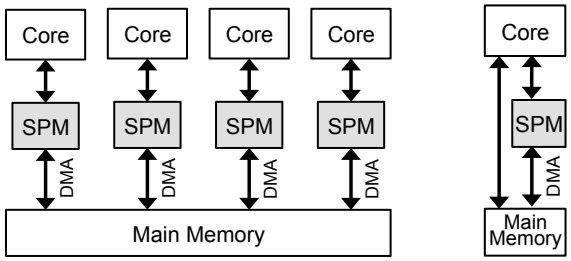


Fig. 1. (a) SMM architecture vs. (b) traditional architecture with SPM. Cores cannot access main memory directly in SMM architecture. All code and data must be present in SPM at the time of execution.

SMM (Software Managed Multicore) architectures [8], [9] are a promising alternative for real-time systems. In SMM, each core has a scratchpad memory (SPM), so-called local memory, as shown in Fig. 1(a). A core can only access its SPM in an SMM architecture, as opposed to the traditional architecture in Fig. 1(b) where a core can access both main memory and SPM with different latencies. Accesses to the main memory must be done explicitly through the use of direct memory access (DMA) instructions. The absence of coherency makes such architectures scalable and simpler to design and verify compared to traditional multicore architectures [3]. An example of an SMM architecture is the Cell processor that is used in Playstation 3 [10].

If all code and data of a task can fit in the SPM, the timing model of memory accesses is trivial: each load and store always take a constant number of clock cycles. However, if all code or data does not fit in the SPM, it must be dynamically managed by executing DMA instructions during runtime. Dynamic code management strongly affects timing and must consequently be an integral part of WCET analysis.

In traditional architectures that have SPMs, cores can directly access main memory, though it takes a longer time to access main memory than the SPM. In such architectures, the question is what to bring in the SPM to reduce the WCET of a task. This approach is not, however, feasible in SMM architectures because all relevant code must be present in the SPM at the time of execution. For this reason, existing WCET-aware dynamic code management techniques for SPMs [11], [12]—which select part of the code to be loaded in the SPM and keep the rest in the main memory—are not applicable in SMM architecture.

There exists previous work on developing dynamic code

management techniques for SMM architecture [9], [11], [13]–[15]. Such techniques manage code at the granularity of functions; the code of a function is loaded as a whole using a static function-to-region mapping. The mapping specifies where in the SPM each function is loaded. If several functions map to the same region, functions are replaced during execution. At every function call and return, it is determined if the function to execute is loaded in a specific region on the SPM. If the function to be executed is not present in this region, the processor has to stall while the function is being loaded.

All previous techniques focus on finding mappings for *average-case execution time* (ACET). As a result, they cannot find the best mapping for WCET and often show counterintuitive results; the WCET estimate may increase when larger SPM sizes are used.

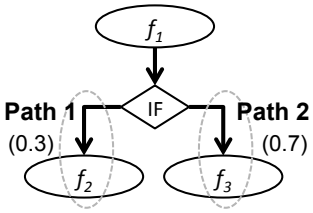
To solve this problem, we present two novel WCET-aware mapping techniques. The first technique can find the optimal mapping solution for WCET and is based on *integer linear programming* (ILP). Because the number of function-to-region mapping choices grows exponentially with the number of functions in the program, this technique does not scale to large programs. Thus, we also present a polynomial-time heuristic that is scalable, but suboptimal. We evaluate our approach on several benchmarks from the Mälardalen WCET benchmark suite [16] and MiBench suite [17]. More specifically, we make the following contributions:

- We present an algorithm together with an ILP formulation that can compute a safe upper bound of the WCET of a program, given a function-to-region mapping (Section III).
- We present an ILP formulation that finds an optimal mapping, leading to the lowest possible WCET estimate for a given SPM size (Section IV-A).
- We present a new polynomial-time WCET-aware heuristic algorithm that finds a sub-optimal mapping (Section IV-B).
- We evaluate our approach in comparison with three previous mapping algorithms that are designed for ACET, one from [14] and two from [13]. We compute safe upper bounds of the WCET using our solutions and the solutions of previous approaches (Section V).

II. Motivation

Previous approaches try to find mappings that reduce the overall amount of DMA transfers in a program [11], [13]–[15]. Such approaches can effectively improve ACET of a program, but do not always reduce WCET. In the following simple motivating example, we demonstrate the differences between optimizing for ACET and WCET.

The example program in Fig. 2(a) has three functions: f_1, f_2, and f_3. The main function f_1 has two paths, calling functions f_2 and f_3 in paths 1 and 2, respectively. The probability of the program to take each path is determined by the branch probability of the `if`-statement in f_1, here assumed to be 0.3 for path 1 and 0.7 for path 2. The execution time of each path, ignoring the cost for DMA transfer to load functions, is shown in Fig. 2(b). The cost for loading each function is assumed to be the same as the size of the function, shown in Fig. 2(c).



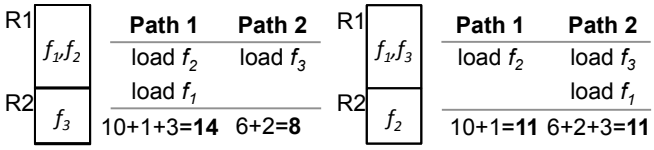
(a) An example program

Path 1	10
Path 2	6

(b) Path execution time (without DMA cost)

f_1	3
f_2	1
f_3	2

(c) Function size



(d) Mapping A (e) Mapping B

	Mem. usage	ACET	WCET estimate
A	5	$14 * 0.3 + 8 * 0.7 = 9.8$	$max(14, 8) = 14$
B	4	$11 * 0.3 + 11 * 0.7 = 11$	$max(11, 11) = 11$

(f) ACET and WCET comparison

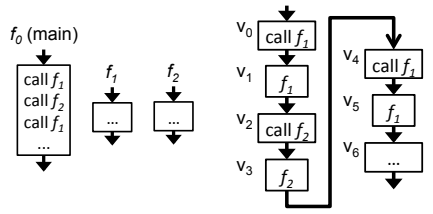
Fig. 2. Mapping solutions for ACET and WCET can be different. The solution for ACET is A, whereas the solution for WCET is B.

Assuming the size of the SPM is 5, multiple functions must share the same region. Two feasible mapping solutions are: mapping f_1 and f_2 to the same region and mapping f_1 and f_3 to the same region. Fig. 2(d) and 2(e) show the sequence of DMA transfers on each path for these mapping choices. For instance, when f_1 and f_2 are mapped to the same region, f_1 has to be loaded again when f_2 returns because f_2 replaced the code of f_1.

Fig. 2(f) shows ACET and WCET for each of the mappings. Considering path probabilities, the best ACET can be achieved by A. The overall amount of DMA transfers is reduced by mapping f_1 and f_3 into different regions because it can avoid evicting the largest function, f_1, on the more frequently executed path[1] (path 2 in this case). For this reason, previous approaches will always try to map f_1 and f_3 to different regions. However, the WCET of the program is better with mapping B because the WCET of the two paths is lower in mapping B than in A.

This example shows that optimizing for ACET may not

[1]For simplicity, all previous approaches abstractly model the cost of DMA transfers on a path as the product of the path probability and the sum of the sizes of the functions that replace each others on the path. Thus, with mapping A, the replacement only occurs on path 1, and its cost is $0.3 \times (3+1) = 1.2$. Similarly, with mapping B, the replacement only occurs on path 2, and its cost is $0.7 \times (3+2) = 3.5$.



(a) CFGs of functions (b) Inlined CFG

	$IS[v, func(v)]$
v_0	$\emptyset$
v_1	$\{f_0\}$
v_2	$\{f_1\}$
v_3	$\{f_0, f_1\}$
v_4	$\{f_2\}$
v_5	$\{f_0, f_2\}$
v_6	$\{f_1\}$

(c) Interference sets

Fig. 3. An inlined CFG the global call sequence and the control flow by inlining CFGs of functions when they called.

always result in a good WCET. Previous mapping techniques try to optimize the ACET and should therefore not be used in hard real-time applications. In this paper, we present mapping techniques that explicitly optimize the WCET of a program.

III. Computing a Safe WCET Bound for a Given Function-to-Region Mapping

In this section, we present how to compute a safe upper bound of the WCET for a given function-to-region mapping. We take a graph representation of the program as input, analyze the graph, and generate an ILP that is finally used for computing a safe upper bound of the WCET.

A. Inlined CFG

As input to our analysis method, we use a variant of a *control flow graph* (CFG), called an *inlined* CFG. In an inlined CFG, CFGs of functions are combined to represent the global call sequence and the control flow. An example is depicted in Fig. 3(a) and Fig. 3(b). We call it *inlined* because a separate copy of the CFG of a function is inlined each time the function is called. For instance, f_1 appears twice at v_1 and v_5 in Fig. 3(b). The benefit with such a representation is that it keeps context information in case a function is called from different locations in the program. The limitation is that recursive functions cannot be represented, something acceptable in an embedded real-time context.

More formally, let $G = (V, E, v_s, v_t, F, func)$ be an inlined CFG. V is the set of vertices, representing basic blocks. The set of edges is defined as $E = \{(v, w)$: there is a path from v to w due to control flow or function call or function return, where $v, w \in V\}$. Vertices v_s and v_t represent the starting basic block and the terminal basic block, respectively. F is the set of functions in the program, and $func : V \rightarrow F$ is a mapping stating that given a vertex v returns the function $func(v)$ that v belongs to. For instance, in Fig. 3(b), $func(v_0) = f_0$, $func(v_1) = f_1$, $func(v_2) = f_0$, $func(v_3) = f_2$, $func(v_4) = f_0$, $func(v_5) = f_1$, and $func(v_6) = f_0$.

B. Interference Analysis

To compute a safe upper bound of the WCET, we predict whether or not a DMA transfer is needed in the worst-case at function call program points and function return program points. This is done by computing *interference sets* at every function call and return. We define interference set as follows.

Algorithm 1 Interference analysis

```
1: function InterferenceAnalysis(G)
2:     initialize IS[v, f] as ∅ for all v ∈ G.V and f ∈ G.F
3:     repeat
4:         prevIS ← IS
5:         IS ← GetIS(G.v_s, IS, G)
6:     until IS = prevIS
7: return IS
```

Algorithm 2 GetIS

```
1: function GetIS(v, IS, G)
2:     if v is not marked visited then
3:         mark v as visited
4:         for each u where (u, v) ∈ G.E do
5:             for each f ∈ G.F do
6:                 if f ≠ G.func(u) then
7:                     IS[v, f] ← IS[v, f] ∪ IS[u, f] ∪ {G.func(u)}
8:     for each w where (v, w) ∈ G.E do
9:         IS ← GetIS(w, IS, G)
10: return IS
```

Definition 1 (Interference Set)**.** *Let $G = (V, E, v_s, v_t, F, func)$ be an inlined CFG. Then, for a basic block $v \in V$ and a function $f \in F$, the* interference set *$IS[v, f] \subseteq (F - \{f\})$ is the set of functions that may have been loaded since the last time f was loaded in all possible call sequences. If f was never loaded before, $IS[v, f]$ is the set of functions that may have been loaded since the start of the program.*

The interference set information implies the following; if any function $g \in IS[v, f]$ is mapped to the same region as f, f may have been evicted from the region at point v in the program. Thus, if v makes a function call to f, in the worst-case, f has to be reloaded by a DMA transfer. The table in Fig. 3(c) shows the interference sets $IS[v, func(v)]$ for the example in Fig. 3(b). For instance, at v_5, f_1 is loaded for the second time. Function f_1 has already been loaded at v_1, but then f_0, f_2 and again f_0 are loaded since then at v_2, v_3, and v_4, respectively. If either f_0 or f_2 are mapped to the same region as f_1, f_1 must have been evicted and has to be reloaded at v_5. Thus, $IS[v_5, func(v_5)] = \{f_0, f_2\}$.

Algorithm 1 shows our algorithm for computing interference sets. It iteratively calls `GETIS` (Algorithm 2) from the root node of the graph, until it reaches a fixed point where IS stops changing. Function `GETIS` computes $IS[v, f]$ for all $v \in V$ and for all $f \in F$, by recursively propagating the history of function executions. We show the resulting interference sets calculated by the algorithm in Table I.

Note that at each $v \in V$, $IS[v, func(v)]$ provides sufficient information to determine whether or not $func(v)$ needs to be reloaded at v. However, the algorithm calculates $IS[v, f]$ for all

TABLE I. Interference sets for the example in Fig. 3

	$func(v)$	$IS[v, f_0]$	$IS[v, f_1]$	$IS[v, f_2]$
v_0	f_0	$\emptyset$	$\emptyset$	$\emptyset$
v_1	f_1	$\emptyset$	$\{f_0\}$	$\{f_0\}$
v_2	f_0	$\{f_1\}$	$\emptyset$	$\{f_0, f_1\}$
v_3	f_2	$\emptyset$	$\{f_0\}$	$\{f_0, f_1\}$
v_4	f_0	$\{f_2\}$	$\{f_0, f_2\}$	$\emptyset$
v_5	f_1	$\emptyset$	$\{f_0, f_2\}$	$\{f_0\}$
v_6	f_0	$\{f_1\}$	$\emptyset$	$\{f_0, f_1\}$

$f \in F$ because it keeps the history of function executions with regard to each function. For example, $IS[v_5, f_0]$ is an empty set because f_0 is executed at the only immediate predecessor v_4, but $IS[v_5, f_1]$ is $\{f_0, f_2\}$ because f_0 and f_2 are executed after v_1 which is the last time f_1 is executed. Similarly, $IS[v_5, f_2]$ is $\{f_0\}$ because only f_0 is executed since f_2 is executed at v_3. We mark $IS[v, func(v)]$ in each row in Table I, which matches the abridged table in Fig. 3(c).

Finally, we state and prove the important property of convergence for the inference algorithm.

Theorem 1 (Convergence)**.** *Algorithm 1 converges to a unique solution in finite number of iterations.*

Proof: To prove that the algorithm converges to a unique solution in a finite number of iterations (termination) we make use of the standard Knaster-Tarski fixpoint theorem, saying that if we have complete lattice A and an order-preserving map $g : A \to A$, then there exists a least fix point x such that $x \leq g(x)$. We need to show that (i) function GETIS can be mapped to a function $g : A \to A$, such that A is a complete lattice and (ii) that the resulting function g is order-preserving. We observe that GETIS is always called with the same start node $G.v_s$ and graph G (line 5 in Algorithm 1). Consequently, for (i), we only need to show that the domain of parameter *IS* is isomorphic to a complete lattice. We can directly conclude that the power set of $\{f_{v,h} | f \in IS[v, h], v \in V, h \in F\}$ is a complete lattice ordered by inclusion. For (ii), we first observe that GETIS terminates because each vertex in G is only visited once and the set of edges $G.E$ and the set of functions F are finite. Moreover, the interference set *IS* can either become larger or remain the same each time the interference set is updated, but not become smaller (line 7 of Algorithm 2). Hence, GETIS and consequently g are order preserving. Finally, according to Knaster-Tarski fixpoint theorem, there is a least fix point which is by definition unique and because A is finite, Algorithm 1 will research this fixed point in finite number of iterations. ■

C. ILP Formulation for WCET Analysis

We model the worst-case execution path similarly to Suhendra *et al.* [18]. Variables in the ILP are written in capital letters, and constants are in small letters.

1) Modeling the Worst-case Execution Path: For each vertex $v \in V$, a variable W_v denotes the WCET starting from v to the end of the program, v_t. W_v is modeled by the following constraints.

$$\forall (v, w) \in E,\ W_v \geq W_w + C_v \tag{1}$$

For v_t, $W_{v_t} = C_{v_t}$ because v_t does not have any successor. C_v represents the execution time of v, which we explain later in this section.

The WCET of the whole program is computed by minimizing the WCET of the start node:

$$\textbf{\textit{minimize}} \quad W_{v_s} \tag{2}$$

Loops are specially treated as follows. Without loss of generality, we require that every loop in a loop nest has a unique loop header. Starting from the innermost loop in each loop nest, we first remove its back edge and substitute the loop body with a supernode that represents the WCET of each loop as Fig. 4 shows. Loop entry edges and loop exit edges are removed from the graph, which makes the loop body disconnected from the rest of the graph. A new edge is inserted for each loop entry edge and loop exit edge to connect the supernode with the rest of the graph. The cost of the supernode is set as the WCET of the loop. The WCET of the loop is calculated isolatedly from the rest of the program using constraint (1) because the loop body becomes disconnected as we remove loop entry edges and loop exit edges.

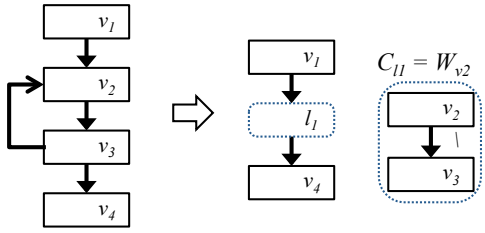


Fig. 4. For each loop, the back edge is removed, and the vertices in the loop body are substituted by a supernode that represents the WCET of the loop.

In the example shown in Fig. 4, a supernode l_1 is inserted for the loop $\{v_2, v_3\}$. Loop entry edge (v_1, v_2) and loop exit edge (v_3, v_4) are substituted with (v_1, l_1) and (l_1, v_4), respectively. The following constraints are generated for the example.

$$W_{v_1} \geq W_{l_1} + C_{v_1}$$
$$W_{l_1} \geq W_{v_4} + C_{l_1}$$
$$W_{v_4} = C_{v_4}$$

The cost of a supernode $C_{l_1} = W_{v_2}$ can be calculated as follows.

$$W_{v_2} \geq W_{v_3} + C_{v_2}$$
$$W_{v_3} = C_{v_3}$$

2) Modeling the Cost of Each Vertex (C_v): The cost of v, C_v, is modeled using two constants n_v and $comp_v$ as follows.

$$C_v = n_v \cdot comp_v + L_v \tag{3}$$

where n_v is the number of times v is executed, and $comp_v$ is the computation cost of v which is the execution time assuming *func*(v) is loaded.

L_v is a variable that represents the execution time involved with loading *func*(v) into SPM. If v is not where any function call or return happens, or *func*(v) does not need to be loaded, L_v will be zero. The number of times *func*(v) needs to be loaded is implicitly considered in L_v, which means that the value of L_v will be the time to load *func*(v) multiplied by the number of times it needs to be loaded. Before we explain how to model L_v, let us consider when a function needs to be loaded.

Considering that functions are loaded only at function calls or returns, we define one property of vertices, called *context-changing*, to denote such vertices with a call or a return.

Definition 2 (Context-changing vertex)**.** *A vertex is called context-changing if it is the beginning of a new function or the target of a return instruction.*

The following constant denotes if a vertex is context-changing or not. For all vertex $v \in V$,

$$cc_v = \begin{cases} 1 & \text{if } \exists (u,v) \in E,\ \textit{func}(v) \neq \textit{func}(u) \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

Also, even if v is context-changing, loading $\textit{func}(v)$ is only necessary when $\textit{func}(v)$ is not loaded in its region, and the contents of the region depends on the given function-to-region mapping and the interference set. Therefore, we define the following constants to feed information to the ILP. When R is the set of all regions, the function-to-region mapping is specified as follows. For all function $f \in F$, $g \in F$ and region $r \in R$,

$$m_{f,g,r} = \begin{cases} 1 & \text{if both } f \text{ and } g \text{ are mapped to } r \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

The interference sets are specified as follows. For all vertex $v \in V$ and function $f \in F$,

$$i_{f,v} = \begin{cases} 1 & \text{if } f \in IS[v, \textit{func}(v)] \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

Using the above constants, we can model L_v as follows. For all function $f \in F$ and region $r \in R$,

$$L_v \geq n_v \cdot cc_v \cdot m_{\textit{func}(v),f,r} \cdot i_{f,v} \cdot dma_{\textit{func}(v)} \quad (7)$$

where $dma_{\textit{func}(v)}$ is a constant that denotes the time to load function f to SPM by DMA.

cc_v checks if function loading can happen at v. $m_{\textit{func}(v),f,r} \cdot i_{f,v}$ checks if $\textit{func}(v)$ and f are mapped to the same region, r, and at the same time, f is included in v's interference set. Thus, $m_{\textit{func}(v),f,r} \cdot i_{f,v}$ is 1 only when $\textit{func}(v)$ is mapped to region r and may have been evicted from r by function f.

We assume that the execution of a program starts with an empty SPM, which means that regardless of the given mapping, a function has to be loaded at least once when it is called for the first time during execution. The most accurate way of modeling this is to enumerate all paths and find the first occurrence of functions on each path, but exhaustive path enumeration is not scalable because the number of paths increases exponentially to the program size. In this paper, we topologically sort the inlined CFG (with back edges removed), and find the first occurrences of functions in the sorted graph. Considering that a DMA transfer will take place at least once at such vertices for initial loading, we can rewrite constraint (7) as follows. For all functions $f \in F$ and regions $r \in R$,

$$L_v \geq \begin{cases} \text{If } v \text{ is the first occurrence of } \textit{func}(v) \\ \quad dma_{\textit{func}(v)} + \\ \quad (n_v - 1) \cdot (cc_v \cdot (m_{\textit{func}(v),f,r} \cdot i_{f,v}) \cdot dma_{\textit{func}(v)}) \\ \text{Otherwise,} \\ \quad n_v \cdot (cc_v \cdot (m_{\textit{func}(v),f,r} \cdot i_{f,v}) \cdot dma_{\textit{func}(v)}) \end{cases} \quad (8)$$

By minimizing the objective function, L_v can be set to one of the three values: $0, dma_{\textit{func}(v)}$, or $n_v \cdot dma_{\textit{func}(v)}$, It will be zero when v is not context-changing or when we can guarantee that $\textit{func}(v)$ is loaded at v. It will be $dma_{\textit{func}(v)}$ when v is the first occurrence of $\textit{func}(v)$, so it will be loaded once and remain loaded from now on. L_v can be $n_v \cdot dma_{\textit{func}(v)}$ otherwise.

Solving the above ILP, we can obtain the WCET estimate of a program using the given mapping, as the objective value W_{v_s}.

IV. Function-to-Region Mapping for WCET

In this section, we present our two techniques of finding a mapping for WCET. The first one is optimal and based on ILP, and the second one is a polynomial-time heuristic, but sub-optimal.

A. ILP Formulation to Find an Optimal Mapping

We can extend the ILP from the previous section to explore all feasible mappings. Then, thanks to the minimizing objective function, we can find the mapping that achieves the lowest WCET estimate and the WCET estimate itself.

Since the solver explores all feasible mappings, the set of regions R cannot be fixed because different mappings can have different number of regions. We initially assume there are the same number of regions as the number of functions because that is the maximum number of region when we can map each function into a different region. Mapping solutions that use smaller number of regions can be modeled by having regions to which no function is mapped.

We model a mapping decision using the following variables for all functions $f \in F$ and regions $r \in R$.

$$M_{f,r} = \begin{cases} 1 & \text{if } f \text{ is mapped to } r \\ 0 & \text{Otherwise} \end{cases} \quad (9)$$

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

$$\forall f \in F, \quad \sum_{r \in R} M_{f,r} = 1 \quad (10)$$

Secondly, the total required space for the mapping is not greater than the given SPM size.

$$\forall f \in F, \quad S_r \geq m_{f,r} \cdot s_f \quad (11)$$

$$\text{spm\_size} \geq \sum_{r \in R} S_r \quad (12)$$

where spm_size is the size of the given SPM, and s_f is the size of function f. S_r is a variable that represents the size requirement of the region r which is calculated by taking the maximum of the size of every function mapped to r.

In the previous formulation, the mapping is specified by constants $m_{f,g,r}$, and L_v is modeled using $m_{f,g,r}$. We need to express mappings in the same form as $m_{f,g,r}$, but now the mapping solutions that are explored by the solver are expressed with variables $M_{f,r}$. We use variables $M_{f,g,r}$ whose values are defined as follows.

$$M_{f,g,r} = \begin{cases} 1 & \text{if } M_{f,r} = 1 \wedge M_{g,r} = 1 \\ 0 & \text{otherwise.} \end{cases} \quad (13)$$

The logical conditions can be linearized by the following approach from [19]. For each region r, $M_{f,g,r}$ can be 1 only when both $M_{f,r}$ and $M_{g,r}$ are 1, which means the sum of two terms will be greater than 1. Otherwise, the sum will be less than or equal to 1, and $M_{f,g,r}$ needs to be 0 in such case. The following constraints ensure that $M_{f,g,r}$ is 1 if and only if both $M_{f,r}$ and $M_{g,r}$ is 1.

$$2 \cdot (1 - M_{f,g,r}) + M_{f,r} + M_{g,r} > 1 \quad (14)$$

$$2 \cdot M_{f,g,r} + 1 \geq M_{f,r} + M_{g,r} \quad (15)$$

Then, the constraint (16) is rewritten using $M_{\mathit{func}(v),f,r}$ instead of $m_{\mathit{func}(v),f,r}$ as follows.

$$L_v \geq \begin{cases} \text{If } v \text{ is the first occurrence of } \mathit{func}(v) \\ \quad dma_{\mathit{func}(v)} + \\ \quad (n_v - 1) \cdot (cc_v \cdot (M_{\mathit{func}(v),f,r} \cdot i_{f,v}) \cdot dma_{\mathit{func}(v)}) \\ \text{Otherwise,} \\ \quad n_v \cdot (cc_v \cdot (M_{\mathit{func}(v),f,r} \cdot i_{f,v}) \cdot dma_{\mathit{func}(v)}) \end{cases} \quad (16)$$

Solving the modified ILP with the above, we can obtain the lowest WCET estimate of the given program achievable by function-level dynamic code management. The mapping that achieves such estimate can be obtained by looking at the values of $M_{f,r}$ of the solution.

B. Our Heuristic

The ILP-based mapping technique presented in Section IV-A can find an optimal solution, but is not scalable. For example, the number of functions in a program is one of the factors that have the greatest impact on the time to solve the ILP. The number of mapping choices (the solution space of the ILP) grows exponentially with the number of functions in a program. In our experiments with benchmarks, it takes less than a second for 'matmult' which has 6 functions but more than a week for 'adpcm' which has 17 functions.

To solve this problem, we present a polynomial-time heuristic technique which 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) [13]. 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 decrease, 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 which estimates the WCET of the program.

Algorithm 3 shows its pseudocode. Given an inlined CFG G, the interference sets *IS* and the size of SPM, it returns a mapping M. A mapping solution, M, is represented by an integer array. We assign a unique number to each function from 1 to $|G.F|$. Similarly, each region is also identified by a unique id from 1 to $|G.F|$. The number of regions is 1 when all functions are mapped to the same region, and $|G.F|$ when each function is mapped to a separate region. Then, any mapping solution can be represented by an array, for example, if function 1 is mapped to region 2, $M[1] = 2$.

Algorithm 3 A heuristic to find a mapping for WCET

```
1: function FindMapping(G, IS, SPMSIZE)
2:     Remove all back edges in G
3:     Topologically sort G
4:     M_m ← Merge(G, IS, SPMSIZE)
5:     M_p ← Partition(G, IS, SPMSIZE)
6:     if Cost(G, IS, M_m) < Cost(G, IS, M_p) then
7:         M ← M_m
8:     else
9:         M ← M_p
10: return M
11:
12: function Cost(G, IS, M)
13:     for each v ∈ G.V do                          ▷ initialize dist
14:         dist[v] ← 0
15:     for each v ∈ G.V in topological order do
16:         w ← comp_v                                ▷ the weight of v
17:         if cc(v) = 1 then
18:             for each g ∈ IS[v, func(v)] do
19:                 if M[g] = M[func(v)] then
20:                     w ← w + dma_func(v)           ▷ add the cost for DMA
21:         dist[v] ← max_{(u,v)∈G.E} dist[u] + w
22: return dist[G.v_t]
```

We first remove all back edges and topologically sort the graph (line 2-3). This is needed for calculating cost of a mapping during the course of algorithm, by calling function Cost. We find two mapping solutions by merging and partitioning (line 4-5) which are shown in Algorithm 4 and Algorithm 5. After comparing their costs (line 6-9), we return the one with the smaller cost for the final solution.

The function Cost calculates the length of the longest path in G. We use an integer array $dist$ for storing the longest distance from $G.v_s$ to v for all $v \in G.V$. Since G is already topologically sorted, for any v, the value $dist$ for all predecessors are already calculated. For each vertex v, $dist[v]$ is set to the sum of maximum $dist$ value of all predecessors and its own weight w (line 21). The algorithm returns the $dist$ value of the terminal node v_t which is the longest path from v_s to v_t. The weight of v is assigned according to the mapping M and the interference sets *IS*. If vertex v is not context-changing, the weight of v is set to $comp_v$ (line 5). If v is context-changing, and if any function interferes with it (line17-19), the cost of transferring $\mathit{func}(v)$ by DMA is added to the weight (line 20).

Algorithm 4 and Algorithm 5 show our two algorithms for searching mapping solutions, Merge and Partition, respectively. In both algorithms, we use function Size. Size(M) calculates the memory size requirement of mapping M, which can be calculated by summing up the size of the largest function in each region. We omit its algorithm for brevity.

In Algorithm 4, Merge starts with mapping each function to a separate region (line 2-3). Then, in the while loop at line 4-21, we take every combination of two regions (line 7-8), and create a temporary mapping M' where two regions are merged (line 9-13). We check the cost of the temporary mapping (line 14) and keep a record of the best two regions to be merged (line 15-18). After trying all combinations, we update the current mapping by merging the best pair of regions (line 19-21).

Algorithm 4 Search the solution space by merging

```
1: function MERGE(G, IS, SPMSIZE)
2:     for i = 1 to |G.F| do                          ▷ initial mapping M
3:         M[i] ← i
4:     while SIZE(M) > SPMSIZE do
5:         c ← COST(G, IS, M)
6:         bc ← c                                     ▷ the best cost found so far
7:         for r1 = 1 to |G.F| − 1 do
8:             for r2 = r1 + 1 to |G.F| do
9:                 for f = 1 to |G.F| do              ▷ merge r1 and r2
10:                    if M[f] = r2 then
11:                        M'[f] = r1
12:                    else
13:                        M'[f] = M[f]
14:                tc ←COST(G, IS, M')
15:                if tc < bc then                    ▷ record the best pair, r1 and r2
16:                    br1 = r1
17:                    br2 = r2
18:                    bc ← tc
19:        for f = 1 to |G.F| do                      ▷ merge br1 and br2
20:            if M[f] = br2 then
21:                M[f] = br1
22: return M
```

Algorithm 5 Search the solution space by partitioning

```
1: function PARTITION(G, IS, SPMSIZE)
2:     for i = 1 to |G.F| do                          ▷ initial mapping M
3:         M[i] ← 1
4:     nr ← 1                                         ▷ the number of region is 1
5:     while nr ≤ |G.F| do
6:         c ← COST(G, IS, M)
7:         bc ← c                                     ▷ the best cost found so far
8:         for f = 1 to |G.F| do
9:             or ← M[f]                              ▷ record f's original region
10:            for r = 1 to nr + 1 do
11:                if M[f] = r then
12:                    continue
13:                M[f] ← r                           ▷ move f back to r
14:                tc ← COST(G, IS, M)
15:                if SIZE(M) ≤ SPMSIZE then
16:                    if tc < bc then                ▷ record the best f and r
17:                        bf ← f
18:                        br ← r
19:                        bc ← tc
20:                M[f] ← or                          ▷ move f back to or
21:        if bc < c then                             ▷ move bf to br
22:            M[bf] ← br
23:            if br = nr + 1 then
24:                nr ← nr + 1                        ▷ update the number of region
25:        else
26:            break
27: return M
```

In Algorithm 5, PARTITION starts with mapping all functions to one region (line 2-4). We move one function to a different region per iteration of the while loop at line 5-26. For every function, we try moving it to a different region or creating a new region $nr + 1$ for it (line 10-13). We keep a record of the best combination of a function to be moved, bf, and the new region for it, br (line 15-19). After trying all functions, we update the current mapping by moving function bf to region br (line 21-22). br being greater than nr means that we just made a new region, so we increase the number of regions (line 23-24). We stop when the cost of the mapping does not get improved.

The while loop in MERGE at line 4 takes at most $|G.F|-1$ times because the number of regions decreases by one at every iteration. The for-loop nest at line 7-9 takes $|G.F|^3$ times, and the time complexity of COST is $O(|G.V| \cdot |G.F|)$ since it visits every vertex only once, and an interference set can have all functions in the worst-case. Thus, the time complexity of MERGE is $O(|G.F|^4 \cdot |G.V|)$. PARTITION has the same time complexity which can be calculated similarly.

V. EVALUATION

We evaluate our approach by comparing it with our previous mapping algorithms for ACET: FMUM, FMUP [13], and *simultaneous determination of regions and mapping* (SDRM) [14]. Their cost functions estimate the overall amount of data transfer for a given mapping. As explained in Section IV-B, FMUM and FMUP start from a simple initial mapping solution and iteratively improve it so that the cost of the mapping decreases. SDRM calculates the cost for each function, which is the product of the function size and the number of execution, and iteratively assigns a separate region starting from the function with the highest cost.

A. Experimental Setup

We use various benchmarks from the Mälardalen WCET benchmark suite [16] and MiBench suite [17]. We compile benchmarks for ARM v4 ISA [20] and generate inlined CFGs from the disassembly of the binaries. Loop bounds are obtained by profiling.

There are 31 benchmarks in Mälardalen suite and 29 benchmarks in MiBench suite. Among these, we exclude ones that are recursive since inlined CFGs cannot represent recursive programs as we discussed in Section III. We also exclude benchmarks that have less than six functions for more effective comparison of mapping algorithms as the number of functions forms the solution space in a function-to-region mapping problem[2]. After excluding 1 for recursion and 22 for the number of functions, we use all of the remaining benchmarks, which are 8, from Mälardalen suite. MiBench suite is in general much larger in size and more complicated,

[2]We only consider functions in the user code, and library function calls are treated as normal instructions that take one cycle.

TABLE II. BENCHMARKS USED IN OUR EVALUATION

	Total code size (bytes)	Largest function size (bytes)	Number of functions	Benchmark Suite
susan	51440	9968	19	MiBench
rijndael	23136	8028	7	MiBench
statemate	11120	3568	8	Malardalen
adpcm	10564	2896	17	Malardalen
edn	5232	1972	9	Malardalen
sha	4092	1276	8	MiBench
compress	3936	1056	9	Malardalen
lms	3696	900	8	Malardalen
fft1	3304	1836	6	Malardalen
dijkstra	2244	1052	6	MiBench
matmult	1632	472	6	Malardalen
cnt	1368	384	6	Malardalen

and we were not able to generate the inlined CFGs of 6 benchmarks due to the presence of recursion or function pointers[3], and 19 due to the complexity of compiled binaries[4]. We use the remaining 4 benchmarks in our evaluation. Table II shows our benchmarks[5].

To simplify computing WCET estimates, we assume that every instruction takes exactly one cycle as it is on processors designed for timing predictability, such as PRET [5].

We model the cost of loading x bytes into SPM by DMA as $n \cdot (46 + \lceil x/4 \rceil)$, where n is the number of cores. When n is 1, this is the same as the model for a single core by Whitham *et al.* [21]. We take the most conservative way of considering the inference of other cores, i.e., multiplying the cost of a single core by the number of cores. We assume the number of cores is 4 in our experiments, although our comparison results of mapping algorithms are independent of this number.

The correctness of our WCET estimation is verified by running selected benchmarks on gem5 simulator [22]. We modified the simulator so that every instruction takes one cycle, and added an SPM component and DMA instructions so that binaries with our management code can run on the simulator. The number of cycles each benchmark took on the simulator was always less than or equal to the WCET estimates we obtained by analysis.

Table II show the size of each benchmark after we insert DMA instructions and the code for checking the state of regions. Since the sizes of benchmarks are largely different, we use different SPM sizes for different benchmarks. This is to make our comparison experiments more effective. For example, if we use the same SPM size for all benchmarks, for some small benchmarks, the whole code can fit in the SPM. In this case, any mapping technique would assign a separate region to every function. Also, if we choose small enough SPM size to make the mapping problem interesting for small benchmarks, such SPM size is smaller than the size of the largest function for larger benchmarks, which makes the mapping problem infeasible because the largest function must fit in the SPM. Therefore, we use two different SPM sizes for each benchmark: 60% and 75% of the code size.

B. Experimental Results

We evaluate our mapping techniques in comparison with previous mapping techniques. For each mapping technique, we obtain mapping solutions for benchmarks and then calculate the WCET estimates of benchmarks using the ILP from Section III. Then, the WCET estimates are normalized to the WCET estimates obtained with the mappings found by solving our ILP formulation from Section IV-A. The mapping solutions found by the ILP are optimal except for two cases, both for 'adpcm' when the SPM size is 60% of the code size and 75% of the code size[6]. The ILP solver did not finish for these two cases after a week on 2Ghz dual-core machine with 2GB of main memory. Instead, we set a time limit of three hours and used the best solutions found by the time limit.

In Fig. 5, we compare the normalized WCET estimates in three different SPM sizes. On the x-axis, the number of functions in the benchmark and the SPM size are written under the name of each benchmark. The maximum value of the y-axis is limited at 3, so any columns that go beyond that are truncated. We also show the geometric means of different mapping techniques in the right most columns for each SPM size. Overall, our heuristic can reduce the WCET estimate up to 69% compared to FMUM (in 'compress' when the SPM size is 60% of the code size), 79% compared to FMUP (in 'fft1' when the SPM size is 75% of the code size), and 79% compared to SDRM (in 'adpcm' and 'fft1' when the SPM size is 75% of the code size). Note that all techniques find the same mapping for 'edn' because it has a very simple pattern of function calls where all functions are called only once in main function in sequence.

The normalized WCET estimates are always greater than or equal to 1, and this means that no technique performs better than the ILP. Even for the time-limited ILP used for 'adpcm', the ILP outperforms all other techniques.

The normalized WCET estimates of our heuristic are 1 in most cases, and this means that it can find the same solution as the ILP in most cases. However, there are cases where our heuristic cannot find the same solution as the ILP such as 'adpcm'. Even in such cases, our heuristic can find better mapping solutions than all previous techniques except for 'adpcm' when the SPM size is 60% of the code size. Our heuristic cannot always find a better solution because it does not try all mappings and can get stuck in a local optimum.

Note that all previous heuristics suffer from the same problem since they do not explore the whole solution space either. Thus, we cannot conclude that any mapping technique can always find better mappings than others, except that our ILP-based technique can always find the optimal mapping. However, the strength of our heuristic is in WCET-awareness. Our heuristic's optimization is always focused on the WCET of the program, whereas previous heuristics try to optimize the whole program. This fact can be demonstrated by the counterintuitive results we observe in our experiments in which the WCET estimate of a benchmark increases with previous mapping techniques for larger SPM sizes. For example, in case of FMUP, the WCET estimate of 'matmult' is increased from 564094 to 752456 (33% increase) when the SPM size is increased from 992B to 1232B. This means that previous techniques do not have any notion of the WCET, so their optimization may improve the ACET but can end up increasing the WCET.

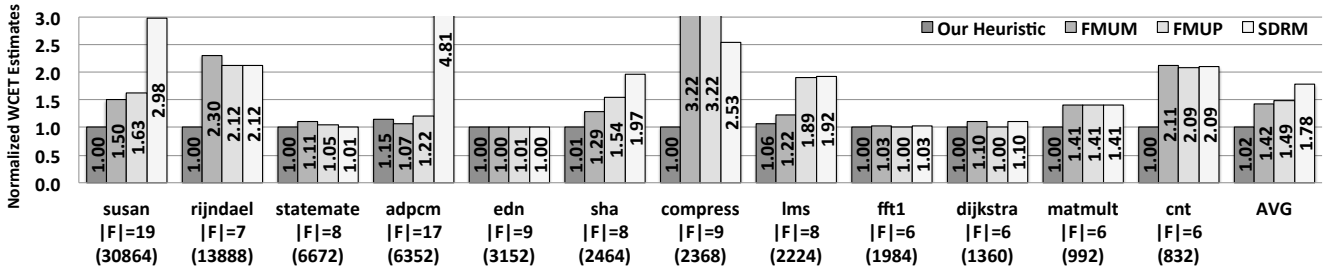
Our heuristic mapping algorithm is intended for the scalability rather than the optimality. To obtain a mapping for a benchmark using our heuristic algorithm, we need to perform the interference analysis and then run our heuristic algorithm.

[3] We were not able to identify callee functions in compile-time.

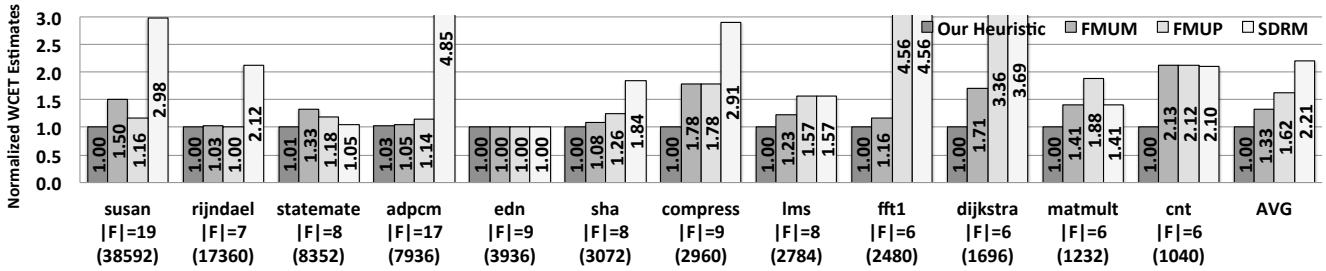
[4] We generate inlined CFGs by parsing disassembly files, and we were not able to parse the disassembly files of these benchmarks within a limited time frame due to their irregular patterns, which is only a technical difficulty and not a limitation of our approach.

[5] 'dijkstra' has a recursive function call for printing out results, so we commented out the recursive function call. The core algorithm of the benchmark is not recursive.

[6] We can see that 'adpcm' is not the largest in terms of code size nor the number of functions in Table II. The complexity of an ILP problem is dependent on various parameters such as the number of functions, number of basic blocks, the number of paths, the function sizes and the size of the SPM.



(a) Normalized WCET Estimates when the SPM size is 60% of the code size



(b) Normalized WCET Estimates when the SPM size is 75% of the code size

Fig. 5. WCET estimates are normalized to the WCET estimate obtained by the mapping found by the ILP. Under the name of each benchmark, we show the number of functions and the size of SPM in bytes in parentheses. In most cases, our heuristic can find the optimal solution that are found by solving the ILP, and reduce the WCET estimates significantly compared to previous mapping techniques.

We also need to solve the ILP from Section III to get a WCET estimate for the given mapping. The total elapsed time for all these three steps was less than one second for all benchmarks on 2Ghz dual-core machine with 2GB of main memory.

Note that the ILP solver was also able to find the optimal mapping in less than one minute for all benchmarks except for 'adpcm' and 'sha' — it took 104 minutes for 'sha' when the SPM size is 75% of the code size, but for 'adpcm' the solver did not finish even after a week for both memory sizes. Interestingly, even for these two, the best objective value that the ILP solver can find did not get improved any more after twenty minutes, and the solver kept iterating on the same objective value. As it can be seen in Fig. 5(a) and Fig. 5(b), even those suboptimal solutions are better than mappings found by any other mapping techniques. This means that solving the ILP with reasonable time limits (e.g. twenty minutes) can be a good heuristic method itself.

VI. Related Work

Previous code management techniques have been focusing on reducing average-case execution time or energy consumption [23]–[25]. These approaches are not suitable for hard real-time systems because improving the ACET does not always improve the WCET and can even increase the WCET, as discussed in Section II.

There are approaches for reducing the WCET [26], [27], but these are static. Thus, the contents in the SPM are fixed before execution and no changes occur during runtime. Static approaches cannot exploit the locality of large programs because they cannot load all code with locality before execution. Dynamic approaches, on the other hand, can update the contents in the SPM during runtime, thus better exploiting the locality in different parts of a program. Puaut and Pais [11] propose a dynamic management technique that selects basic blocks to be loaded in the SPM and finds reload points where such basic blocks are loaded at runtime. Wu *et al.* [12] propose an optimal algorithm for non-nested loops and a heuristic for loop nests.

All of the above techniques for reducing the WCET are for traditional architectures with SPM where cores can directly access main memory. They find a set of basic blocks to be loaded in the SPM, and those basic blocks not loaded in the SPM have to be accessed directly from main memory which will be slower than the SPM. Note that such techniques are not usable in SMM architecture where cores cannot directly access main memory.

In SMM architecture, all code must be present in SPM at the time of execution. Function-level dynamic code management techniques [9], [13]–[15], originally proposed for Cell processor [10] which is an example of SMM architectures, are the only applicable here and thus our closest related work. All previous approaches are, however, optimized for reducing the ACET and do not consider the WCET. Our approaches can not only reduce the WCET but also find the optimal mapping for WCET. A point worth noting here is that function-level dynamic code management techniques are not exclusively for SMM architectures, which means that our technique can also be used in traditional architectures with SPM.

SMM architectures were originally proposed for power-efficiency [8], [9], not for predictability. Some recent processor designs for real-time applications are, however, using an SMM style of memory model, as a way to achieve fine-grained timing predictability. For instance, FlexPRET [6] has both hard and soft real-time hardware threads, where the hard real-time threads can only access the SPM. As a consequence, our approach is directly applicable for this kind of processor when the hard real-time threads have code sizes that cannot fit in the SPM. Our work may also be applicable in WCET-aware

compilers [28] and can potentially, after further extensions, be used as an important component of a precision timed infrastructure [29].

VII. Conclusion

SMM architectures are promising for real-time embedded systems due to power efficiency, scalability, and predictability. In SMM architectures, all code must be present in SPM before execution because cores cannot directly access main memory. Previous WCET-aware code management techniques for SPMs in traditional architectures are not applicable in SMM architecture because they require cores to have a direct access to main memory. All previous dynamic code management techniques for SMM architectures are optimized for reducing the ACET of a program. In this paper, on the other hand, we present techniques for optimizing the WCET. Our ILP-based technique finds an optimal solution for WCET, whereas our heuristic is sub-optimal but scalable. We evaluate our techniques in comparison with three previous mapping techniques. Experimental results with benchmarks from Mälardalen WCET suite and MiBench suite show that our approaches can reduce the WCET estimates significantly compared to previous approaches.

Acknowledgments

We are grateful to all members in the PRET project at UC Berkeley for discussions and feedback. We would like to give special thanks to Ke Bai for help with our experiments and to Michael Zimmer for proofreading our early manuscripts. We are also very thankful to Edward Lee and Christopher Brooks for their generous support that enabled our research collaboration at UC Berkeley.

References

[1] G. C. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. Springer, 2011.

[2] D. Broman, P. Derler, and J. C. Eidson, "Temporal issues in cyber-physical systems," *Journal of Indian Institute of Science*, vol. 93, no. 3, pp. 389–402, 2013.

[3] G. Blake, R. G. Dreslinski, and T. Mudge, "A Survey of Multicore Processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, 2009.

[4] M. Paolieri, J. Mische, S. Metzlaff, M. Gerdes, E. Quiñones, S. Uhrig, T. Ungerer, and F. J. Cazorla, "A Hard Real-time Capable Multi-core SMT Processor," *ACM Transactions on Embedded Computing System*, vol. 12, no. 3, pp. 79:1–79:26, Apr. 2013.

[5] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance," in *Proceedings of the International Conference on Computer Design*, 2012, pp. 87–93.

[6] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A Processor Platform for Mixed-Criticality Systems," in *Proceedings of the Real Time and Embedded Technology and Applications Symposium*, 2014.

[7] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A Unified WCET Analysis Framework for Multi-core Platforms," in *Proceedings of the Real Time and Embedded Technology and Applications Symposium*, 2012, pp. 99–108.

[8] J. Lu, K. Bai, and A. Shrivastava, "SSDM: Smart Stack Data Management for software managed multicores (SMMs)," in *Proceedings of the Design Automation Conference*, 2013, pp. 1–8.

[9] K. Bai, J. Lu, A. Shrivastava, and B. Holton, "CMSM: An Efficient and Effective Code Management for Software Managed Multicores," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2013, pp. 1–9.

[10] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 589 –604, july 2005.

[11] I. Puaut and C. Pais, "Scratchpad Memories vs Locked Caches in Hard Real-Time Systems: A Quantitative Comparison," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, 2007, pp. 1–6.

[12] H. Wu, J. Xue, and S. Parameswaran, "Optimal WCET-Aware Code Selection for Scratchpad Memory," in *Proceedings of the International Conference on Embedded Software*, 2010, pp. 59–68.

[13] S. Jung, A. Shrivastava, and K. Bai, "Dynamic Code Mapping for Limited Local Memory Systems," in *Proceedings of the International Conference on Application-specific Systems Architectures and Processors*, July 2010, pp. 13–20.

[14] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee, "SDRM: Simultaneous Determination of Regions and Function-to-region Mapping for Scratchpad Memories," in *Proceedings of the International Conference on High Performance Computing*, 2008, pp. 569–582.

[15] M. A. Baker, A. Panda, N. Ghadge, A. Kadne, and K. S. Chatha, "A Performance Model and Code Overlay Generator for Scratchpad Enhanced Embedded Processors," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2010, pp. 287–296.

[16] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks - Past, Present and Future," in *Proceedings of the Workshop on Worst-Case Execution Time Analysis*, July 2010.

[17] 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 *Proceedings of the International Workshop on Workload Characterization*, 2001, pp. 3–14.

[18] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET Centric Data Allocation to Scratchpad Memory," in *Proceedings of International Real-Time Systems Symposium*, 2005, pp. 10 pp.–232.

[19] D.-S. Chen, R. G. Batson, and Y. Dang, *Applied Integer Programming: Modeling and Solution*. Wiley, 2010.

[20] D. Seal, *ARM Architecture Reference Manual*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 2000.

[21] J. Whitham and N. Audsley, "Implementing Time-Predictable Load and Store Operations," in *Proceedings of the International Conference on Embedded Software*, 2009, pp. 265–274.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[23] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic Management of Scratch-pad Memory Space," in *Proceedings of the Design Automation Conference*, 2001, pp. 690–695.

[24] L. Li, L. Gao, and J. Xue, "Memory Coloring: A Compiler Approach for Scratchpad Memory Management," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2005, pp. 329–338.

[25] M. Verma, L. Wehmeyer, and P. Marwedel, "Cache-Aware Scratchpad Allocation Algorithm," in *Proceedings of the Design, Automation and Test in Europe Conference Exhibition*, 2004, pp. 21 264–.

[26] J. C. Kleinsorge, "WCET-Centric Code Allocation for Scratchpad Memories," Diploma Thesis, TU Dortmund, 2008.

[27] S. Plazar, J. C. Kleinsorge, P. Marwedel, and H. Falk, "WCET-Aware Static Locking of Jnstruction Caches," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2012, pp. 44–52.

[28] H. Falk and P. Lokuciejewski, "A Compiler Framework for the Reduction of Worst-case Execution Times," *Real-Time Systems*, vol. 46, no. 2, pp. 251–300, 2010.

[29] D. Broman, M. Zimmer, Y. Kim, H. Kim, J. Cai, A. Shrivastava, S. A. Edwards, and E. A. Lee, "Precision Timed Infrastructure: Design Challenges," in *Proceedings of the Electronic System Level Synthesis Conference*, 2013.