WCET-Aware Data Selection and Allocation for Scratchpad Memory

Qing Wan Hui Wu Jingling Xue

School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{qingwan,huiw,jingling}@cse.unsw.edu.au

Abstract

In embedded systems, SPM (scratchpad memory) is an attractive alternative to cache memory due to its lower energy consumption and higher predictability of program execution. This paper studies the problem of placing variables of a program into an SPM such that its WCET (worst-case execution time) is minimized. We propose an efficient dynamic approach that comprises two novel heuristics. The first heuristic iteratively selects a most beneficial variable as an SPM resident candidate based on its impact on the k longest paths of the program. The second heuristic incrementally allocates each SPM resident candidate to the SPM based on graph coloring and acyclic graph orientation. We have evaluated our approach by comparing with an ILP-based approach and a longest-path-based greedy approach using the eight benchmarks selected from Powerstone and Mälardalen WCET Benchmark suites under three different SPM configurations. Our approach achieves up to 21% and 43% improvements in WCET reduction over the ILP-based approach and the greedy approach, respectively.

Categories and Subject Descriptors C.3 [*Real-time and embedded systems*]; B.3.3 [*Memory Structures*]: Worst-case analysis; D.3.4 [*Programming Languages*]: Compilers—Optimization

General Terms Algorithms, Performance

Keywords Worst-Case Execution Time, Scratchpad Memory Allocation, Graph Coloring, Acyclic Graph Orientation

1. Introduction

Large off-chip memory latency is a major obstacle for achieving the high performance of modern processors. The classical solution is to use cache memory to store code and data to reduce memory access latency. However, cache memory introduces three major problems. Firstly, cache memory consumes a significant amount of energy due to its tag store and associated circuitry. Secondly, cache memory makes it difficult to compute the WCET of a program. Thirdly, data cache makes it hard to handle data hazards occurring on modern RISC processors with deep pipelines. In contrast, SPM does not have a tag store and associated circuitry as in cache memory. Thus, it consumes less energy than cache memory and makes it much easier to compute the WCET of a program. In addition, since SPM is managed by the compiler, the compiler knows if each variable access is in the SPM or the off-chip memory. Therefore, it is much easier to compute the WCET of a program and handle data hazards. As a result, SPM is an attractive alternative to cache memory in embedded systems. SPM has been used in many processors and DSPs. Examples are NVIDIA's PhysX PPU (physics processing unit) and the Cell jointly developed by Sony, IBM, and Toshiba.

An optimizing compiler plays a major role in making efficient use of SPM. The compiler selects variables and code as SPM residents and inserts instructions to transfer selected data and code between off-chip memory and SPM. There are two major challenges in SPM management. The first one is to find an optimal subset of data and code as SPM residents. The second one is to allocate as many SPM resident candidates to the SPM as possible. Extensive research has been conducted to find solutions to these two major challenges. Many SPM management approaches have been proposed. All the existing approaches can be classified into two categories: static allocation [1, 6, 10, 11, 22] and dynamic allocation [2, 3, 12–15, 20, 24, 27]. In the case of static allocation approaches, once an SPM resident is loaded into the SPM, its space in the SPM cannot be allocated to other SPM residents during the execution of the program. As a result, static allocation approaches can lead to low SPM utilization for some programs. Dynamic approaches reason about the live ranges of the SPM residents. Two SPM residents can share a section of the SPM if their live ranges do not overlap. Therefore, better SPM utilization can be achieved.

Most of the existing approaches for SPM management aim at minimizing either the average execution time or the average energy consumption of a program. Thus, they are not suitable for real-time embedded systems. In a real-time embedded system, the WCETs of the tasks running on the system can have a major impact on the schedulability of the entire task set. As a result, it is desirable to minimize the WCET of each task.

Suhendra et al. [22] study the problem of selecting variables as SPM residents such that the WCET of a task is minimized. They have investigated three approaches, a longest-path-based greedy approach, a branch and bound approach and an ILP-based approach. These approaches are all static without reasoning about the live ranges of the variables considered, leading to low SPM utilization for some programs. In addition, both the branch and bound approach and the ILP-based approach have an exponential-time complexity. Therefore, they can be slow for large programs as discussed by the authors of that work. Finally, the longest-path-based approach selects a variable with the most occurrences on the longest path. On modern RISC processors, the compiler may hide memory latency by scheduling other instructions to execute during

LCTES 2012 June 12–13, 2012, Beijing, China.

a memory access. As a result, allocating a variable with the most occurrences into the SPM does not necessarily reduce the length of the longest path in the program.

Deverge and Puaut [2] present a dynamic approach to data selection and SPM allocation that aims at minimizing the WCET of a task. Their approach iteratively solves an ILP (Integer Linear Programming) problem to find a set of variables with the largest impact on WCET reduction and allocates these variables to the SPM by using a heuristic. Since the ILP problem itself is NP-complete, this ILP-based approach can be slow for large programs as discussed by the authors of this work.

Puaut and Pais [20] propose a static approach for selecting basic blocks of a task as SPM residents. Their approach iteratively selects a basic block with the most occurrences on the longest path of the task as an SPM resident. Falk and Kleinsorge [4] introduce an ILP-based, static approach to the problem of allocating basic blocks of a program to SPM such that the WCET of the program is minimized. Recently, Wu et al. [28] propose an optimal static code selection algorithm to minimize the WCET of a program.

In the context of caches, the problem of locking code of a program into the instruction cache such that the WCET of a program is minimized has been studied. Falk at el. [5] present a WCET-aware greedy approach to selecting a set of functions to be locked into the instruction cache. Their approach iteratively computes the gain on the WCET reduction of each function and selects the function with the highest gain to be locked into the instruction cache. Liu et al. [17] extend the work in [5] by using an ILP formulation to select a function with the highest gain to be locked. Plazar et al. [19] further extend the work in [17]. They propose a new ILP-based approach that is able to model the intra-function worst-case execution path.

Vera et al. [25] propose an approach to locking data in the data cache such that the worst-case memory performance (WCMP) of a program can be estimated in a safe, tight and fast way. In order to obtain predictable cache behavior, their approach locks the cache for those parts of the code where the static analysis fails and loads the cache with data likely to be accessed. Vera et al. [26] propose a WCET-aware framework for locking data cache for multitasking systems. Their framework combines cache partitioning and dynamic cache locking to provide worst-case performance estimates in a safe and tight way for multitasking systems. Their framework partitions the data cache among all the tasks to eliminate intertask cache interferences and uses static cache analysis and cache-locking mechanisms to ensure that all intratask conflicts, and consequently, memory access times, are exactly predictable.

In this paper, we study the problem of placing data variables (global and stack variables) of a program into an SPM to minimize the WCET of the program. We propose an efficient polynomial-time approach to solving this problem by employing two novel heuristics, one for variable selection and one for SPM allocation. Specifically, this paper makes the following contributions:

1. We introduce a new heuristic for selecting variables as SPM resident candidates. Our heuristic iteratively selects a variable as an SPM resident candidate that has the most impact on the k longest paths of the program, where k is a positive integer.
2. We introduce a new heuristic for allocating selected variables to the SPM by using graph coloring and acyclic graph orientation.
3. We have evaluated our approach by comparing with an ILP-based approach [2] and a longest-path-based greedy approach [22]. For the eight benchmarks selected from the Powerstone suite [21] and the Mälardalen WCET Benchmark suite [8] under three different SPM configurations, our approach achieves up to 21% and 43% improvements in WCET reduction over the ILP-based approach and the greedy approach, respectively.

The rest of this paper is organized as follows. Section 2 presents our processor model and program representation. Section 3 examines a link between SPM allocation and graph coloring coupled with acyclic graph orientation. Section 4 describes our heuristics for variable selection and SPM allocation. Section 5 presents our experimental results and analysis. Section 6 concludes the paper.

2. Processor Model and Program Representation

We assume that the target processor uses SPM to replace data cache. The SPM occupies a contiguous memory space, starting at address 0 with m bytes in total. Given a program to be executed on the processor, our objective is to select a subset of variables of the program and allocate them to the SPM such that the WCET of the program is minimized. Each access (read/write) to the SPM takes one processor cycle, and each access to the off-chip memory takes c cycles. For illustration purposes, we consider C programs only. However, our approach is applicable to any imperative languages.

We make use of the CFG (*control flow graph*) of a program in computing its WCET. Given a program P, its CFG is a weighted directed graph $\mathcal{C} = (V, E, W)$, where $V = \{v_1, v_2, \cdots, v_n\}$ is the set of the basic blocks in P, $E = \{(v_i, v_j) \mid v_j$ is directly control dependent on $v_i\}$ is the set of control flow edges, and $W = \{w(v_i, v_j) \mid w(v_i, v_j)$ is an edge weight on (v_i, v_j) denoting the execution time of v_j in the case where v_j is executed immediately after $v_i\}$ labels each edge with an execution time. Throughout this paper, we assume that all execution times are in processor cycles.

Modern RISC processors are pipelined. On a RISC processor, the execution time of a basic block may vary due to data hazards and branch penalty [9]. Therefore, we use an edge weight instead of a node weight to denote the execution time of each basic block.

Let us consider an example. For the C function, *foo()*, given in Listing 1, its CFG is shown in Figure 1a.

```
int foo(){
  int a, b, c=0;
  for(a=0; a<30; a++){ /*Loop1*/
    if(a>15){
      for(b=1; b<50; b++) /*Loop2*/
        if(b>5)
          c=c+a*b+10;
        else
          c=c+2*a*b;
    }else{
      for(b=10; b<=20; b++) /*Loop3*/
        c=c+a*a*b+2;
    }
  }
  return c;
}
```

Listing 1: An example function *foo()*.

In order to facilitate computing the lengths of k-longest paths needed by our variable selection heuristic, we break down the CFG of a program into a set of *weighted DAGs* (directed acyclic graphs). Each function or loop has its own weighted DAG. Given a loop or function x of a program, x is recursively represented as a weighted DAG $\mathcal{D}(x) = (V, E, W)$, where $V = \{v_i \mid v_i$ is a basic block, or a loop immediately nested in x, or a function directly called in $x\}$, E is a set of control flow edges, and W is a set of edge weights. There are three types of nodes, basic block nodes, loop nodes and function nodes. A basic block node represents a basic block in x. A loop node denotes a loop that is immediately nested in x. A function node denotes a function directly called in x. Here, a loop refers to all the code of this loop, including other loops nested in it and all

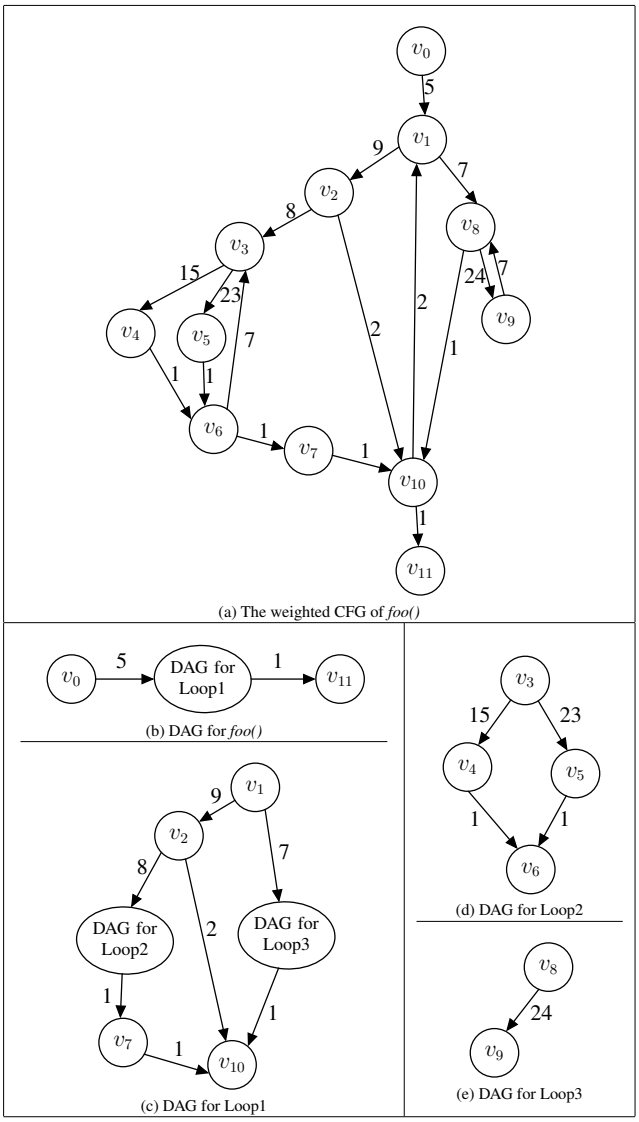

Figure 1: The weighted CFG and DAGs of *foo()*.

the functions called in it. When constructing a weighted DAG for a loop, we ignore all the back edges in the CFG of the loop. Without loss of generality, we assume that each weighted DAG has only one source node and only one sink node. The weighted DAGs for the function *foo()* given in Figure 1 are shown in Figures 1b – 1e.

Given a program P, its *call graph*, denoted CallGraph(P), is defined in the standard manner. Its nodes represent the set of its functions and its directed edges represent calling relationships between functions such that each edge (f, g) indicates that f calls g directly. We assume that the program has no recursive calls. As a result, the call graph is a DAG.

Given a program P, its *interference graph* is an undirected graph $\mathcal{G} = (V, E)$, where V is the set of all the variables (or live ranges) and $E = \{(v_i, v_j) \mid$ the live ranges of v_i and v_j overlap$\}$. In a C program, if two variables have disjoint live ranges, they can be allocated to two overlapping areas in the SPM. For example, if any two local variables belong to two different functions that are not on the same path of the call graph, they can share a section of the SPM. As discussed later, our approach will start by performing live-range splitting [13] to ensure that our overall solution is dynamic.

Given an interference graph $\mathcal{G} = (V, E)$, a legal vertex coloring ω of $\mathcal{G}$ is a function $\omega : V \rightarrow C$ such that for each $(v_i, v_j) \in E$, $\omega(v_i) \neq \omega(v_j)$ holds, where C is a set of colors.

3. SPM Allocation and Acyclic Graph Orientation

There are two main problems in WCET-aware SPM management for a program, variable selection and SPM allocation. The former is to select a set of variables as SPM resident candidates such that the WCET of the program is minimized. The latter is to find an optimal allocation scheme for allocating all the variables selected to the SPM.

In order to maximize SPM utilization, we need to place as many variables as possible into the SPM. The presence of interference constraints between variables complicates the SPM allocation problem. Graph coloring has been used as a central paradigm for register allocation in modern compilers. Recently, graph coloring has also been adopted in SPM allocation [13–15, 29]. Building on prior work, our WCET-aware approach exploits a link between SPM allocation and graph coloring coupled with acyclic graph orientation to reduce the WCET of a program.

Given an undirected graph, we obtain an *acyclic orientation (graph)* of the graph if we assign a direction to each of its edges such that the resulting directed graph has no cycles, The following theorem describes the relationship between an acyclic oriented interference graph and the SPM allocation problem.

Theorem 1. *Consider a program with a set V of variables, an acyclic oriented graph $\mathcal{O}$ of the interference graph $\mathcal{G}$ of the program, and an SPM of size m. If the longest path length of $\mathcal{O}$ is not greater than m, then all the variables can be allocated to the SPM.*

Proof. We construct an SPM allocation scheme as follows:

1. For each source node v_i, $start\_addr(v_i)$ of v_i is 0.
2. For each non-source node v_i, its $start\_addr(v_i)$ is recursively defined as $\max\{start\_addr(v_j) + size(v_j) \mid v_j$ is an immediate predecessor of v_i in $\mathcal{O}\}$.

Clearly, the size of the SPM used by the above allocation scheme is equal to the longest path length of $\mathcal{O}$. Moreover, all the interference constraints are satisfied in the above allocation scheme. □

The SPM allocation problem, in essence, is to find an acyclic orientation of the interference graph such that its longest path length is minimized. In the special case where the sizes of all the variables are equal, the problem of finding an acyclic oriented graph with the minimum longest path length is equivalent to the problem of finding the minimum number of colors required to color the interference graph. Since the graph coloring problem is NP-complete for general graphs, the SPM allocation problem is also NP-complete. Therefore, we use a graph coloring heuristic to construct an acyclic oriented graph for SPM allocation.

Definition 1. *Given a legal vertex coloring ω of an interference graph $\mathcal{G} = (V, E)$, the ω-oriented graph of $\mathcal{G}$ is a directed graph $\mathcal{G}'(\mathcal{G}, \omega) = (V', E')$, where $V' = V$ and $E' = \{(v_i, v_j) \mid (v_i, v_j) \in E$ and $\omega(v_i) < \omega(v_j)\}$.*

In this definition, the ω-oriented graph $\mathcal{G}'$ of $\mathcal{G}$ is clearly an acyclic orientation of $\mathcal{G}$. For an acyclic oriented graph, we have defined an SPM allocation scheme in the proof of Theorem 1. For the ω-oriented graph $\mathcal{G}'$, this is made more explicit in Algorithm 1.

Algorithm 1: SPM-Coloring($V, \mathcal{G}'$)

Input: a set V of n variables and an ω-oriented graph $\mathcal{G}'$
Output: An SPM allocation scheme if true is returned

Find the longest path $P_{\max}$ of in $\mathcal{G}'$;
if $P_{\max}$'s length $> m$ **then**
  return false;
else
  foreach variable $v_i (i = 1, 2, \cdots, n)$ in topological order of all the nodes in $\mathcal{G}'$ **do**
    if v_i is a source node **then**
      $start\_addr(v_i) = 0$;
    else
      $start\_addr(v_i) = \max\{start\_addr(v_j) + size(v_j) \mid v_j$ is an immediate predecessor of v_i in $\mathcal{G}'\}$;
  return true;

4. Our WCET-Aware Approach

Our WCET-aware approach iteratively selects a most beneficial variable of a program as an SPM resident candidate and allocates it to the SPM. For variable selection, the most beneficial variable is the one with the most impact on the k longest paths in the program. In order to compute the impact of each variable on the k longest paths, our approach needs to construct a set of weighted DAGs as defined in Section 2 and compute the lengths of the k longest paths in the program. For SPM allocation, our approach uses an incremental graph coloring heuristic coupled with acyclic graph orientation. Finally, we will call SPM-Coloring given in Algorithm 1 to produce an allocation scheme for the program.

Section 4.1 gives an algorithm for constructing the weighted DAGs for a program. Section 4.2 gives an algorithm for computing the lengths of the k longest paths in a program. Section 4.3 is concerned with variable selection and SPM allocation. Section 4.4 shows that our approach has a polynomial-time complexity.

4.1 Constructing Weighted DAGs

A tree representation for a loop nest is introduced below.

```
while(a+b>c) /* L1 */
{
  if(x>y)
    for(i=0; i<=100; i++) /* L2 */
      {...;}
  else
    for(i=0; i<=200; i++) /* L3 */
      {...;}
  for(i=0; i<=100; i++) /* L4 */
  {
    if(a>b)
      for(i=0; i<=100; i++) /* L5 */
        {...;}
    else
      for(i=0; i<=200; i++) /* L6 */
        {...;}
  }
}
```

Listing 2: A loop nest.

Definition 2. *Given a loop nest LN, its loop nest tree is defined as $\mathcal{T}(LN) = (V, E)$, where $V = \{L_i \mid L_i$ is a loop in $LN\}$ and $E = \{(L_i, L_j) \mid L_i, L_j \in V$ and L_j is immediately nested within $L_i\}$.*

Consider a loop nest shown in Listing 2, with six loops numbered L_1 – L_6. Its loop nest tree is shown in Figure 2.

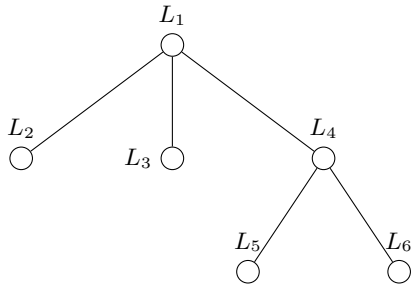

Figure 2: A loop nest tree.

According to Algorithm 2, we construct the weighted DAGs for a program by starting with a sink function and working towards the source functions in its call graph. For each loop nest contained in a function, we process its loops inside out, starting from the innermost ones and working towards the outermost loop. Applying this algorithm to Listing 1 yields Figure 1.

Algorithm 2: Weighted-DAGs-Constructor($\mathcal{C}$)

Input: The weighted CFG $\mathcal{C}$ of a program P
Output: A weighted DAG $\mathcal{D}(main)$ for P that is defined in terms of the weighted DAGs for its functions/loops

Construct the call graph, CallGraph(P), of P;
foreach $f_i (i = 1, 2, \cdots, m)$ in reverse topological order of all the function nodes in CallGraph(P) **do**
  foreach loop nest LN_j of f_i **do**
    Construct the loop nest tree $\mathcal{T}(LN_j)$;
    foreach node L_s of $\mathcal{T}(LN_j)$ in reverse topological order of all the loop nodes in $\mathcal{T}(LN_j)$ **do**
      Construct $\mathcal{D}(L_s)$, i.e., the weighted DAG of L_s from $\mathcal{C}$ with the back edge of L_s removed;
      Shrink the subgraph of L_s in $\mathcal{C}$ into one loop node;
  Construct $\mathcal{D}(f_i)$, i.e., the weighted DAG of f_i;
  Shrink the subgraph of f_i in $\mathcal{C}$ into one function node;
return $\mathcal{D}(main)$;

It is assumed that the main function of a program is named $main$. Therefore, $\mathcal{D}(main)$ represents the weighted DAG of the main function, i.e., the program itself, which is recursively defined in terms of the other weighted DAGs in the program.

4.2 Computing the Lengths of k Longest Paths

Computing the lengths of the k longest paths in a program is a key part of our heuristic for selecting a most beneficial variable as an SPM resident candidate. This is done according to Algorithm 3. For each node v_i of a weighted DAG $\mathcal{D}(x)$, we introduce a vector $v_i[1 : k]$ of length k to store the lengths of the k longest paths from the source node of $\mathcal{D}(main)$, i.e., the weighted DAG of the program to v_i in $\mathcal{D}(x)$ in non-increasing order. In the case when v_i is a loop node, $N(v_i)$ denotes its worst-case number of iterations. We approximate the lengths of its k longest paths by multiplying the lengths of the k longest paths of its one iteration with $N(v_i)$.

Algorithm 3: Find-k-Longest-Paths-Lengths($\mathcal{D}(x)$)

Input: $\mathcal{D}(x)$: The weighted DAG $\mathcal{D}(x)$ of a function/loop x
Output: The length vector containing the k longest paths of x

```
1  foreach node v_i(i = 1, 2, ··· , m) in topological order of all
   the nodes in D(x) do
2      if v_i is a basic block then
3          if v_i is the source node of D(x) then
4              A = {v_s | v_s is an immediate predecessor of v_i
               in the weighted DAG containing x as a node};
5          else
6              A = {v_s | v_s is an immediate predecessor of v_i
               in D(x)};
7          foreach v_s(s = 1, ··· , p) ∈ A do
8              for r = 1 to k do
9                  tmp[s][r] = v_s[r] + w(v_s, v_i);
10         Let x_1, x_2, ··· , x_k be the k largest, in
           non-increasing order, among all the p × k numbers
           stored in the 2D matrix tmp;
11         v_i[1 : k] = (x_1, x_2, ··· , x_k);
12     else
13         if v_i is a function node then
14             v_i[1 : k] =
               Find-k-Longest-Paths-Lengths(D(v_i));
15         else
               // v_i is a loop node
16             v_i[1 : k] = N(v_i)*Find-k-Longest-Paths-
               Lengths(D(v_i));
17 Let v_t be the sink node of D(x);
18 return v_t[1 : k];
```

A program can contain some infeasible paths, i.e., the paths that are never executed. This paper is not concerned with path feasibility analysis, which is a difficult problem in itself. Several approaches [7, 23] have been proposed to find infeasible paths of a program. We can use existing approaches to remove infeasible paths.

In order to compute the lengths of the k longest paths of a program, we invoke Find-k-Longest-Paths-Lengths($\mathcal{D}(main)$) given in Algorithm 3. When computing the lengths of the k longest paths, this algorithm traverses each node of a weighted DAG in topological order. That is, the lengths of the k longest paths to a node is computed only after the lengths of the k-longest paths to all its immediate predecessors have been computed. For a loop or function node, a recursive call is made to compute the lengths of the k-longest paths to each node in its weighted DAG.

4.3 Variable Selection and Allocation

Given a program and an SPM, let S be a set of variables such that if all the variables in S are allocated to the SPM, the WCET of the program under the SPM size constraint is minimized. We can partition S into a set of disjoint subsets S_1, S_2, $\cdots$, S_r such that for each S_i, the longest path of the program will no longer be the longest after allocating the variables in S_i to the SPM.

Therefore, an optimal set of SPM residents can be found by repeatedly selecting variables from the longest path of the program. However, there can be many variables on the longest path. Which variables should be selected? The heuristic introduced in [23] selects the variable with the most occurrences on the longest path. However, the best variable may not be selected for two main reasons. Firstly, an optimizing compiler can hide memory latency by scheduling other instructions to execute during a memory access. Secondly, when selecting a variable as an SPM resident candidate, we need to check not only its impact on the longest path but also its impact on other paths whose lengths are close to that of the longest path. Our variable selection heuristic is inspired by a desire to improve this existing heuristic by considering not only the longest path but also those potential ones in subsequent iterations.

Our approach consists of two procedures: selecting a variable with the largest benefit and allocating a variable to the SPM. For each variable v_i, we define its benefit vector $benefit(v_i)$ as:

$$benefit(v_i) = \frac{l - l(v_i)}{size(v_i)} \tag{1}$$

where l is a vector of the lengths, in non-increasing order, of the k longest paths of the program without allocating v_i to the SPM, $l(v_i)$ is a vector of the lengths, in non-increasing order, of the k longest paths of the program after allocating v_i to the SPM, and $size(v_i)$ is the size of v_i. Intuitively, the benefit vector of a variable v_i is the normalized contribution of v_i on the k longest paths of the program. Given two benefit vectors $benefit(x)$ and $benefit(y)$, their associated benefits are compared lexicographically. The larger one is preferred as it leads to a larger WCET reduction.

Our approach for selecting and allocating variables, given in Algorithm 4, iteratively selects a variable with the maximum non-negative benefit vector among all the variables on the current longest path and checks to see if it can be allocated to the SPM. If a variable can be allocated to the SPM, it is included in the set S of SPM residents and will not be considered again. If a variable has been selected as an SPM resident candidate but cannot be allocated to the SPM, it will be ignored now and also in subsequent iterations. This process is repeated until all the variables have been tested. Finally, SPM-Coloring given in Algorithm 1 is called to produce an SPM allocation scheme for all the SPM residents found.

Once a variable v_i is selected as a potential SPM resident, SPM-Allocator given in Algorithm 5 is called to check to see if it can be allocated to SPM. First of all, our SPM allocator tries to assign the 'best' color to v_i such that all interference constraints are satisfied. If such a color does not exist, our SPM allocator will assign a new color to v_i. After v_i has been assigned a color, our allocator adds v_i and all the interference edges between v_i and other SPM residents to the ω-oriented graph $\mathcal{G}'$ and computes the longest path $P_{\max}$ of the ω-oriented graph. If the length of $P_{\max}$ is greater than the SPM size m, v_i cannot be allocated to the SPM. In this case, all the changes to the ω-oriented graph $\mathcal{G}'$ are undone. Otherwise, v_i is selected as an SPM resident and the edge weights in the weighted DAGs of the program are updated.

Consider the interference graph shown in Figure 3, where the first element and the second element of each tuple are the benefit vector and the size of the variable, respectively. For simplicity, we assume that k is equal to 1, i.e., each benefit vector has only one element. Assume that the size of the SPM is 16 bytes. Our variable selector will pick variables d, x, e, y, a, z, b, f, and finally, c in sequence, and assign a color to each variable. The coloring result is shown in Figure 4, where the colors of d, x, e and y are 0, the colors of a, z, b and f are 1, and the color of c is 2. When assigning a color to a variable, our SPM allocator constructs the ω-oriented graph incrementally. The final ω-oriented graph is shown in Figure 5. The SPM allocation scheme constructed by our approach using Algorithm 1, based on the final ω-oriented graph, is shown in Figure 6. As we can see, our ω-SPM allocator finds an optimal allocation scheme for this interference graph using only 14 bytes of SPM space.

Algorithm 4: Variable-Selection-Allocation(P, m)

Input: A whole program P and an SPM of size m
Output: A set of variables allocated to the SPM

Construct the weighted CFG $\mathcal{C}$ of P;
$\mathcal{D}(main)$ = Weighted-DAGs-Constructor($\mathcal{C}$);
Construct the interference graph $\mathcal{G}$ of P;
V = all the variables in P;
S = set of variables already colored, initially $\emptyset$;
Create an empty ω-oriented graph $\mathcal{G}'(\mathcal{G}, \omega)$ (Definition 1), where ω is a legal vertex coloring built in Algorithm 5;
$max\_color = 0$;
while $V \neq \emptyset$ **do**
    $R = V \cap \{v \mid v$ is a variable on the longest path$\}$;
    if $R = \emptyset$ **then**
        `// The longest path cannot be reduced`
        break ;
    l = Find-k-Longest-Paths-Lengths($\mathcal{D}(main)$);
    foreach $v_i \in R$ **do**
        Assume that v_i can be allocated to the SPM;
        Recalculate the edge weights in the weighted DAGs;
        $l(v_i)$ = Find-k-Longest-Paths-Lengths($\mathcal{D}(main)$);
        $benefit(v_i) = (l - l(v_i))/size(v_i)$;
        Undo the changes to the weighted DAGs;
    $alloc$ = false ;
    while $alloc$ = false $\wedge$ $R \neq \emptyset$ **do**
        Select a variable $v_j \in R$ with the maximum non-negative benefit vector, i.e., $benefit(v_j) > 0$;
        $V = V - \{v_j\}$;
        $R = R - \{v_j\}$;
        if SPM-Allocator(v_j) = true **then**
            `//` v_j `has been allocated to the SPM`
            $S = S \cup \{v_j\}$;
            $alloc$ = true;
`// Allocate an address to each SPM resident`
SPM-Coloring($S, \mathcal{G}'$);

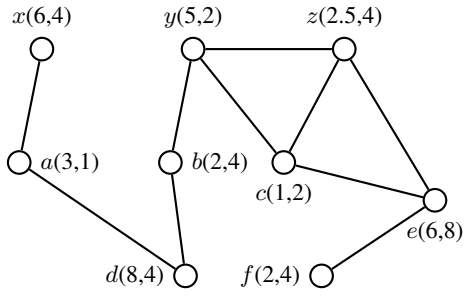

Figure 3: An interference graph.

4.4 Time Complexity Analysis

In this section, we analyze the time complexity of our WCET-aware approach. Modern compilers hide memory latency by scheduling some ready instructions to execute during a memory access. If a variable is allocated to the SPM, its latency will change to one processor cycle. For VLIW (very long instruction word) processors, the compiler needs to compute a schedule for each basic block and each innermost loop. If a variable is allocated to the SPM, the compiler needs to recompute a schedule for each of the basic blocks containing this variable and all the other basic affected by this variable. The problem of scheduling a basic block on a RISC processor such that its execution time is minimized is NP-complete [18]. The compiler typically uses a heuristic to compute a schedule for a basic block. Since instruction scheduling is dependent on the target processor model, we will ignore the time complexity of computing a schedule and the execution time for each basic block in the following time complexity analysis. The time complexity of our approach is dominated by the following parts:

1. Constructing the weighted CFG. Since the compiler has already generated the CFG, this part takes at most $O(e)$ as we ignore

Algorithm 5: SPM-Allocator(v_i)

Input: A variable v_i to be allocated to the SPM
Output: A color assigned to v_i if v_i is allocated to SPM and a boolean value indicating success or failure

`// All variables used but not explicitly defined here are defined in Algorithm 4`
if $S = \emptyset$ **then**
    `// First variable to go into the SPM`
    if $size(v_i) \leq m$ **then**
        `// Assign color 0 to` v_i
        $\omega(v_i) = 0$;
    else
        return false;
else
    $C = \{0, 1, \cdots, max\_color\}$;
    `//` C `is the set of the colors used`
    $B = C - \{r \mid r$ is a color assigned to an adjacent variable $v_j (v_j \in S)$ of v_i in the interference graph $\mathcal{G}\}$;
    if $B = \emptyset$ **then**
        `// Select a new color that is different from any color in` C
        $max\_color = max\_color + 1$;
        $\omega(v_i) = max\_color$;
    else
        Find a variable $v_j \in S$ with the maximum size among all the variables with a color in B;
        $\omega(v_i) = \omega(v_j)$;
Add a new node v_i to $\mathcal{G}'$;
foreach edge $(v_j, v_i)(v_j \in S)$ in the interference graph $\mathcal{G}$ **do**
    if $\omega(v_i) < \omega(v_j)$ **then**
        Add the directed edge (v_i, v_j) to $\mathcal{G}'$;
    else
        Add the directed edge (v_j, v_i) to $\mathcal{G}'$;
Find the longest path $P_{\max}$ of $\mathcal{G}'$;
if $P_{\max}$'s length $> m$ **then**
    `//` v_i `cannot be allocated to the SPM`
    Delete v_i and all edges incident on v_i from $\mathcal{G}'$;
    return false ;
else
    Recalculate the edge weights in the weighted DAGs;
    return true;

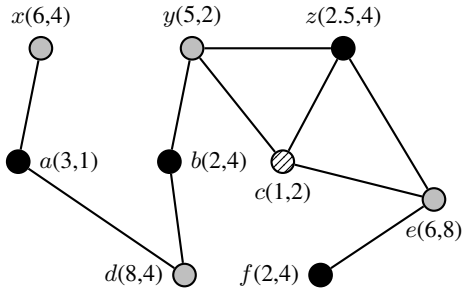

Figure 4: Coloring results.

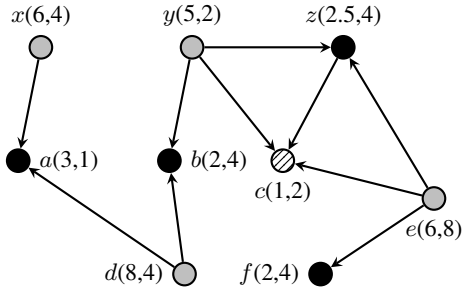

Figure 5: ω-oriented graph.

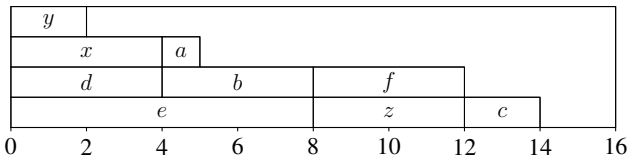

Figure 6: ω-oriented graph-based SPM allocation result.

the time complexity for computing schedules for basic blocks, where e is the number of edges in the CFG.

2. Constructing the interference graph. We can reuse the interference graph constructed by the compiler. Therefore, the time consumed by this part is not included in our approach.
3. Computing all the weighted DAGs. All the weighted DAGs are constructed from the weighted CFG. So this part takes $O(e)$.
4. Selecting a most beneficial variable. This part takes $O(e * n)$ time, where n is the number of variables.
5. Allocating a variable to the SPM. This part takes $O(e')$ time, where e' is the number of edges in the ω-oriented graph. Since there are at most n SPM residents, e' is at most $O(n^2)$. As a result, this part takes at most $O(n^2)$ time.

Since at most n variables can be selected and allocated to the SPM, the total time for selecting and allocating variables is $O(e*n^2)$ if we ignore the time complexity for computing schedules for basic blocks. As a result, the time complexity of our approach is $O(e * n^2)$. Even if we consider the time complexity for computing the schedules of basic blocks, it is easy to see that our approach is still polynomial assuming that the compiler uses a polynomial time algorithm to compute a schedule for each basic block.

5. Evaluation

In this section, we present and analyze the experimental results of our WCET-aware variable selection and allocation approach.

5.1 Experiment Setup

We have implemented our approach in order to evaluate its performance. The target architecture is an out-of-order, pipelined processor, with an instruction cache and perfect branch prediction. The hit and miss latencies for the instruction cache are 1 and 10 cycles, respectively. The target processor uses SPM to replace data cache. The latencies of SPM and off-chip memory accesses are 1 cycle and 10 cycles, respectively. We have used the 10-cycle off-chip memory latency to facilitate comparing with the two prior approaches [2, 22], where the latency used is 10 cycles [22] and 12 cycles [2].

We have selected eight benchmarks from two benchmark suites: Powerstone [21] and Mälardalen WCET Benchmarks [8]. Some statistics for the benchmarks are given in Table 1. As shown, both scalars and arrays are considered in SPM allocation. For all the benchmarks, we apply the live range splitting techniques proposed in [13] first so that better SPM utilization can be achieved.

The benchmarks are compiled using gcc 2.7.2.3 targeted for SimpleScalar. We have modified Chronos [16] 4.0 to analyze a binary program, calculate the execution times of its basic blocks, and construct its whole-program CFG. For comparison reasons, we have also implemented the longest-path-based greedy approach proposed in [22] and the ILP-based approach proposed in [2]. The former approach is static and thus assumes essentially that all the variables have the same live ranges. The latter approach is dynamic with live-range splitting being considered in its ILP formulation.

A program may have some infeasible paths. By default, the path feasibility analysis that comes with Chronos 4.0 is turned on so that some infeasible paths are eliminated for all approaches evaluated.

Our experiments are performed under three different SPM configurations with three different SPM sizes. The three SPM sizes we have selected are 5%, 10% and 20% of the total data size of each benchmark. For each SPM size, we evaluate the performance of our approach and two prior approaches. For our k longest paths approach, we have selected different values for k, ranging from 1 to 6. Our experimental results show that the performance of our approach improves significantly when k increases from 1 to 3. However, the improvements become less pronounced when k increases further. The reason is that in our selected benchmarks, the lengths of the 4th to 6th longest paths are almost the same as that of the 3rd longest path. They are redundant according to the benefit vector metric used. Therefore, in our benchmarks, the best performance results are achieved when $k = 3$, making our approach rather efficient.

5.2 Results

The experimental results are plotted in Figure 7, where k is set to 3. The horizontal axis in each figure is the percentage of SPM size over the total data size, and the vertical axis represents the percentage of WCET reduction in terms of execution cycles over the one with no data allocated to SPM. There are three approaches evaluated. For every SPM size, there are three bars, each representing the percentage of WCET reduction of the corresponding approach. To show the efficiency of our approach, we have also recorded the execution times (in seconds) of the three approaches under 10% SPM configuration, which are included in Table 1.

5.3 Analysis

Examining our results, we make the following observations:

1. In general, as shown in Figure 7, our approach outperforms the other two approaches across all the eight benchmarks un-

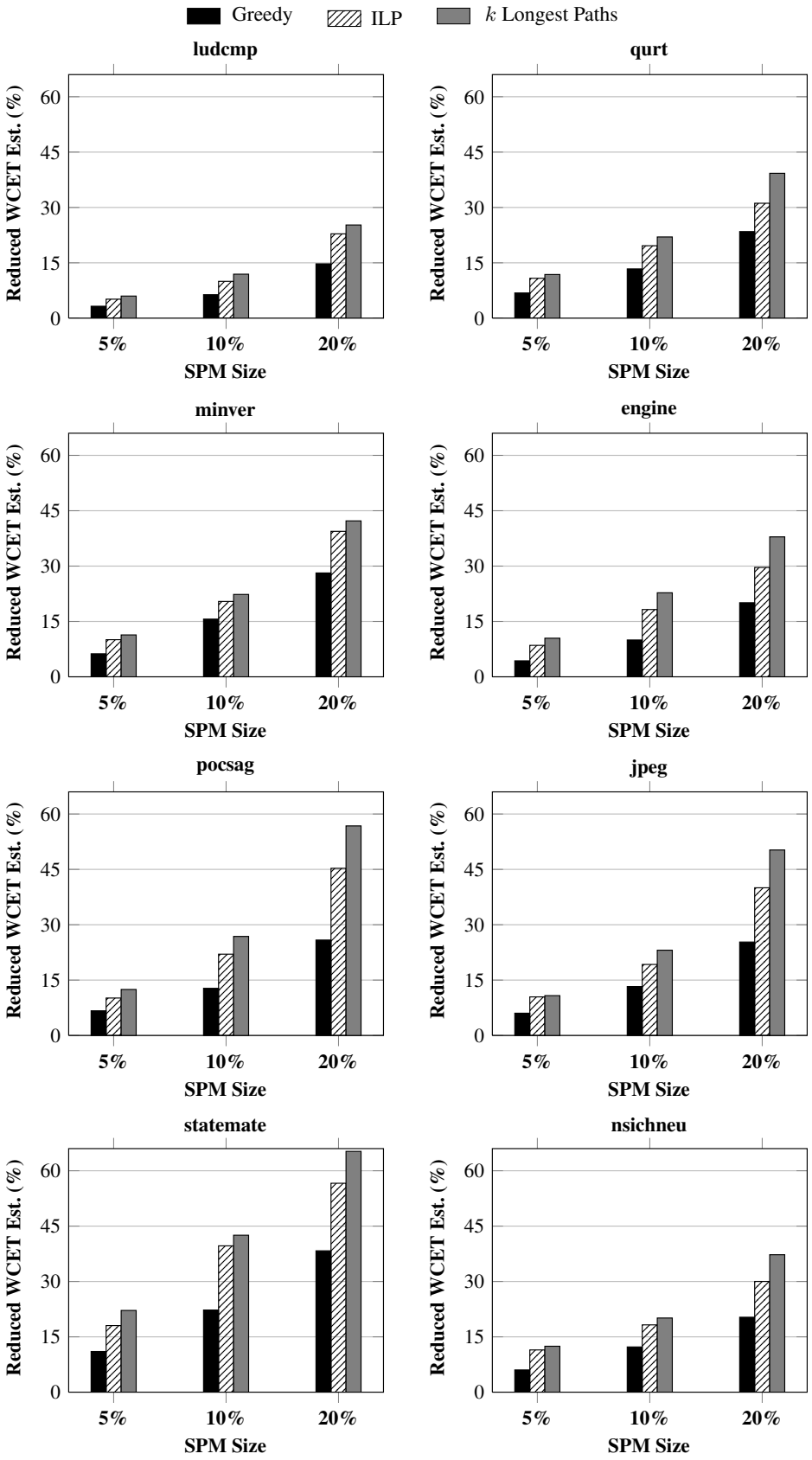

Figure 7: Comparing three approaches in WCET reduction for benchmarks under three SPM configurations.

Benchmark	Lines of Code	No. of Basic Blocks	Data size (Bytes)	Arrays (Bytes)	Scalars (Bytes)	Running Time (secs)			Description
						Greedy	ILP	k Longest Paths	
ludcmp	147	45	21680	21600	80	1.2	8.5	1.6	LU decomposition
qurt	166	91	148	56	92	3.7	20.8	5.3	Quadratic root computation
minver	201	82	2604	2432	172	3.1	18.4	4.2	Matrix inversion
engine	289	56	478	370	108	2.1	35.7	2.8	Engine control
pocsag	521	203	1216	1038	178	7.2	52.6	9.6	Communication protocol
jpeg	529	109	77561	77273	288	4.3	41.4	6.3	JPEG decode
statemate	1198	342	227	163	64	11.9	125.6	13.8	Car window lift control
nsichneu	4253	751	1588	56	1532	17.5	246.3	19.7	Simulation for an extended Petri Net

Table 1: Benchmark statistics and the results under the 10% SPM configuration.

der all the three SPM configurations. In WCET reduction, our approach enhances performance by up to 21% (for `pocsag` under 20% SPM) over the ILP-based dynamic allocation approach, and up to 43% (for `statemate` under 20% SPM) over the longest-path-based greedy approach. On average, under the 20% SPM configuration, our approach outperforms the ILP-based approach and greedy approach by 12% and 26%, respectively.

2. The performance of our approach increases as the SPM size increases. For all benchmarks, our k longest-paths-based approach achieves only small improvements over the other two approaches under the 5% SPM configuration. The reason is that the SPM size is too small, and only the most beneficial data variables on the longest path can be allocated to the SPM. When the SPM size increases to 10%, the improvements from our approach become more noticeable. Under the 20% SPM configuration, our approach achieves the best improvements.
3. Under the 20% SPM configuration, for `statemate`, `jpeg`, `pocsag`, `nsichneu`, `engine` and `qurt`, our approach shows significant improvements over the other two approaches. These benchmarks have multiple long execution paths with balanced branches. The lengths of 3 longest paths are close, thereby favoring our k longest-paths-based approach.
4. For `ludcmp` and `minver`, our approach works slightly better than the ILP-based approach under every SPM configuration. The reason is that data variables in these two benchmarks are dominated by a few very large arrays. Even if we increase the SPM size to 20%, they still cannot fit in the SPM. So we can only select other variables as SPM residents.
5. Our approach is much faster than the ILP-based approach. Table 1 compares the execution times of the three approaches for all the benchmarks under the 10% SPM configuration. As we can see, our approach runs slightly more slowly than the greedy approach but a lot more efficiently than the ILP-based approach.

From the experimental results we can conclude that our approach is more effective for programs with more balanced long paths, particularly when mapped to relatively larger SPMs. Even if the available SPM size is relatively small or the program is dominated by one single long path, our algorithm does not perform worse than the other two approaches.

There are two reasons why our approach performs better than the ILP-based approach proposed in [2]. The first reason is that our variable selection heuristic considers not only the longest path, but also the other paths whose lengths are close to that of the longest path. One of these other paths may become the longest path as the analysis proceeds. In contrast, the ILP-based approach only considers the longest path. The second reason is that our SPM allocation heuristic relies on both graph coloring and acyclic graph orientation, while the ILP-based approach uses a simple first-fit-based SPM allocation technique.

6. Conclusion

We have studied the problem of selecting variables of a program as SPM residents to minimize the WCET of the program. We have proposed an efficient, k-longest-paths-based dynamic approach. Our approach consists of two novel heuristics, a heuristic for iteratively selecting a most beneficial variable as an SPM resident candidate and a heuristic for incrementally allocating the most beneficial variable to the SPM. The benefit of each variable is its contribution to the path length reduction of the k longest paths of the program. Our SPM allocation heuristic is based on an efficient graph coloring technique coupled with acyclic graph orientation. We have also evaluated our approach by comparing with an ILP-based approach and a longest-path-based greedy approach using the eight benchmarks selected from Powerstone and Mälardalen WCET Benchmark suites under three different SPM configurations. Our approach achieves up to 21% and 43% improvements in WCET reduction over the ILP-based approach and the greedy approach, respectively.

Our approach can be extended in three directions. Firstly, WCET-aware live range splitting can be employed to further reduce the WCET of a program. Different live range splitting techniques have been proposed in the previous approaches to the register and SPM allocation problems. These techniques aim at minimizing the average execution time of a program. The challenge in WCET-aware live range splitting is to find an optimal set of live ranges to split in order to minimize the WCET of a program. Secondly, allocating heap data into the SPM is another challenging problem. Accesses to heap data are usually performed via pointers. The value of a pointer dynamically changes, making it difficult for the compiler to manage heap accesses in the SPM. Finally, a typical real-time embedded system consists of a set of tasks with timing constraints. All the tasks executed on a processor need to share the SPM of the processor. The challenging problem is to optimally partition the SPM among a set of tasks to find a feasible schedule whenever such a feasible schedule exists.

7. Acknowledgement

Thanks to the reviewers for their comments on the work. This research is supported by the Australian Research Council (ARC) grants, DP0881330 and DP110104628.

References

[1] ANGIOLINI, F., MENICHELLI, F., FERRERO, A., BENINI, L., AND OLIVIERI, M. A post-compiler approach to scratchpad mapping

of code. In *Proceedings of the 2004 International Cconference on Compilers, Architecture and Synthesis for Embedded Systems* (2004), pp. 259–267.

[2] DEVERGE, J.-F., AND PUAUT, I. WCET-directed dynamic scratchpad memory allocation of data. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems* (2007), pp. 179–190.

[3] EGGER, B., LEE, J., AND SHIN, H. Dynamic scratchpad memory management for code in portable systems with an MMU. *ACM Transactions on Embedded Computing Systems 7*, 2 (2008), 11:1–11:38.

[4] FALK, H., AND KLEINSORGE, J. C. Optimal static WCET-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference* (2009), pp. 732–737.

[5] FALK, H., PLAZAR, S., AND THEILING, H. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis* (2007), pp. 143–148.

[6] FRANCESCO, P., MARCHAL, P., ATIENZA, D., BENINI, L., CATTHOOR, F., AND MENDIAS, J. M. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st annual Design Automation Conference* (2004), pp. 238–243.

[7] GOLDBERG, A., WANG, T., AND ZIMMERMANN, D. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis* (1994), pp. 80–94.

[8] GUSTAFSSON, J., BETTS, A., ERMEDAHL, A., AND LISPER, B. The Mälardalen WCET benchmarks – past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis* (2010), pp. 137–147.

[9] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2007.

[10] JANAPSATYA, A., IGNJATOVIC, A., AND PARAMESWARAN, S. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference* (2006), pp. 612–617.

[11] JANAPSATYA, A., PARAMESWARAN, S., AND IGNJATOVIC, A. Hardware/software managed scratchpad memory for embedded system. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design* (2004), pp. 370–377.

[12] KANDEMIR, M. T., RAMANUJAM, J., IRWIN, M. J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th annual Design Automation Conference* (2001), pp. 690–695.

[13] LI, L., FENG, H., AND XUE, J. Compiler-directed scratchpad memory management via graph coloring. *ACM Transactions on Architecture and Code Optimization 6*, 3 (2009), 9:1–9:17.

[14] LI, L., NGUYEN, Q. H., AND XUE, J. Scratchpad allocation for data aggregates in superperfect graphs. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (2007), pp. 207–26.

[15] LI, L., XUE, J., AND KNOOP, J. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Transactions on Embedded Computing Systems 10*, 2 (2011), 28:1–28:42.

[16] LI, X., LIANG, Y., MITRA, T., AND ROYCHOUDHURY, A. Chronos: A timing analyzer for embedded software. *Science of Computer Programming 69*, 1-3 (2007), 56–67.

[17] LIU, T., LI, M., AND XUE, C. J. Minimizing WCET for real-time embedded systems via static instruction cache locking. In *Proceedings of the 15th IEEE Symposium on Real-Time and Embedded Technology and Applications* (2009), pp. 35–44.

[18] PALEM, K. V., AND SIMONS, B. B. Scheduling time-critical instructions on risc machines. *ACM Transactions on Programming Languages and Systems 15*, 4 (1993), 632–658.

[19] PLAZAR, S., FALK, H., KLEINSORGE, J. C., AND MARWEDEL, P. WCET-aware static locking of instruction caches. In *Proceedings of the International Symposium on Code Generation and Optimization* (2012), pp. 44–52.

[20] PUAUT, I., AND PAIS, C. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, Automation and Test in Europe* (2007), pp. 1484–1489.

[21] SCOTT, J., LEE, L. H., ARENDS, J., AND MOYER, B. Designing the low-power M*CORE architecture. In *Proceedings of IEEE Power Driven Micro Architecture Workshop* (1998), pp. 145–150.

[22] SUHENDRA, V., MITRA, T., ROYCHOUDHURY, A., AND CHEN, T. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium* (2005), pp. 223–232.

[23] SUHENDRA, V., MITRA, T., ROYCHOUDHURY, A., AND CHEN, T. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd annual Design Automation Conference* (2006), pp. 358–363.

[24] UDAYAKUMARAN, S., AND BARUA, R. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (2003), pp. 276–286.

[25] VERA, X., LISPER, B., AND XUE, J. Data cache locking for higher program predictability. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2003), pp. 272–282.

[26] VERA, X., LISPER, B., AND XUE, J. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systems 7*, 1 (2007), 4:1–4:38.

[27] VERMA, M., AND MARWEDEL, P. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Transactions on Very Large Scale Integration Systems 14*, 8 (2006), 802–815.

[28] WU, H., XUE, J., AND PARAMESWARAN, S. Optimal WCET-aware code selection for scratchpad memory. In *Proceedings of the 10th ACM International Conference on Embedded Software* (2010), pp. 59–68.

[29] YANG, X., WANG, L., XUE, J., DENG, Y., AND ZHANG, Y. Comparability graph coloring for optimizing utilization of stream register files in stream processors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2009), pp. 111–120.