Optimal WCET-Aware Code Selection for Scratchpad Memory

Hui Wu, Jingling Xue, Sri Parameswaran
School of Computer Science and Engineering
The University of New South Wales
{huiw,jingling,sridevan}@cse.unsw.edu.au

ABSTRACT

We propose the first polynomial-time code selection algorithm for minimising the worst-case execution time of a non-nested loop executed on a fully pipelined processor that uses scratchpad memory to replace the instruction cache. The time complexity of our algorithm is $O(m(ne + n^2 \log n))$, where n and e are the number of basic blocks and the number of edges in the control flow graph of the loop, and m is the size of the scratchpad memory. Furthermore, we propose the first dynamic code selection heuristic for minimising the worst-case execution time of a task by using our algorithm for a non-nested loop. Our simulation results show that our heuristic significantly outperforms a previously known heuristic.

Categories and Subject Descriptors

D.3.4 [**Processors**]: [Compilers, Memory Management, Optimisation]; D4.7 [**Organisation and Design**]: Real-time systems and embedded systems

General Terms

Algorithms, Design, Performance

Keywords

Scratchpad Management, Worst-case Execution Time, Minimum Node Cut

1. INTRODUCTION

During past decades, the speed disparity between processor and off-chip memory has been increasing. To bridge the growing speed disparity, modern processors use caches to speed up accesses to off-chip memory. Nevertheless, caches cause two major problems. Firstly, they consumes a significant amount of processor power. Secondly, they introduce additional complexity for computing the worst-case execution time (WCET) of a task. In embedded systems SPM (scratchpad memory) can be used to overcome these two problems. SPM is on-chip static random access memory (SRAM). It does not contain a tag store and associated circuitry as in caches. Therefore, SPM consumes much less energy than caches. Moreover, SPM makes it much easier to compute the worst-case execution time of a task as all the accesses to SPM are known at compiler time. For these two reasons, SPM has been used in many embedded processors and most DSPs. Examples are NVIDIA's PhysX PPU(physics processing unit)[2] and the Cell multiprocessor jointly developed by Sony, IBM, and Toshiba[11].

In order to use SPM, a compiler must explicitly insert instructions to transfer selected data and code between offchip memory and SPM. The data and code which are transfered from offchip memory to SPM are called scratchpad residents. There are two major issues in SPM management. The first issue is which subset of data and code should be selected as scratchpad residents. The second issue is where the selected data and code should be stored in SPM. To solve these two issues, researchers have done extensive research and proposed many SPM management approaches. All the existing approaches can be classified into two categories: static allocation[18, 7, 4, 8, 10, 9] and dynamic allocation[15, 5, 12, 14, 13, 19, 6, 20]. In static allocation approaches once a scratchpad resident is loaded into SPM, its space in SPM cannot be allocated to other scratchpad residents during the execution of its task. As a result, static allocation approaches lead to low SPM utilisation. Dynamic allocation approaches consider the SPM allocation problem as a generalised register allocation problem and transfer scratchpad residents from off-chip memory to SPM dynamically. When allocating scratchpad residents to SPM dynamic allocation approaches consider their live ranges. If the live ranges of two SPM residents do not overlap, these two SPM residents can be allocated to the same area of the SPM. Therefore, dynamic allocation approaches result in more efficient SPM utilisation.

Most of the existing approaches for scratchpad management aim to minimise either the average execution time or the average energy consumption of a task. These approaches are not suitable for real-time embedded systems. The primary objective of real-time embedded systems design is to meet all the timing constraints. The worst-case execution times of all the tasks of a real-time embedded system are the key factors that affect the satisfiability of timing constraints. Therefore, the objective of SPM management for real-time systems should be minimising the worst-case execution time of each task. Nevertheless, only a few approaches have been

EMSOFT'10, October 24–29, 2010, Scottsdale, Arizona, USA.

proposed to achieve this objective. For the first time Suhendra et al.[18] studied the problem of selecting data as SPM residents so that the worst-case execution time of a task is minimised. They proposed several approaches for selecting data as SPM residents. However, their approaches use the static allocation technique and therefore may result in low SPM utilisation. Deverge and Puaut proposed a heuristic for selecting data as scratchpad residents that aims to minimise the worst-case execution time of a task. Unlike the approaches in[18], their algorithm uses a dynamic allocation approach. Puaut and Pais[15] proposed an approach to selecting basic blocks of code such that worst-case execution time of a task is reduced. However, their approach may perform very poorly in the worst-case.

In this paper, we consider a target processor that uses SPM to replace an instruction cache. We study the problem of selecting code of a task as scratchpad residents such that the worst-case execution time of the task is minimised. We propose the first polynomial-time code selection algorithm that minimises the worst-case execution time of a non-nested loop that runs on a fully pipelined processor. Our algorithm introduces a novel optimal basic block splitting technique and converts the problem of minimising the worst-case execution time of a non-nested loop into the problem of finding minimum node cuts of a set of graphs. Furthermore, we propose a dynamic allocation heuristic for minimising the worst-case execution time of a task by using our optimal algorithm for a non-nested loop.

This paper is organised as follows. Section 2 describes the system model and introduces several key definitions. In Section 3, firstly, we propose a polynomial-time code selection algorithm for a non-nested loop with equal basic block sizes, analyse its time complexity and prove its optimality; secondly, we propose a polynomial-time code selection algorithm for non-nested loops with arbitrary basic block sizes. Section 4 proposes a code selection heuristic for loop nests. Section 5 discusses the related work and presents our simulation results.

2. SYSTEM MODEL AND DEFINITIONS

We consider the real-time embedded systems where a processor uses SPM to replace the instruction cache to speed up instruction fetches. The target processor uses a fully pipelined architecture where all instructions are pipelined. The execution of each instruction takes one processor cycle if the instruction is already in the SPM. Otherwise, it takes p cycles due to the off-chip memory latency. One example of such processors is MIPS 2000 processor. The target processor provides a special instruction *fetch* to load a sequence of contiguous instructions from off-chip memory into the SPM. Throughout this paper, the ratio α of the off-chip memory speed and the processor speed is $1/p$.

A basic block is a sequence of code that has only one entry and only one exit. Obviously, if a basic block is not within a loop, it is not beneficial to load it from the off-chip memory into the SPM unless the target processor provides a DMA (Direct Memory Access) mechanism to prefetch basic blocks. In this paper, we only consider the basic blocks within a loop as the candidates of SPM residents. Basic blocks may be split into two smaller blocks in order to minimise the worst-case execution time of a task.

CFG (Control Flow Graph) is a classical data structure for representing a program. In a CFG, each node is a basic block and each edge represents the control flow from one block to another. Typically, the CFG of a loop contains a *preheader node*, a *header node*, and two types of edges: *forward edges* and *back edges*. The preheader node is the source of loop entry edge. The header node is a node that dominates all other nodes in the CFG. A back edge is an edge whose target is the entry node.

We assume that given a loop each path of its CFG is feasible and the worst-execution time of the loop is equal to the execution time of its longest path multiplied by the maximum number of iterations of the loop. Since SPM is much faster than off-chip memory, the execution time of a basic block is dominated by the time of fetching the basic block from the off-chip memory. Under these assumptions, the problem of minimising the worst-case execution time of a non-nested loop reduces to the problem of minimising the worst-case execution time of a single iteration of the non-nested loop.

To model a single iteration of a non-nested loop, we do not consider the back edges of the CFG of the loop. Therefore, a single iteration of a loop can be represented by a weighted DAG (Directed Acyclic Graph) $G = \langle V, E, W\rangle$, where V is the set of all non-preheader nodes of the CFG of the loop, E is the set of all forward edges of the CFG, and W is the set of weights of all nodes. The weight of each node v_i, denoted by w_i, is the size of the corresponding basic block. For ease of description, we assume that the execution time of a basic block v_i is w_i if v_i is not a SPM resident; otherwise it is αw_i.

In a weighted DAG G, if there is a path from v_i to v_j. v_i is a *predecessor* of v_j and v_j is a *successor* of v_i. If (v_i, v_j) is an edge of G, v_i is an *immediate predecessor*, or a *parent* of v_j and v_j is an *immediate successor*, or a *child* of v_i. A node is a *source* node if it has no parents. A node is a *sink* node if it has no children. The path length of a path is the sum of the weights of all constituent nodes. A path is an *induced path* of a node v_i if the path includes v_i. The length of the longest path of a weighted DAG G is denoted by $l_{\max}(G)$. A weighted DAG G' is a subgraph of G if the vertex set, edge set and node weight set of G' are subsets of those of G. Given a set S, its size is denoted by $|S|$.

Definition 2.1. *Given a weighted DAG $G = \langle V, E, W\rangle$, a node cut of G is a subset $V' \subseteq V$ such that each path from a source node to a sink node must contain a node in V'. A node cut is a minimum node cut if it has the minimum number of nodes among all node cuts.*

Definition 2.2. *Given a weighted DAG $G = \langle V, E, W\rangle$, a subset S of V and a real number α, the DAG $G(S, \alpha) = \langle V', E', W'\rangle$ is defined as follows: $V' = V$, $E' = E$ and $W' = \{w'_i$: if $v_i \in S$, $w'_i = \alpha * w_i$; otherwise, $w'_i = w_i$, where w_i is the weight of v_i in $G\}$.*

Intuitively, $G(S, \alpha)$ is the resulting graph of G after all the basic blocks represented by S are selected as scratchpad residents.

Definition 2.3. *Given a weighted DAG $G = \langle V, E, W\rangle$ and a subset S of V, the weighted DAG $G(S) = \langle V', E', W'\rangle$ is defined as follows: $V' = V - S$, $E' = \{(v_i, v_j) : (v_i, v_j) \in E$ or there is a path from v_i to v_j in G such that all the nodes, excluding v_i and v_j, of the path are in $S\}$ and $W' = \{w'_i$: $v_i \in V'$ and w'_i is equal to the weight w_i of v_i in $G\}$.*

The weighted DAG $G(S)$ is used to denote the state of a loop after all its basic blocks represented by S are selected as scratchpad residents. So $G(S)$ does not contain all the nodes in S and the connectivity of each pair of nodes in $G(S)$ remains the same as in G.

DEFINITION 2.4. *Given a weighted DAG $G = \langle V, E, W\rangle$ and a real number x, the x-spanning graph of G is a subgraph $G(x)$ of G such that for each source node v_i and each sink node v_j in $G(x)$ the length of each path from v_i to v_j in $G(x)$ is greater than x.*

3. CODE SELECTION FOR NON-NESTED LOOPS

For a non-nested loop, we load the selected basic blocks of the loop in the preheader block. During the execution of the loop, all the basic blocks in the SPM can be fetched much faster than the basic blocks stored in the off-chip memory. The objective of the basic block selection is to minimise the worst case execution time of the loop while satisfying the SPM capacity constraint.

Let $G = \langle V, E, W\rangle$ be a weighted DAG for a non-nested loop, where $V = \{v_1, v_2, \cdots, v_n\}$ is a set of basic block, $E = \{(v_i, v_j) : v_i$ is directly control-dependent on $v_j\}$, and $W = \{w_i$: w_i is the size of the basic block $v_i\}$. Assume that the scratchpad size is m. Our objective is to find a subset $S \subseteq V$ such that the following constraints are satisfied:

1. $\sum_{v_i \in S} w_i \leq m$.
2. $l_{\max}(G(S, \alpha)) = min\{l_{\max}(G(S', \alpha)) : S' \subseteq V$ and $\sum_{v_i \in S'} w_i \leq m\}$.

The first constraint implies that the total size of all scratchpad residents is at most m, the size of the scratchpad. The second constraint states that the subset $S \subseteq V$ minimises the longest path of the non-nested loop.

3.1 Equal Weights

We first consider a special case of the basic block selection problem for a non-nested loop where all weights are equal. This problem can be solved optimally in polynomial time. Assume that all weights are equal to k.

Our algorithm has four inputs: a weighted DAG G that represents a loop, the number of iterations of the loop, a set A of nodes (basic blocks) that have been already selected as the SPM residents, and the scratchpad size m. For a non-nested loop, A is an empty set. A is not empty if the loop contains another loop. We will discuss nested loops in next section. Note that the weight of a selected node can be an arbitrary integer. Our algorithm returns an optimal set of basic blocks selected as the SPM residents and the size of the free space of the SPM that is not occupied by the optimal set of basic blocks.

Our algorithm uses a greedy strategy. At each stage, it finds the minimum set of basic blocks such that the length of the longest path of the DAG is reduced by $k(1-\alpha)$ after loading all the basic blocks in the minimum set into the SPM. The minimum set of basic blocks is found by computing the minimum node cut of the $(l_{\max}(G) - k(1-\alpha))$-spanning graph of the DAG G of the loop. All the basic blocks of a minimum node cut are loaded in the SPM only if the number of nodes of the minimum node cut is less than the number of iterations of the loop. If all the basic blocks of a minimum set cannot be loaded into the SPM, our algorithm ranks all the basic blocks in the minimum set according to their impacts on the longest path and selects the basic block with the highest impacts on the longest path of G based on the SPM capacity constraint. Our algorithm is shown in pseudo code as follows.

Algorithm *OptimalCodeSelection1*(G, A, m, r)
input: *A set A of basic blocks that have been selected as the SPM residents, a weighted DAG G where the weight of each node that is not in A is equal to k, the number of iterations r of the loop, and the scratchpad size m.*
output: *A subset S of nodes of G as SPM residents for minimising the longest path length and the size of the free space of the SPM.*

```
begin
  size = ∑_{v_i ∈ A} w_i;
  if size = m
    return (∅, 0);    /* No free SPM space */
  S = A;
  compute the longest path length l_max(G) of G;
  while size < m do
    construct G(l_max(G) − k(1 − α));
    let G' be G(l_max(G) − k(1 − α));
    construct G'(S);
    find a minimum node cut C of G'(S);
    if |C| >= r /* Do not load C into the SPM */
      break; /* Exit from the loop */
    if size + |C| ∗ k ≤ m
      S = S ∪ C;
      size = size + |C| ∗ k;
      for each v_i ∈ C do
        change the weight w_i of v_i to k ∗ α;
        /* the longest path of G is reduced by k(1 − α) */
        l_max(G) = l_max(G) − k(1 − α);
    else
      for each node v_i ∈ C do
        compute the maximum length of all the paths
        that include v_i;
      sort all nodes in C in non-increasing order of their
      maximum lengths;
      let B be the set of the first ⌊(m − size)/k⌋ nodes
      in the sorted set C;
      S = S ∪ B;
      for each node in B do
        change the weight w_i of v_i to k ∗ α;
  inserts the fetch instructions in the preheader
  of the loop to load all the basic blocks in S;
  return (S, m − size);
end
```

Example 1 Consider a non-nested loop that is represented by the DAG shown in Figure 1, where $\alpha = 0.2$, $k = 100$. Assume that the scratchpad size m is 400. The longest path length of the DAG is 600. In the first iteration, our algorithm constructs the $G(520)$-spanning graph in which the path length of each path is greater than 520 as shown in Figure 2. One minimum cut of the $G(520)$-spanning graph is $\{v_8\}$. So v_8 is selected as a SPM resident and its weight is changed to $100\alpha = 20$. In the second iteration, our algorithms constructs the $G(440)$-spanning graph in which the path length of each path is greater than 440 as shown in Figure 3. Our algorithm removes the selected node v_8 from the $G(440)$-spanning graph and finds the minimum node cut $\{v_1\}$ as shown in Figure 4. In the last iteration, our algorithms constructs the $G(360)$-spanning graph in which the path length of each path is greater than 360 as shown in Figure 5. Our algorithm removes the selected nodes v_1 and v_8 from the $G(360)$-spanning graph and finds the minimum node cut $\{v_9, v_{14}\}$ as shown in Figure 6. After our algorithm terminates, it selects the optimal set $\{v_1, v_8, v_9, v_{14}\}$ of nodes (basic blocks) as SPM residents. The resulting longest path length is 360.

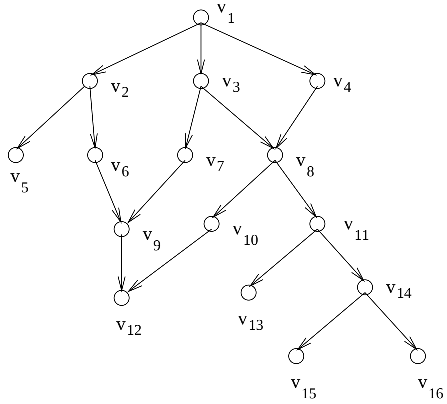

Figure 1: DAG G in Example 1

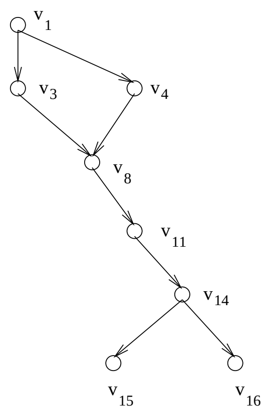

Figure 2: $G(520)$-spanning graph

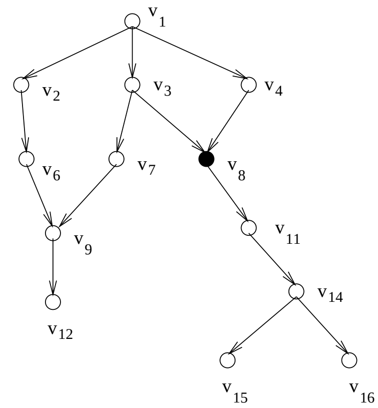

Figure 3: $G(440)$-spanning graph

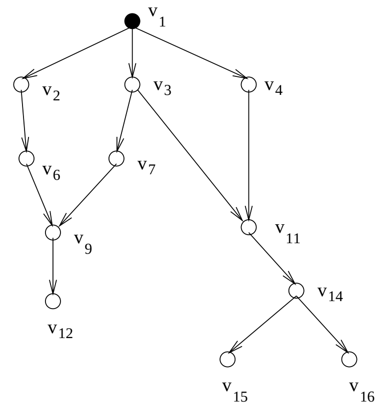

Figure 4: $G(440)$-spanning graph with v_8 removed

3.2 Optimality Proof and Complexity Analysis

THEOREM 3.1. *Given a weighted DAG where all weights are equal, the algorithm OptimalCodeSelection1 is guaranteed to find a subset of nodes as the SPM residents such that the longest path length of the DAG is minimised and the total weight of all the nodes of the subset does not exceed the size of the SPM.*

PROOF. Given a weighted DAG G and a scratchpad size m, let S_{opt} be the optimal set of nodes that minimises the longest path length of G. Assume that S_{opt} reduces the longest path length of G by $k(r-1)(1-\alpha)+s$ $(0 \le s < k(1-\alpha))$, where r is a natural number. S_{opt} can be partitioned into r subsets S_1, S_2, $\cdots$, S_r such that the following conditions hold.

1. For each S_i $(i = 1, 2, \cdots, r-1)$ the longest path length of the DAG G is reduced by $k(1-\alpha)$ if the weight of each node in S_i is changed to $k(1-\alpha)$.

2. The longest path length of the DAG G is reduced by s if the weight of each node in S_r is changed to $k(1-\alpha)$.

Without loss of generality, assume that $|S_1| \le |S_2| \le \cdots \le |S_{r-1}|$. To facilitate our proof, we use the following notations:

G_1: G.

$G_i (i = 2, \cdots, r)$: $G(\cup_{j\in[1,i-1]} S_j, \alpha)$, that is, the DAG G with the weight of each node in $S_1 \cup \cdots \cup S_{i-1}$ being changed to $k\alpha$.

$l_{\max_i} (i = 1, 2, \cdots, r)$: the longest path length of G_i.

$G'_i (i = 1, 2, \cdots, r)$: The $l_{\max_i}$-spanning graph $G_i(l_{\max_i})$ of G_i.

G''_1: G'_1.

$G''_i (i = 2, \cdots, r)$: $G'_i(S')$, where $S' = \{v_j : v_j \in G'_i\} - \cup_{j\in[1,i-1]} S_j$.

Next we show that S_i $(i = 1, 2, \cdots, r)$ must be a minimum node cut of the graph G''_i. When $i = 1$, S_1 reduces the maximum path length of G_1 by $k(1-\alpha)$. Therefore, S_1 must be a node cut of G''_1. Since S_{opt} is an optimal solution, S_1 must be a minimum node cut of G''_1. Similarly, S_i $(i = 2, \cdots, r-1)$ must be a minimum node cut of G''_i.

Now consider S_r. If $S_r = \emptyset$, our proof is complete. Assume that $S_r \ne \emptyset$. In this case, S_r reduces the longest path length by s $(0 \le s < k(1-\alpha))$. Therefore, S_r is not a node cut of the graph G''_r. Since S is the optimal set, S_r must a subset of a minimum node cut of G''_r and all the nodes in S_r must be those nodes in the minimum node cut with larger longest induced path lengths than the remaining nodes in the minimum node cut. $\square$

THEOREM 3.2. *The time complexity of the algorithm OptimalCodeSelection1 is $O(n(ne + n^2 \log n))$, where n and e are the number of nodes and the number of edges, respectively, of the DAG of the loop.*

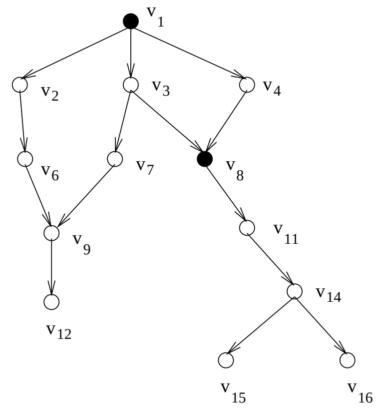

Figure 5: $G(360)$-spanning graph

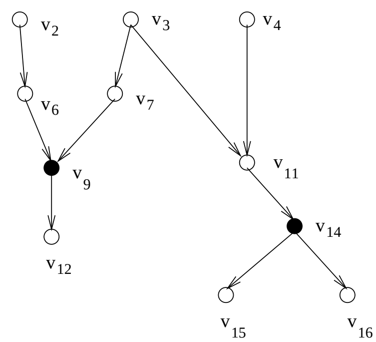

Figure 6: $G(360)$-spanning graph with v_1 and v_8 removed

PROOF. The time complexity of the algorithm OptimalCodeSelection1 is dominated by the **while** loop. The **while** loop consists of the following major parts:

1. Constructing $G(l_{\max} - k(1-\alpha))$. We can use breadth-first search or depth-first search to find the the $(l_{\max} - k(1-\alpha))$-spanning graph $G(l_{\max} - k(1-\alpha))$. Therefore, this part takes $O(e)$ time.

2. Constructing $G'(S)$. This part takes $O(e)$ time.

3. Finding a minimum node cut C of $G'(S)$. The minimum node cut problem can be converted into the minimum edge cut problem[16]. The conversion takes $O(e)$ time. Given a weighted DAG, the minimum edge cut can be found in $O(ne + n^2 \log n)$ time[17], where n and e are the number of nodes and the number of edges, respectively, of the DAG.

4. The **if** part takes $O(n)$ time.

5. The **else** part is executed only once during the execution of the loop. For each $v_i \in C$ it takes $O(e)$ time to compute the maximum length of all the paths that includes v_i. Therefore, this part takes $O(ne)$ time.

The number of iterations of the **while** loop is at most n. Therefore, the time complexity the algorithm OptimalCodeSelection1 is $O(n(ne + n^2 \log n))$. $\square$

3.3 Arbitrary Weights

Next we show how to select code of a non-nested loop with arbitrary block sizes such that the worst-case execution time of the loop is minimised. Our key idea is to split all basic blocks into smaller basic blocks with equal sizes. We first propose a brute-force algorithm for the arbitrary weight problem.

Let $n_1, n_2, \cdots, n_p$ be p different weights of all basic blocks of a loop and c be the greatest common divisor of all the different weights. Let the DAG of the loop is $G = \langle V, E, W \rangle$, where $V = \{v_1, v_2, \cdots, v_n\}$. The brute-force algorithm work as follows:

1. Create a new DAG $G' = \langle V', E', W' \rangle$ as follows:
   (a) For each node $v_i \in G$ create w_i/c new nodes v_{i_j} $(j = 1, 2, \cdots, w_i/c)$ in G'.
   (b) For each pair of nodes v_{i_j} and $v_{i_{j+1}}$ $(j = 1, 2, \cdots, w_i/c - 1)$ in G', create an edge $(v_{i_j}, v_{i_{j+1}})$ in G'.
   (c) For each edge (v_i, v_j) in G, create an edge $(v_{i_{w_i/c}}, v_{j_1})$ in G'.
2. Find the minimum set S of nodes in G' such that the longest path of G' is minimised.
3. For each i $(i = 1, 2, \cdots, n)$ let $c_i = |\{v_{i_j} : v_{i_j} \in S\}|$.
4. For each node v_i $(i = 1, 2, \cdots, n)$ in G, do the following:
   (a) If c_i is equal to w_i/c, select the whole basic block v_i as a SPM resident.
   (b) If $0 < c_i < w_i/c$ holds, split the corresponding basic block of v_i into two basic blocks v_i^1 and v_i^2 such that the size of v_i^1 is $c_i - d$ and the size of v_i^2 is $w_i - c_i + d$, where d is the length of a **jump** instruction that needs to be inserted at the end of v_i^1. Select v_i^1 as a SPM resident.

The **jump** instruction jumps to the start of the basic block v_i^2. Notice that our algorithm splits a basic block into two basic blocks only, i.e., for each basic block, at most one **jump** instruction is inserted in the original program. Therefore, the code size increase caused by the basic block splitting is negligible.

Example 2 Assume that the SPM size is 160 and the speed ratio α of the SPM and the off-chip memory is 0.2. Consider a loop with a DAG G as shown in Figure 7 where each number in brackets is the node weight. Firstly, our algorithm converts the problem into a problem with equal weights as shown in Figure 8, where all weights are equal to 20. Secondly, our algorithm uses the optimal algorithm for the problem with equal weights to find the following set of nodes for the SPM residents: $S = \{v_{1_1}, v_{1_2}, v_{1_3}, v_{5_1}, v_{5_2}, v_{5_3}, v_{7_1}, v_{7_3}\}$. Lastly, our algorithm splits basic block v_7 into two basic blocks: the SPM resident basic block v_7^1 with size of $40 - d$ and the off-chip memory resident basic block v_7^2 with size of $40 + d$, where d is the length of the **jump** instruction. The optimal set of the basic blocks selected by our algorithm is $\{v_1, v_5, v_7^1\}$. The path length of the resulting longest path of G is 113 assuming that d is 1. Notice that if we do not split basic blocks, the path length of the resulting longest path of an optimal solution is 128.

According to THEOREM 3.2, the time complexity of the

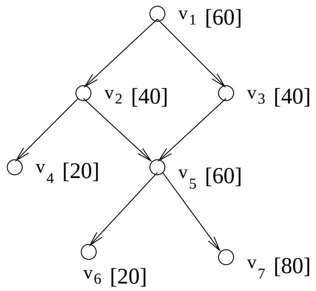

Figure 7: The DAG G in Example 2

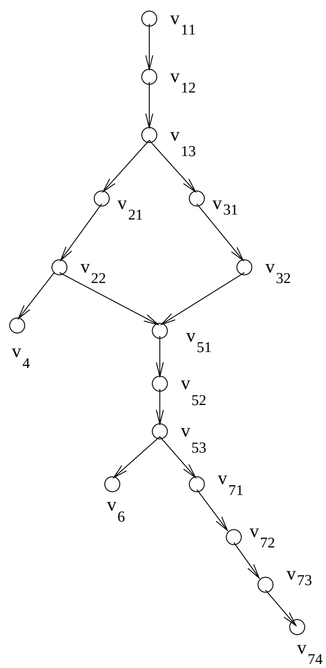

Figure 8: The equivalent DAG in Example 2

brute-force algorithm is $O(nC(nCe' + (nC)^2 \log nC))$, where C is the maximum block size, and e' is the number of edges in G'. If the maximum block size C is very big, the time complexity of the brute-force algorithm is quite high. Next we show how to reduce the time complexity of the brute-force algorithm significantly.

The key idea of our faster algorithm is that the original DAG implicitly keeps the structure of the equivalent DAG with equal weights constructed by the brute-force algorithm. Let $G = \langle V, E, W \rangle$ be the DAG with arbitrary node weights and $G' = \langle V', E', W' \rangle$ be the DAG with equal weights created by the brute-force algorithm. Assume that each node v_i in G is split into m_i nodes $v_{i_1}, \cdots, v_{i_{m_i}}$ in G'. v_i is called the *originator node* of v_{i_j} $(j = 1, \cdots, m_i)$. v_{i_j} $(j = 1, \cdots, m_i)$ is called an *offspring node* of v_i, and v_{i_j} is called a sibling node of v_{i_k} $(j, k = 1, \cdots, m_i)$.

Let S be a subset of V and S' a subset of V'. If each node in S' is an offspring node of a node in S and each node in S is an originator node of a node in S', S' is called an *offspring set* of S and S is called the *originator set* of S'. We can prove the following property:

Property 3.1. *Given a subset S' of V', S' is a minimum node cut of G' iff the originator set S of S' is minimum node cut of G.*

This property suggests one of the two key ideas of our faster algorithm for a non-nested loop with arbitrary weights, that is, we can use the original DAG G to find the minimum cut of a subgraph $(l_{\max} - k(1-\alpha))$-spanning graph of G'. Thus, we can find a minimum node cut of a subgraph of G' much faster than the brute-force algorithm. To do so, we introduce a variable u_i for each node v_i of G. u_i is used to keep track of the size of the part of v_i that has been selected as the SPM resident. The second key idea of our faster algorithm is to reduce the number of executions of the algorithm for finding the minimum node cut of a subgraph of G'. Let $l_{\max}$ and $l_{\max_2}$ be the lengths of the longest path and the second longest path of the current DAG G, respectively. Clearly, for any numbers x_1 and x_2 in $[l_{\max_2}, l_{\max}]$ the x_1-spanning graph and the x_2-spanning graph of G' are identical. Therefore, we just need to find $l_{\max} - l_{\max_2}$ minimum node cuts of the $l_{\max} - k(1-\alpha)$-spanning graph of G'. Assume that $C = (v_{i_1}, v_{i_2}, \cdots, v_{i_p})$ is a minimum node cut of the $l_{\max} - k(1-\alpha)$-spanning graph of G. Let s be $\min\{w_{i_j} - u_{i_j} : w_{i_j}$ is the weight of v_{i_j} and u_{i_j} is the size of the part of v_{i_j} that has been selected as the SPM resident $\}$. Clearly C implies s minimum node cuts in G'.

Based on these two key ideas, our after algorithm for a non-nested loop with arbitrary weights is shown as follows:

Algorithm *OptimalCodeSelection*$2(G, A, m, r)$
input: *A weighted DAG G with arbitrary weights and a set of basic blocks that have been selected as the SPM residents, the number of iterations r of the loop, and the scratchpad size m.*
output: *A subset S of nodes of G as SPM residents for minimising the longest path length and the size of the free space of the SPM.*

begin
  size $= \sum_{v_i \in A} w_i$;
  if *size* $= m$
    return $(\emptyset, 0)$;   /* No free SPM space */
  $S = A$;
  for *each* $v_i \in V$ **do**
    $u_i = 0$;
  compute the longest path length $l_{\max}$ and the second longest path length $l_{\max_2}$ of G;
  if $l_{\max} - l_{\max_2} < 1$
    $d = 1$;
  else
    $d = l_{\max} - l_{\max_2}$;
  while *size* $< m$ **do**
    construct $G(l_{\max} - k(1-\alpha))$;
    let G' *be* $G(l_{\max}(G) - k(1-\alpha))$;
    construct $G'(S)$;
    find a minimum node cut C of $G'(S)$;
    if $|C| >= r$ /* Do not load C into the SPM */
      break; /* Exit from the loop */
    $s = \min\{w_i - u_i : v_i \in C\}$;
    if $s \leq d$
      $r = \min\{\lfloor m - size \rfloor / |C|, s\}$;
      for *each* $v_i \in C$ **do**
        $u_i = u_i + r$;
        if $u_i = w_i$
          change the weight w_i of v_i to $k * \alpha$;
     $size = size + |C| * k * r$;
     $C = C - \{v_i\}$;
    $l_{\max} = l_{\max} - k * r * (1-\alpha)$;
      if *size* $< m$ *and* $|C| \neq \emptyset$
        for *each node* $v_i \in C$ **do**
          compute the maximum length of all the paths that include v_i;
        sort all nodes in C in non-increasing order of their maximum lengths;

```
            let B be the set of the first ⌊(m − size)/|C|⌋ nodes
            in the sorted set C;
            for each node v_i ∈ B do
              u_i = u_i + 1;
  for each node v_i ∈ V do
    if u_i = w_i
      S = S ∪ {v_i};
    else
      if u_i > 0
        split the corresponding basic block of v_i into two
        basic blocks v_i^1 and v_i^2 such that the size of
        v_i^1 is c_i − g and the size of v_i^2 is w_i − c_i + g,
        where g is the length of the jump instruction that
        needs to be inserted at the end of v_i^1;
  inserts the fetch instructions in the preheader of the loop to
  load all the basic blocks in S;
  return (S, m − size);
end
```

Since this algorithm is equivalent to the brute-force algorithm and the brute-force is guaranteed to minimise the worst-case execution time of a non-nested loop with arbitrary weights, the following theorem holds.

THEOREM 3.3. *Given a weighted DAG G with arbitrary weights, the algorithm OptimalCodeSelection2 is guaranteed to find an optimal set of nodes as the SPM residents such that the longest path length of the DAG is minimised and the total weight of all nodes of the optimal set does not exceed the size of the SPM.*

Given a weighted DAG G, the time complexity of each iteration of the **while** loop is $O(ne + n^2 \log n)$ as we explained before. The number of iterations of the **while** loop is at most m/r, where r is the greatest common divisor of all the block sizes. Therefore, the following theorem holds.

THEOREM 3.4. *The time complexity of the algorithm OptimalCodeSelection2 is $O(m(ne + n^2 \log n))$, where n and e are the number of nodes and the number of edges, respectively, of the DAG, and m is the size of the scratchpad memory.*

4. CODE SELECTION FOR LOOP NEST

In this section, we propose a dynamic code selection heuristic for minimising the worst-case execution time of a loop nest by using our algorithm for a non-nested loop. First we introduce *loop nest tree*.

DEFINITION 4.1. *Given a loop nest, its loop nest tree $T = \langle V, E, W \rangle$ of L_1, is a weighted tree where $V = \{L_i : L_i$ is a loop in the loop nest$\}$, $E = \{(L_i, L_j)$: L_j is immediately nested within loop L_i $\}$, $W = \{w_i$: the size of loop L_i is $w_i\}$.*

The size of a loop is the sum of sizes of all the basic blocks of the loop, excluding the preheader.

For a non-innermost loop L_i, its DAG $G(L_i)$ is defined as follows. $G(L_i) = \langle V(L_i), E(L_i), W(L_i) \rangle$, where each node $v_i \in V$ denotes either a basic block immediately nested in L_i or a loop immediately nested in L_i, $E(L_i)$ is the set of all forward edges of the CFG of L_i, and W is the set of weights of all nodes. For a node that represents a basic block, its weight is the size of the corresponding basic block. For a node that represents a loop, its weight is the worst-case execution time of the loop which is determined only after our heuristic is applied to the loop. Consider a loop nest in C shown as follows:

```
while (a*a + b*b > c)                    /* L1 */
  {
    if (x > y)
      for (i = 0; i <= 200; i++)              /* L2 */
        {...;}
    else
      for (i = 0; i <= 200; i++)              /* L3 */
        {...;}
    for (i = 0; i <= 100; i++)                /* L4 */
      {
        if (a > b)
          for (i = 0; i <= 300; i++)          /* L5 */
            {...; }
        else
          for (i = 0; i <= 300; i++)          /* L6 */
            {...; }
      }
  }
```

There are 6 loops $L_i (i = 1, 2 \cdots, 6)$ in this loop nest. The loop nest tree is shown in Figure 9. The DAG of L_1 is shown in Figure 10 where a circle denotes a basic block node, a rectangle denotes a loop node.

To facilitate descriptions, we use following notations:

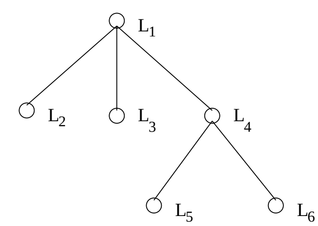

Figure 9: A loop nest tree

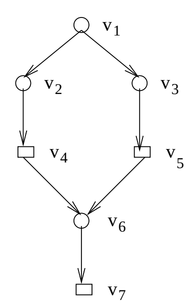

Figure 10: The DAG $G(L_1)$

$B(L_i)$: all the basic blocks, excluding the preheader block, of L_i.

$l_{\max}(L_i)$: the length of the longest path of the DAG of L_i after our algorithm selects basic blocks as the SPM residents for L_i.

n_i: the number of iterations of L_i in the worst-case.

$child(L_i)$: the set of all children of L_i in the loop nest tree of L_i.

Our heuristic works in the reverse topological sort order of the loop nest tree, that is, from inner most loops to the outmost loop. When our heuristic visits a loop, it uses our optimal algorithm for a non-nested loop to find the optimal set of basic blocks of the loop. Then it reduces the loop into two basic blocks: the preheader block and the loop block. The preheader block contains the instructions that load all the selected basic blocks of the loop into the SPM. The loop block is an artificial block whose size is equal to the worst-case execution time of the loop. The worst-case execution time of the loop is equal to the number of iterations of the loop multiplied by the maximum path length of the DAG of the loop computed by our optimal algorithm for a single loop. Note that our heuristic never selects a loop block as a SPM resident. Our heuristic is recursively shown in pseudo code as follows.

Algorithm $HeuristicforLoopNest(L)$
input: *A loop* L
output: *A set* S *of basic blocks of* L *selected as the SPM residents*

begin
  if $child(L) = \emptyset$
    $(S, s) = OptimalCodeSelection2(G(L), \emptyset, m)$;
    return (S, s);
  $S = \emptyset$;
  for *each loop* $L_i \in child(L)$ **do**
    $(S_i, s_i) = HeuristicforLoopNest(L_i)$;
    set the weight of the node denoting L_i *in* $G(L)$
    to $l_{\max}(L_i) * n_i$;
    $S = S \cup \{v_i : v_i$ *is the node denoting* L_i *in* $G(L)\}$;
    $B = B \cup S_i$;
  $s = \min\{s_i : L_i \in child(L)\}$;
  if $s > 0$
    $(S, s) = OptimalCodeSelection2(G(L), S, k)$;
    inserts the **fetch** *instructions in the preheader of the loop* L *to load all the basic blocks in* S;
    return (S, s);
end

Consider the loop nest shown in Figure 9. Assume that the size of the SPM is 4 KB, the sizes of the loops L_2 and L_3 are 4 KB, and the size of L_4 is 1.6 KB. By our heuristic, the SPM will be shared by L_2, L_3 and L_4 as shown in Figure 11.

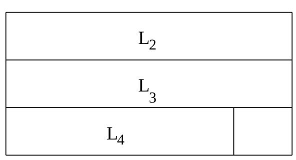

Figure 11: A dynamic SPM allocation scheme

5. COMPARISON WITH RELATED WORK

The problem of allocating code of a task to SPM has been studied by a number of researchers. Two optimisation objectives have been used. One is to minimise the average energy consumption or execution time of a task. The other is to minimise the worst-case execution time of a task.

Egger, Lee and Shin studied the problem of dynamic SPM management for the code of a task aiming to minimise the average execution time of the task[6]. The target systems have an MMU and use a scratchpad memory and a small minicache to replace the on-chip instruction cache. They proposed a dynamic memory allocation technique for a horizontally partitioned memory subsystem. The proposed technique uses the profiling information to classify the code into a pageable and a cacheable code region. The cacheable code region is placed at a fixed location in the offchip memory and cached by the minicache. The pageable code region is copied on demand to the SPM before execution. Both the pageable code region and the SPM are logically divided into pages. Using the MMU's page fault exception mechanism, a runtime scratchpad memory manager tracks page accesses and copies frequently executed code pages to the SPM before they are executed.

Angiolini et al. proposed an optimal scratchpad mapping approach for code segments[4]. The mapping approach applies the dynamic programming algorithm to the execution traces of the target application. The mapping approach is able to find the optimal set of basic blocks to be moved into a dedicated SPM, either minimising the energy consumption or execution time of the target application.

Janapsatya et al. proposed a heuristic which aims to minimise the average total energy consumption of a program[10]. Their heuristic uses the profiling information of a program and converts the code selection problem into a graph partitioning problem. They also proposed a better heuristic for selecting code with the same optimisation objective[9]. The heuristic introduces a novel metric called concomitance to find basic blocks which are executed frequently and in close proximity in time.

The primary design goal of real-time embedded systems is to ensure that all timing constraints are satisfied. Given a target hardware platform, the satisfiability of timing constraints is determined by the worst-case execution times of all tasks. All the above-mentioned code selection approaches aim to minimise the average execution time or total energy consumption of a program. They rely on the profiling information of the program and try to select the most frequently executed basic blocks as scratchpad residents. the worst-case execution time of a task is determined by its longest execution paths. typically the longest execution paths of a program are not the most frequently executed paths. In other words, selecting most frequently executed basic blocks of a program as scratchpad residents may not reduce its worst-case execution time. Therefore, all the afore-discussed code selection approaches are not suitable for real-time embedded systems.

The only previous work on the code selection for real-time systems was done by Puaut and Pais[15]. They proposed a heuristic for the problem of selecting basic blocks of a loop such that the worst-case execution time of the loop is minimised. The main idea of their heuristic is to repeatedly select as a SPM resident a basic block with the highest frequency on the longest path of the loop until the SPM has no more free space for a basic block.

Their heuristic does not consider sharing the SPM among all the inner loops, resulting a lower SPM utilisation. In the worst-case, their heuristic may perform very poorly. Consider Figure 12. Assume that the frequencies of all basic blocks are equal, the sizes of all basic block are equal to k and the size of SPM is $(2n+1)k$. Their heuristic may select $\{v_2, v_3, v_5, v_6, \cdots, v_{3n-2}, v_{3n-1}\}$, leaving the longest path length unchanged. An optimal set of nodes is v_1 and any $2n$ nodes in $\lceil 2n/3 \rceil$ node cuts of size 3.

To make a quantitative comparison between our heuristic for a loop nest and the heuristic proposed by Puaut and Pais, we simulated both heuristics by using the SimpleScalar simulator[3]. We modified the instruction cache part to carter for code SPM and disabled data cache of the SimpleScalar. The target processor uses the PISA instruction set with single-issue in-order pipeline. The off-chip memory latency is 10 cycles.

We selected four benchmarks: *susan*, *statemate*, *compress* and *jfdctint* from the benchmark suites maintained by the Mälardalen WCET research group[1]. We modified the main

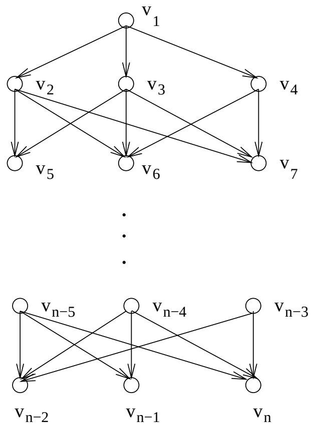

Figure 12: A worst-case scenario

function of *statemate* to include a loop that calls the functions *interface* and *FH_DU* 20 times. The simulation was performed on Intel(R) Xeon(R) Dual Core CPU 5160 with a clock frequency of 3 GHz and 4 MB cache. We used different SPM sizes for different benchmarks based on the size of the largest loop of a benchmark. For the benchmarks *susan*,*statemate* and *compress* 3 different SPM sizes are 1 KB, 2 KB and 4 KB. For the benchmark *jfdctint* 3 different SPM sizes are 256 bytes, 512 bytes and 1 KB. For the selected benchmarks the running times of our algorithm are negligible. The simulation results are shown in Figures 13-16, where an OPT bar and a PP bar indicate the estimated numbers of processor cycles of a benchmark in the worst case given a particular SPM size by using our heuristic and the heuristic proposed by Puaut and Pais, respectively.

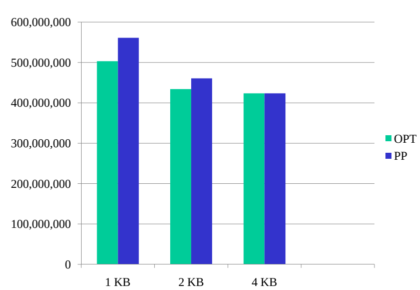

Figure 13: susan

Based on our simulation results we have the following observations:

1. For these four benchmarks the largest performance improvement of our heuristic over Puaut and Pais's is 20%. In general, the improvement becomes smaller when the SMP size approaches the size of the largest outmost loop of the benchmark. When the SPM size is no less than the size of the largest outmost loop of the benchmark, both heuristics have the same performance.

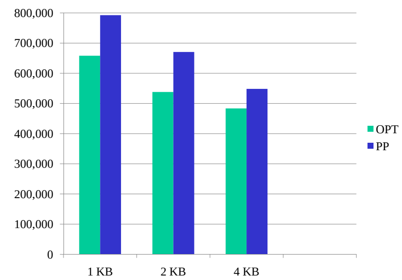

Figure 14: statemate

2. Our novel minimum node cut-based strategy works better if the control flow graph of a loop is wide. Such an example is *statemate*.

3. Our basic block splitting strategy performs better when there are big basic blocks in a loop with respect to the SPM size. such an example is *jfdctint* where the size of the largest basic block exceeds 512 bytes.

4. Our heuristic performs better for loop nests where multiple inner loops of the same level are nested in an outer loop.

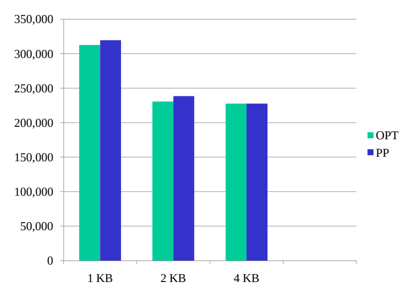

Figure 15: compress

6. CONCLUSION

In this paper, we studied the problem of minimising the worst-case execution time of a loop nest executed on a processor that uses SPM to replace the instruction cache. We proposed the first polynomial-time algorithm for selecting the code of a non-nested loop such that the worst-case execution time is minimised on a fully pipelined processor. For non-pipelined processors or non-fully-pipelined processors our optimal algorithm is suboptimal. Our optimal algorithm uses a novel approach to splitting basic blocks and converts the optimal code selection problem into the problem of finding the minimum node cuts of a set of weighted DAGs. Furthermore, we proposed a dynamic code selection heuristic for minimising the worst-case execution time of a

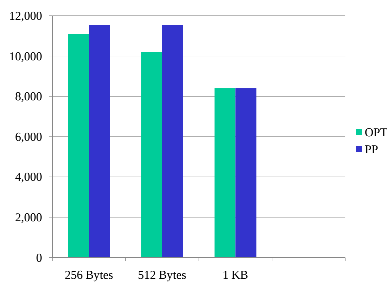

Figure 16: jfdctint

loop nest by using our algorithm for a non-nested loop. We have performed simulations of our dynamic code selection heuristic and the heuristic proposed by Puaut and Pais [15] on four selected benchmarks. Simulation results show that our dynamic code selection heuristic performs significantly better.

In real-time embedded systems multiple tasks may run concurrently on one processor. As a result, the SPM is shared by all the tasks. An open research problem is how to efficiently share the SPM among all the tasks.

7. REFERENCES

[1] Mälardalen wcet research group. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[2] Nvidia physx for developers. `http://developer.nvidia.com/object/physx.html`.

[3] Simplescalar lic. `http://www.simplescalar.com/`.

[4] 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 conference on Compilers, architecture, and synthesis for embedded systems* (2004), pp. 259–267.

[5] 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.

[6] 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 (TECS) 7*, 2 (2008).

[7] 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 Design Automation Conference* (2004), pp. 238–243.

[8] 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 conference on Design automation* (2004), pp. 238–243.

[9] Janapsatya, A., Ignjatovic, A., and Parameswaran, S. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation* (2006), pp. 612–617.

[10] 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.

[11] Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D. J. Introduction to the cell multiprocessor. *IBM Journal of Research and Development 49*, 4/5 (2005), 589–604.

[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 Design Automation Conference* (2001), pp. 690–695.

[13] Li, L., Gao, L., and Xue, J. Memory coloring: A compiler approach for scratchpad memory management. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques* (2005), pp. 329–338.

[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*, pp. 207–26.

[15] Puaut, I., and Pais, C. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of DATE 2007*, pp. 1484–1489.

[16] Skiena, S. S. *The algorithm design manual*. Springer, 1998.

[17] Stoer, M., and Wagner, F. A simple min-cut algorithm. *Journal of the ACM 44*, 4 (July 1997), 585–591.

[18] Suhendra, V., Mitra, T., and Roychoudhury, A. Wcet centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE Real-Time Systems Symposium* (2005), pp. 223–232.

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

[20] Verma, M., and Marwedel, P. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 4*, 8 (August 2006), 802–815.