WCET centric Scratchpad memory allocation

Sireesha R Basavaraju  
Embedded Systems Group, Technical University of Kaiserslautern  
basvaraj@rhrk.uni-kl.de

Abstract

Small and fast scratchpad memories are popular in real-time embedded systems. They are different from caches, as allocation of data to scratchpad memory must be handled by software. Scratchpad memories are used to enhance the predictability of memory accesses latency. In this report we discuss different scratchpad memory allocation techniques for program code and data. These techniques mainly aim at reducing the worst-case execution time of programs. Worst-case execution time is a key metric in real-time embedded systems. We also discuss an instruction allocation technique for precision timed architecture. Four out of six techniques discussed here are based on integer linear programming and hence can provide optimal allocation.

1 Introduction

Real-time(RT) embedded systems include applications like automotive engine control, aircraft flight control, etc. The correctness of such a system depends on timeliness of logical results. Such systems demand predictability of software and underlying hardware. To guarantee that safety of RT system is met, worst case execution time (WCET) analysis of program is performed.

In order to bridge the gap between processor speed and off-chip memory, on-chip memories are employed in embedded systems. Typically caches are used as on-chip memories in embedded systems. While caches are hardware controlled, they make WCET analysis of programs very difficult due to the unpredictability of the access latency of cache. As an alternative to caches, scratchpad memories (SPMs) are used in RT systems. SPM are small on-chip memories that are mapped into the address space of the processor as shown in Figure 1. The access latency to SPM is predictable when compared to caches. Also SPMs are more energy efficient when compared to caches. With the use of SPMs, the decision of placing data and code objects now lies in hands of compiler.

Efficient allocation techniques for scratchpad memory that have been developed so far aim to reduce the average-case execution time (ACET) by utilizing extensive data memory access profiles. An optimal allocation for ACET may not necessarily be the optimal allocation for WCET. The elegant techniques used in ACET-guided optimal allocations, such as 0-1 knapsack are not applicable for WCET-guided allocation. The main problem in developing optimal SPM allocation technique for WCET is the following. In WCET-guided allocation, we are interested in the access frequencies of the data and code objects along the worst-case path. As as when the code or data objects along the worst-case path are allocated into the SPM, a new path may become the worst-case path. This leads to a different access frequency profile corresponding to the new worst-case path. We see that locally optimizing the current worst-case path may not lead to the globally optimal solution. Hence compiler must be aware of worst-case path changes and provide optimal allocation to SPM.
In this study, we discuss SPM allocation for data and code objects with aim of reducing the WCET of programs in RT systems. Some of the methods described here are based on Integer linear programming (ILP). ILP based approaches are most popular due to the optimality of the results. A static WCET-aware instruction allocation method based on ILP is discussed [3]. In RT systems, Precision timed (PRET) architectures are popular due to their predictable behaviors. PRET provides extended instruction-set architecture which allows explicit timing specifications in programs. In such an architecture reducing WCET does not guarantee the timing requirements explicitly specified in the program. We need a method that allocates code to SPM ensuring that the explicit timing constraints of the program are met. An instruction allocation based on ILP for PRET architecture is discussed [5] which assigns basic blocks in order to ensure the timing requirements are met. This method provides a minimal allocation for a thread so that same SPM can be used for allocating basic blocks from other threads. Static allocation for program data which is based on ILP and allocation technique based on Branch-and-bound algorithm is discussed [6]. It is important to consider the infeasibility information into account when analyzing and optimizing for WCET. ILP based method does not consider infeasible paths of programs. WCET calculation is extended using infeasible path information and this is used while finding the allocation using branch-and-bound search. Lastly a dynamic allocation technique for program data is discussed [2].

The rest of the report is organized as follows: Section 2 presents explanation for some basic terms used. In Section 3 we discuss SPM allocation for program code. Section 4 we discuss SPM allocation for data objects. Section 5 we discuss and compare the approaches and finally Section 6 concludes the report.

2 Background

In a RT embedded system, the time at which a result is computed is as important as the result itself. RT operating systems provide timing-aware scheduling policies, but without precise worst-case execution time (WCET) bounds they cannot provide guarantees. To understand the worst case timing behaviour of software, a designer must provide a guaranteed upper bound of WCET of the software. A program $P$'s WCET is the maximal time $P$'s execution can ever take. The control flow graph (CFG) of $P$, whose nodes represent basic blocks and whose edges indicate that one basic block can be reached from the
other, reflects all possible ways of executing $P$. Among all paths from $P$’s start node in the CFG to some end node, there is one longest path, called worst-case execution path (WCEP), and its length is equal to $P$’s WCET. Here, path length is the sum of the products of WCET and worst-case execution frequency for all basic blocks of the path [6].

SPM allocation for program code for Precision Timed (PRET) architecture is described in [5]. Precision timed (PRET) architecture is a SPARC-based processor with predictable timing and repeatable temporal behaviors. It has instruction-set architecture (ISA) extensions that provide precise timing control. Its pipeline executes multiple, independent hardware threads to avoid costly, unpredictable bypassing, and its exposed memory hierarchy provides predictable latency. ISA extensions that provide precise timing control are presented in Figure 2. Using these instructions we can realize different blocks of code with explicitly specified timing constraints [1]. An example is shown in Figure 3. This timed block means execute this block of code and if during the execution of the block, a specified amount of time is exceeded, branch immediately to an exception handler. We can also have other possibilities:

- Execute a block of code taking at least a specified time
- Execute a block of code. Conditionally branch if the execution of the block exceeded a specified amount of time (the deadline)
- Execute a block of code in at most a specified amount of time (the deadline)

The timing instructions from Figure 2 are used to define timed blocks. A timed block encloses a sequence of instructions that have a specific temporal requirement. A timed block enclose many basic blocks.

SPM allocation techniques can be either static or dynamic. In static SPM allocation the SPM’s contents is computed at compile time and remains fixed during run time. Whereas in dynamic SPM

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_time %r, offset</td>
<td>Load the currentTime+ offset into register %r</td>
</tr>
<tr>
<td>delay_until %r</td>
<td>Stall pipeline until currentTime &gt;= [%r].</td>
</tr>
<tr>
<td>branch_expired %r, target</td>
<td>Conditionally branch to the target if the</td>
</tr>
<tr>
<td></td>
<td>currentTime &gt; [%r].</td>
</tr>
<tr>
<td>exception_on_expire %r, id</td>
<td>Processor throws an exception with id when</td>
</tr>
<tr>
<td></td>
<td>currentTime &gt; [%r].</td>
</tr>
<tr>
<td>deactivate_exception id</td>
<td>Disable exception handler for exception id.</td>
</tr>
</tbody>
</table>

Figure 2: ISA extension for PRET architecture [5]

```c
1  void display(char outbuf[][]) {
2      // v3: rl, 100ns
3      SET_TIME(1, 100);
4      EXCEPTION_EXPIRE(1, 2);
5      writeToVGA(outbuf);
6      DEACTIVATE_EXCEPTION(2);
7      DELAY_UNTIL(1);
8  }
```

Figure 3: Example code for Timed block in PRET [5]
allocation SPM contents change during the run-time.

3 SPM allocation for program code

3.1 WCET-aware optimal static allocation

We present WCET-aware SPM allocation for program code which is an optimal approach because it is based on ILP. ILP modeling of function’s control flow is described here.

### 3.1.1 ILP Formulation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{main}$</td>
<td>WCET of basic block $b_i$, when executed from main memory</td>
</tr>
<tr>
<td>$C_{spm}$</td>
<td>WCET of basic block $b_i$, when executed from SPM</td>
</tr>
<tr>
<td>$s_i$</td>
<td>Growth of basic block for different jump scenarios in program</td>
</tr>
<tr>
<td>$S_{spm}$</td>
<td>Total SPM size</td>
</tr>
<tr>
<td>$S_i$</td>
<td>Size of basic block $b_i$ in its original form</td>
</tr>
</tbody>
</table>

Table 1: Table of symbols used in the allocation

Table 1 shows the meanings of all relevant variables that we use in the formulation. ILP allocation variable $x_i$ is a boolean variable and it is defined for each basic block $b_i$ as follows:

$$x_i = \begin{cases} 
1 & \text{if basic block } b_i \text{ is assigned to main memory} \\
0 & \text{if basic block } b_i \text{ is assigned to SPM} 
\end{cases}$$

The cost of each basic block $c_i$ is either the cost of executing it from the main memory $C^i_{main}$ or from SPM $C^i_{spm}$ and it is given as:

$$c_i = C^i_{main} \times (1 - x_i) + C^i_{spm} \times x_i$$  (1)

For each function, for an innermost loop $L$, the body of loop can be represented by an acyclic graph $G=(V,E)$ where $V$ represents set of all nodes in $G$ and $E$ represents set of all edges in $G$. The WCET of this innermost loop body is calculated by starting from the basic block at the exit and then moving up towards the basic block at the entry of loop. This acyclic path has exactly one unique exit node given by basic block $b^L_{exit}$ and one unique entry node $b^L_{entry}$. The WCET $w^L_{exit}$ of $b^L_{exit}$ is equal to the costs of the node $b^L_{exit}$ given by: $w^L_{exit} = c^L_{exit}$

The WCET of a path leading from any node $b_i$ different from $b^L_{exit}$ to $b^L_{exit}$ must be greater than or equal to the WCET of successor of $b_i$, i.e.,

$$\forall b_i \in V \setminus \{b^L_{exit}\} : \forall (b_i, b_{suc}) \in E : w_i \geq w_{suc} + c_i$$  (2)

The WCET of all paths of loop body of $L$ can be obtained by modeling the basic block at the entry of loop, i.e., $w^L_{entry}$ for one time execution of loop. Using $L$’s maximal loop iteration count $C^L_{max}$, the cost of loop is calculated as follows:

$$c_L = w^L_{entry} \times C^L_{max}$$  (3)
Now this inner loop can be replaced as a single node to obtain modified CFG, which turns the surrounding loop into another acyclic graph and the above procedure is repeated iteratively. By successive collapsing of loops in the CFG we can obtain the WCET of the entire function F.

### 3.1.2 Allocation of consecutive basic blocks

This method also considers the jump penalties which is caused due to placement of consecutive basic blocks in different memories i.e., SPM and main memory. Different jump scenarios like implicit jump, unconditional jump and conditional jump are separately considered to derive the jump penalty $j_{pi}$. Jump penalty $j_{pi}$ is used to extend WCET of basic block as:

$$\forall b_i \in V\backslash \{b_{exit}^L\} : \forall (b_i, b_{suc}) \in E : w_i \geq w_{suc} + c_i + j_{pi} \tag{4}$$

### 3.1.3 Modeling global control flow

It might happen that within a basic block there is a call to another function F. In this case the WCET of this basic block will be cost of itself and the WCET of the function F. Along with this a function call penalty $c_{pi}$ must be considered since branch to the function results in a jump scenario. Considering all the overhead we can arrive finally at the WCET $w_i$ of a basic block $b_i$ as:

$$\forall b_i \in V\backslash \{b_{exit}^L\} : \forall (b_i, b_{suc}) \in E : w_i \geq w_{suc} + c_i + j_{pi} + c_{pi} \tag{5}$$

The overall objective function for ILP solving is obtained by calculating the WCET of entry point of the main function in the program. The variable $w_{main}^{entry}$ denotes the WCET of the program including all the jump and call penalties. The overall aim of ILP will be to minimize the value of this variable under SPM size constraint.

The allocation must assure that size of all basic blocks assigned to SPM must be maximum equal to the size of SPM $S_{spm}$. It is necessary to determine the size of all the basic blocks assigned to SPM. If consecutive basic blocks are not placed adjacently in the same memory then a conventional jump instruction is added in basic block and if they are placed in different memories, then to generate a jump across memories, complex computations and register indirect branches are necessary. This is due to large difference between address spaces of main memory and SPM. Variable $s_i$ is used to model this growth in size of basic block $b_i$. Now constraint equation for ILP is given by:

$$\sum_{b_i} (S_i \cdot x_i + s_i) \leq S_{spm} \tag{6}$$

### 3.2 Instruction SPM Allocation for PRET architecture

Instruction SPM allocation for Precision Timed architecture is described here [5]. The instruction allocation for PRET architecture must be aware of timing requirements specified in the program. The allocation method described here identifies the basic blocks within timed block and allocates them to SPM so that the timing requirements are just met. This allocation is static. This method does not simply reduce the WCET path as done in the previous method but it finds an allocation to ensure that the executions of all timed blocks meet their timing constraints. The allocation considers not only the basic blocks from one thread but basic blocks of all the threads in the system.
### 3.2.1 ILP formulation

ILP is formulated to give an optimal allocation for instructions from multiple threads. The solution should allocate the minimum number of basic blocks such that we just meet our requirements. This allows us to utilize the remaining SPM space for other timed blocks from the same thread, and from other threads.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_t(k)$</td>
<td>Boolean variable representing the allocation of basic block $k$ of thread $t$ to SPM; ‘1’ means $k$ is allocated to SPM</td>
</tr>
<tr>
<td>$g_{p,j,t}$</td>
<td>Auxiliary variable that assists in reducing the difference between the timing requirement and the actual execution time of path $p$ in timed block $j$ of thread $t$</td>
</tr>
<tr>
<td>$F_{p,j}(k)$</td>
<td>Frequency of execution of basic block $k$ in a timed block $j$ in thread $t$</td>
</tr>
<tr>
<td>$T_{t,\text{main}}(k)$</td>
<td>WCET of basic block $k$, when executed from main memory</td>
</tr>
<tr>
<td>$T_{t,\text{spm}}(k)$</td>
<td>WCET of basic block $k$, when executed from SPM</td>
</tr>
<tr>
<td>$K_{p,j,t}$</td>
<td>Set of all basic blocks forming path $p$ in a timed block $j$ of thread $t$</td>
</tr>
<tr>
<td>$H$</td>
<td>Total number of threads</td>
</tr>
<tr>
<td>$J_t$</td>
<td>Total number of timed blocks in thread $t$</td>
</tr>
<tr>
<td>$N_t$</td>
<td>Total number of basic blocks in thread $t$</td>
</tr>
<tr>
<td>$S_t(k)$</td>
<td>Size of basic block $k$ in thread $t$</td>
</tr>
<tr>
<td>$P_{p,j,t}$</td>
<td>Total number of paths of timed block $j$ in thread $t$</td>
</tr>
<tr>
<td>$T_{p,j,t}$</td>
<td>Computed WCET of path $p$ within timed block $j$ in thread $t$</td>
</tr>
<tr>
<td>$R_{j,t}$</td>
<td>Timing requirement for timed block $j$ in thread $t$</td>
</tr>
</tbody>
</table>

Table 2: Table of symbols used in the allocation for PRET

Table 2 shows the meanings of all relevant variables that we use in the formulation. The objective function that needs to be minimized is given by function $A$ in Equation (7). Function $A$ allocates just enough instructions to meet the explicit timing requirements. The variable $g_{p,j,t}$ is an auxiliary variable that assists in reducing the difference between the timing requirement specified by timed block $j$, and the WCET estimate of path $p$ in thread $t$. A negative value of free auxiliary variable $g_{p,j,t}$ suggests a violation of the timing requirements. By minimizing the sum of the absolute value of this variable $g_{p,j,t}$ for all paths $p$ in timed block $j$ and in thread $t$, we reduce the difference between the timing requirements of the timed blocks and the WCETs of the enclosed paths.

$$A = \sum_{t=1}^{H} \sum_{j=1}^{J_t} \sum_{p=1}^{P_{p,j,t}} |g_{p,j,t}|$$  \hspace{1cm} (7)

The objective function is minimized under two constraints given by Equations (8) and (9). The first constraint is SPM size constraint. It states that the sum of the size of all basic blocks allocated to the SPM must not exceed the maximum size of the SPM $S^{\text{spm}}$:

$$\sum_{t=1}^{H} \sum_{k=1}^{N_t} X_t(k) S_t(k) \leq S^{\text{spm}}$$  \hspace{1cm} (8)

The second constraint is used to ensure the explicit timing constraints of the program are met by allocation. It indicates that the difference between the specified timing requirement $R_{j,t}$ of timed block
and the execution time $T_{p,j,t}$ of the timed block is greater or equal to the auxiliary variable $g_{p,j,t}$, i.e.,

$$\forall p \in [1, P], \forall j \in [1, J], \forall t \in [1, H]: R_{j,t} - T_{p,j,t} \geq g_{p,j,t}$$  \hspace{1cm} (9)

By minimizing $g_{p,j,t}$, we reduce the WCET of path $p$ by allocating basic blocks from path $p$ to the SPM. ILP objective function given by Equation (7) is solved under constraints specified by Equations (8) and (9) to obtain static allocation of code.

## 4 SPM allocation for data

### 4.1 WCET centric static allocation based on ILP

WCET-aware static allocation of data variables to SPM is proposed here [6]. It is optimal since it is based on ILP. The allocation considers both scalar variables and arrays. An array can be allocated only if the entire array fits into the scratchpad.

#### 4.1.1 ILP Formulation:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v$</td>
<td>Data variable in program</td>
</tr>
<tr>
<td>$S_v$</td>
<td>0-1 decision variable indicating $v$ is selected for SPM allocation</td>
</tr>
<tr>
<td>allvars</td>
<td>Set of all variables in program</td>
</tr>
<tr>
<td>scratchpad_size</td>
<td>Total size of SPM</td>
</tr>
<tr>
<td>$W_i$</td>
<td>Worst-case cost of basic block $i$ under allocation decision $S_v$</td>
</tr>
<tr>
<td>cost$_i$</td>
<td>Cost of basic block $i$ without any allocation</td>
</tr>
<tr>
<td>gain$_v$</td>
<td>Gain in access cycles of $v$ by allocating to SPM</td>
</tr>
<tr>
<td>vars$_i$</td>
<td>Set of program variables appearing in basic block $i$</td>
</tr>
<tr>
<td>$n_{v,i}$</td>
<td>Number of occurrences of $v$ in basic block $i$</td>
</tr>
</tbody>
</table>

Table 3: Table of symbols used in the allocation

Table 3 shows the meanings of all relevant variables that we use in the formulation. Firstly a scheme for allocating data variables appearing in a single program loop is discussed. Later it is extended to general programs. Similar to method proposed in Section 3, an acyclic graph captures the control flow in the loop body (i.e., the control flow graph of the loop body without the loop back-edge). This graph has a unique source node and a unique sink node. Each path from the source to the sink in the graph is an acyclic path — a possible path in a loop iteration. Each variable $v$ in program is associated with variable $S_v$ which gives whether $v$ is allocated to SPM:

$$S_v \geq 0 \quad S_v \leq 1$$  \hspace{1cm} (10)

The allocation must satisfy the SPM capacity constraint:

$$\sum_{v \in \text{allvars}} S_v \cdot \text{area}_v \leq \text{scratchpad\_size}$$  \hspace{1cm} (11)
Consider a worst-case cost \( W_i \) for each basic block \( i \) in the acyclic graph. For each outgoing edge from \( i \) to \( j \) in the acyclic graph, the following constraint gives the worst-case cost of \( i \):

\[
W_i \geq W_j + \text{cost}_i - \sum_{v \in \text{vars}(i)} S_v \cdot \text{gain}_v \cdot \text{n}_{v,i}
\]

(12)

The sink node of the DAG has no outgoing edges. For the sink node we define its worst case cost \( W_{\text{sink}} \) as follows.

\[
W_{\text{sink}} = \text{cost}_{\text{sink}} - \sum_{v \in \text{vars(sink)}} S_v \cdot \text{gain}_v \cdot \text{n}_{v,\text{sink}}
\]

(13)

We start from sink node and move up towards the source node iteratively calculating the worst-case cost of each basic block by using Equation (12). Clearly, the variable \( W_{\text{src}} \) (for the source node of the acyclic graph) captures the worst-case acyclic path under the allocation given by \( S_v \) variables. Using \( lb \), a known constant denoting the maximum number of loop iterations we now define the objective function for ILP as follows:

\[
W_{\text{src}} \cdot lb
\]

The ILP solver finds the assignment of \( S_v \) variables (i.e., the scratchpad allocation) which minimizes the loop’s worst-case execution time.

The above formulation is extended to whole program by generating the constraints for each innermost program loop as mentioned above. Transform the program’s CFG by converting each innermost loop to a basic block. The cost of each innermost loop is given by the function mentioned above. Next construct the constraints for loops in the next level of loop nesting until the topmost level of loop nesting is reached; this gives all the ILP constraints. The new objective function to be minimized is now: \( W_{\text{entry}} \) where entry is the only entry node in the program’s CFG. This overall objective function will be minimized under the SPM memory constraint given by Equation (11).

This methods considers all paths in the CFG of the program are feasible and obtains an allocation. But there are paths which are infeasible in a program and allocation of variables along such a path does not reduce the WCET of program. Hence there is a need to incorporate this infeasible path information while obtaining the allocation. An allocation based on Branch-and-Bound algorithm is describe in next section. This method considers infeasible path information to obtain allocation [6].

### 4.2 Allocation using Branch-and-Bound algorithm

Branch-and-Bound algorithm can be used to find an allocation as a set \( V \in 2^{\text{allvars}} \) i.e., the power set of \( \text{allvars} [6] \). Table 3 shows the meanings of all relevant variables. Allocation must assure the SPM capacity constraint is maintained which is given by:

\[
\sum_{v \in V} \text{area}_v \leq \text{scratchpad size}
\]

The search space consists of the set of all possible allocations \( V \in 2^{\text{allvars}} \). At each level \( k \) in the branch-and-bound search tree, a decision of including or excluding a variable \( v_k \in \text{allvars} \) into the solution set \( V \) is made. The ordering of variables is given by a decreasing order of their \( \text{bound}_v \) for \( v \in \text{allvars} \). Here \( \text{bound}_v \) measures the potential WCET reduction of a variable \( v \) using its maximum contribution over all execution paths.
A leaf node of the search tree gives a complete allocation $V$. At each level $k$ each node $m$ gives a partial allocation $\text{allocation}(m)$ with the decision about the inclusion of variables $v_1$ up to $v_k$ in $V$. At this node $m$ we calculate the reduced WCET corresponding to this allocation. The reduction in WCET is the difference between the original WCET (without any allocation) and the reduced WCET. During the traversal of the search tree, the maximum WCET reduction achieved so far at any leaf node is kept as a bound $B$. At any non-leaf node $m'$ in the search tree, the maximum possible WCET reduction at any leaf node in the sub-tree rooted at $m'$ is computed by a heuristic function as an upper bound, $UB(m')$. If $UB(m') \leq B$, then the search space corresponding to the sub-tree rooted at $m'$ can be pruned. The choice of the heuristic function $UB$ decides the amount of search space pruning achieved. The search is guaranteed to find the optimal solution, but its complexity in the worst case is as high as that of exhaustive search.

4.2.1 Finding WCET considering Infeasibility path information of program

The branch-and-bound search employs a procedure for finding the WCET path. This procedure is typically invoked many times for generating the SPM allocation. This procedure used to find WCET must incorporate the infeasible path information in the program. A simple calculation method for finding the WCET path is discussed here. First step is to find out certain infeasible path patterns in program. This is done by identifying pairs of branches and/or assignments which are guaranteed to “conflict”, that is, can never lie in an execution trace. The only conditional branches appearing in the conflict relation are of the form variable relational operator constant. Similarly, the only assignments which appear in the conflict relation are of the form variable = constant. For such assignments and branches we can define and detect pair-wise conflict in a natural way. For example, $x = 2$ conflicts with $x > 3$ (with no assignment to $x$ appearing in between). After the conflicting pairs of branches and assignments are found, the second step of the method involves WCET calculation which is similar to the method discussed in Section 4.1.1. Only difference is a conflict list for each path list is maintained to avoid exhaustive path enumeration.

4.3 WCET-aware dynamic SPM allocation for data

A compile-time dynamic allocation approach that enables eviction and placement of data to the scratch-pad memory at run-time is presented here. This approach involves two steps as follows. First,
memory accesses to data along the worst-case execution path of the program are analyzed. Next, a 0/1 ILP problem is formulated to select these data for dynamic scratchpad memory allocation. However, we know the worst-case execution path of the program may change after a data allocation. Consequently, the ILP problem is greedily refined to compute a WCET-directed allocation.

Every program has large amount of data accesses of which most of them are load-store instructions. The addresses of load-store instructions are dynamic and hence change for each execution. Any accesses to static data and stack data are important. Thus firstly we need to calculate the targets for any access types of memory accesses found in programs. Irregular and input dependent accesses types represent a large part of load-store instructions. In order to associate any memory access to (possibly-multiple) data target(s), we have to apply pointer analysis to all pointers definitions of the program inter-procedurally, and to the text of the whole-program with its related libraries \[2\]. After determining the targets of load-store instructions of a program we employ these information to define at compile-time a data allocation in a single scratchpad memory device.

A given program is represented using a generic graph representation known as flowgraph. The granularity of flowgraph will vary based on whether it is built from original call graph of an application or from inter-procedural control flow of application. The chosen representation level of the generated program flowgraph may lead to different placements of memory transfer operations.

### 4.3.1 ILP Formulation:

Here we present the ILP formulation used in this allocation. Table 4 shows the meanings of all relevant variables that we use in the formulation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M )</td>
<td>Size of scratchpad memory</td>
</tr>
<tr>
<td>( G )</td>
<td>Number of static data in application</td>
</tr>
<tr>
<td>( v_i )</td>
<td>( i^{th} ) static data where ( i \in [1,G] )</td>
</tr>
<tr>
<td>( S(v_i) )</td>
<td>Size of variable ( v_i ) in bytes</td>
</tr>
<tr>
<td>( X_{copy}^{v_i} )</td>
<td>Time to transfer variable ( v_i ) between main memory and SPM in cycles</td>
</tr>
</tbody>
</table>

Table 4: Table of symbols used in the dynamic allocation

The following set of binary variables are defined as shown, \( \forall i \in [1,G], \forall j \in [1,E] \):

\[
\begin{align*}
\text{load}_{e_j}^{v_i} & = \begin{cases} 
1 & \text{if data } v_i \text{ is transferred to SPM at beginning edge } e_j \text{ of flowgraph} \\
0 & \text{otherwise}
\end{cases} \\
\text{store}_{e_j}^{v_i} & = \begin{cases} 
1 & \text{if data } v_i \text{ is transferred to main memory at end of edge } e_j \text{ of flowgraph} \\
0 & \text{otherwise}
\end{cases}
\end{align*}
\]

Variables \( \text{load}_{e_j}^{v_i} \) and \( \text{store}_{e_j}^{v_i} \) determine where data \( v_i \) is to be loaded and stored onto SPM respectively.

\[
\begin{align*}
\text{alloc}_{rw}^{v_i}_{e_j} & = \begin{cases} 
1 & \text{if mutable data } v_i \text{ is allocated to SPM on edge } e_j \text{ of flowgraph} \\
0 & \text{otherwise}
\end{cases} \\
\text{alloc}_{ro}^{v_i}_{e_j} & = \begin{cases} 
1 & \text{if read-only data } v_i \text{ is allocated to SPM on edge } e_j \text{ of flowgraph} \\
0 & \text{otherwise}
\end{cases}
\end{align*}
\]
Variables $alloc_{rw} v_i$ or $alloc_{ro} v_i$ give the state modified/not modified of the scratchpad-allocated data $v_i$. A modified data $v_i$ must be transferred back to the main memory on end of allocation.

The objective function to maximize is the sum of contributions to the WCET of all memory accesses to allocated static data in the application minus the cost of transfer operations of data between main memory and scratchpad memory.

$$\sum_{i=1}^{G} \sum_{j=1}^{E} (alloc_{rw} v_i e_j \cdot C_e(v_i) + alloc_{ro} v_i e_j \cdot C_e(v_i) - load^v_{e_j} \cdot X_{copy}(v_i) - store^v_{e_j} \cdot X_{copy}(v_i))$$

This ILP problem is solved under the following constraints:

- **Preliminary constraints**: They prevent inconsistencies on binary variables. Data $v_i$ is allocated on SPM with $alloc_{rw} v_i$ or $alloc_{ro} v_i$ exclusively. $\forall i \in [1, G], \forall j \in [1, E]$: 

$$alloc_{rw} v_i e_j + alloc_{ro} v_i e_j \leq 1$$

The MOD and USE annotations indicate if data $v$ on edge $e_j$ is just read or modified. This directly influences on the problem formulation. We have to unset $alloc_{ro} v_i$ variables for edges that may update this data:

$$alloc_{ro} v_i e_j = 0 \text{ if data is modified}$$

- **Flow constraints**: Consider data $v_i$ allocated on scratchpad memory on adjacent and connected edges $e_{j-1}$ and $e_j$ as shown in Figure 5. In this example, the data is loaded on the execution of $e_{j-1}$ and stored back in main memory on the end of $e_j$’s execution. Now $\forall i \in [1, G], \forall (j-1, j) \in ([1, E], [1, E])$, where $e_{j-1}$ is an incoming edge of $e_j$:

$$alloc_{rw} v_i e_j - alloc_{rw} v_i e_{j-1} - alloc_{ro} v_i e_{j-1} - load^v_{e_j} = 0$$

$$alloc_{rw} v_i e_{j-1} - alloc_{rw} v_i e_j - store^v_{e_j} = 0$$
WCET-centric SPM allocation techniques

\[ alloc_{ro}^{v_i}_{e_j} - alloc_{ro}^{v_i}_{e_{j-1}} - load^{v_i}_{e_j} \]  (19)

Constraint given by Equation 17 enables scratchpad-allocation of data \( v_i \) on edge \( e_j \) if this data was already scratchpad-allocated on the incoming edge \( e_{j-1} \), or if this data is loaded on edge \( e_j \). Constraint 18 ensures that data \( v_i \), updated on edge \( e_{j-1} \), must be stored and transferred to main memory or \( alloc_{rw} \) on next edge \( e_j \). Constraint specified by Equation 19 ensures that data \( v_i \), read-only allocated on edge \( e_j \), must be loaded on this edge \( e_j \) or \( alloc_{ro} \) on incoming edge \( e_{j-1} \).

- SPM size constraint: Constraint specifies that the total sum of the size of all SPM allocated data on all edges must be maximum equal to the size of SPM \( M \) given as:

\[
\sum_{i=1}^{M} (alloc_{rw}^{v_i}_{e_j} \cdot S(v_i) + alloc_{ro}^{v_i}_{e_j} \cdot S(v_i)) \leq M
\]  (20)

In addition to the above constraints, an optional support for dynamically scheduled architectures is provided. Here pipeline modeling should take into account all possible timings for each varying timing instruction, increasing the complexity of the WCET analysis. For example, load-store instructions may have multiple executions latency if possible data targets are stored in different memories with heterogeneous latency. To guarantee unique timing for each load-store instruction, we have to express allocation of any targets data of this load-store instruction to the same level of the memory hierarchy (here, the SPM or the main memory). A constraint is used to remove timing anomalies due to data memory accesses.

\[
\forall j \in [1,E], \forall (i1, i2) \in ([1,G], [1,G]) \text{ where } v_{i1} \text{ and } v_{i2} \text{ are possibly accessed on } e_j \text{ by the same load-store instruction:}
\]

\[ alloc_{rw}^{v_{i1}}_{e_j} + alloc_{ro}^{v_{i1}}_{e_j} - alloc_{rw}^{v_{i2}}_{e_j} - alloc_{ro}^{v_{i2}}_{e_j} = 0 \]  (21)

An optimistic allocation is computed from the ILP problem defined above. But, the worst-case execution path of the program may change after some data allocations. We need to evaluate all possible combinations of data allocations to find the optimal reduction of WCET of the program. Since an exhaustive evaluation of all paths is too time consuming, a greedy heuristic has been proposed \cite{2} that greatly enhances the quality of allocation. Algorithm 1 describes the method.

**Algorithm 1** Iterative dynamic allocation algorithm \cite{2}

1: \( allocations \leftarrow \emptyset \)
2: repeat
3: \( change \leftarrow false \)
4: perform WCET estimation
5: extract information on worst-case execution path
6: generate dynamic allocation ILP problem
7: generate additional Constraints for allocations
8: \( new\_allocation \leftarrow call\_solver\_on\_ILP\_problem \)
9: if \( new\_allocation \neq \emptyset \) then
10: \( change \leftarrow false \)
11: \( allocations \leftarrow allocations \cup \text{ most impacting allocation from } new\_allocation \)
12: end if
13: until \( change = false \)
14: return \( allocations \)
The main idea behind Algorithm 2 is to incrementally refine the ILP problem formulation to support greedy allocation of most impacting data. Initially, the worst-case execution path of the application is determined (line 4) and data accesses information are computed (line 5). An initial ILP problem (line 6) is generated and the problem solver computes a list of data to allocate (line 8). On next iterations, WCET estimation is performed again and constraint given by equation 15 is added to the problem formulation (line 7) to enforce allocation of selected data. The algorithm selects and allocates the most (non allocated) impacting data to the scratchpad memory (line 11). This process is applied iteratively until no more allocation can reduce the WCET of the application.

5 Discussion

This section discusses the results on WCET of programs by using SPM allocation for code and data objects. ILP-based static SPM allocation for program code was tested with a total of 73 different real-life benchmarks [3]. For every single benchmark, SPM sizes of 10%, 20%, ..., 100% of the benchmark’s code size were used. The results show the WCET estimates of all benchmarks produced by aiT [4] resulting from WCET-aware SPM allocator as a percentage of the WCET when not using the SPM at all. The method is able to move the most important loops entirely onto the SPM leading to the highest WCET savings. On an average, for all 73 benchmarks, a steadily decreasing WCETs with increasing SPM sizes was obtained (Figure 6). For a CFG with n nodes, the ILP defined in Section 3.1 has a size of \(O(n^2)\) constraints and variables. The two WCET analyses required to generate the constants \(C_{spm}^i\) and \(C_{main}^i\) (Section 3.1) are more expensive. This allocation does not consider instructions from multiple threads.

SPM allocation technique discussed for PRET architecture was tested with Marlardalen benchmarks modified with timing instructions [5]. Results shows that using this allocation technique for increasing SPM sizes from 0% to 20% of the total SPM size available on PRET, an increase in the fraction of timed blocks that meet their timing requirements is obtained. A second set of experiments were conducted to study the advantage of using the shared SPM, and proposed multi-threaded SPM allocation over having...
WCET-centric SPM allocation techniques

Figure 7: Shared SPM versus Dedicated SPM [5]

dedicated SPM segments for each thread. Figure 7 shows the results. The black bar shows the fraction of timing requirements met for shared SPM, and gray bar shows the deadlines met when each thread has its own dedicated SPM segment. The total SPM size for all threads is the same as the shared SPM. Figure 7 shows that the shared SPM clearly helps in meeting timing requirements, and this allocation leverages the shared SPM by using the unused space of one thread for another thread’s timed blocks.

Static ILP based allocation for program data and allocation based on Branch-and-bound algorithm were tested with six data intensive benchmarks [6]. Three different SPM sizes corresponding to 5%, 10%, and 20% of the total data memory size were used for each benchmark. The WCET reduction of 5 - 80% was obtained using these allocation for the different benchmarks. The reduction in WCET obtained via ILP is typically lesser than the reduction in WCET with branch-and-bound. This is because, the ILP-based method cannot take into account the detailed infeasibility information. Since ILP-based allocation does not take infeasibility information into account it is much faster. The branch-and-bound formulation yields an optimal solution for global WCET optimization. But its complexity is exponential with respect to the number of data variables to be allocated. As such, it is not practical to run the branch-and-bound search to generate scratchpad allocation from among a large number of data variables unless the scratchpad size is relatively small.

Dynamic scratchpad allocation discussed in last section has major benefits for small ratios of SPM sizes over the whole program data working set. The approach is notably profitable to systems with a SPM shared among several RT tasks. The method outperforms the static allocation from 12% to 85%. The iterative allocation algorithm selects data in their performance impact order. Consequently, data with high performance impact have higher chance to get assigned to SPM. The running time to solve ILP problems is problematic for the largest benchmarks studied in this paper. This practically limits the proposed method. Second limitation is granularity of flow graph used. A coarse flow graph induces smaller ILP problems, potentially leading to a lower allocation quality.
6 Conclusion

This report presented some WCET-centric SPM allocation techniques for program code and data objects. Static ILP based instruction SPM allocation improves the current state of the art of SPM allocation. It performs ILP-based SPM allocation of code for the very first time under consideration of jump penalties and variable basic block sizes based on jump scenarios. Instruction allocation for PRET architecture was able to successfully meet explicit timing constraints. Along with that it performs its allocation across multiple threads, which leverages the shared SPM for the multi-threaded PRET architecture over dedicated SPMs. ILP based static allocation of data was able to give optimal allocation considering both scalar variables and arrays. Branch-and bound algorithm based allocation was able to effectively consider the infeasible path information in the program and provided better allocation compared to ILP based static allocation. Dynamic SPM allocation algorithm not only supports static data but also stack data and this approach attempts to reduce WCET of real-time programs with the allocation of most impacting data on their worst-case execution paths. The above described methods can be used for reducing WCET of applications in RT systems, which is a key metric in real-time embedded systems.

References


