WCET-Directed Dynamic Scratchpad Memory Allocation of Data

Jean-François Deverge and Isabelle Puaut
Université Européenne de Bretagne / IRISA, Rennes, France

Abstract

Many embedded systems feature processors coupled with a small and fast scratchpad memory. To the difference with caches, allocation of data to scratchpad memory must be handled by software. The major gain is to enhance the predictability of memory accesses latencies. A compile-time dynamic allocation approach enables eviction and placement of data to the scratchpad memory at runtime.

Previous dynamic scratchpad memory allocation approaches aimed to reduce average-case program execution time or the energy consumption due to memory accesses. For real-time systems, worst-case execution time is the main metric to optimize.

In this paper, we propose a WCET-directed algorithm to dynamically allocate static data and stack data of a program to scratchpad memory. The granularity of placement of memory transfers (e.g. on function, basic block boundaries) is discussed from the perspective of its computation complexity and the quality of allocation.

1. Introduction

Worst-case execution time (WCET) of a program is the maximum time this program may take to execute on a specific hardware platform [14, 25, 28]. Knowing program's WCET is of prime importance for hard real-time systems to guarantee computations will complete before their deadline.

Direct-addressed scratchpad memories are being used as an alternative to processor caches as they consume less area and less power. Approaches for static [2] and dynamic [15, 26, 27] allocations have been designed to automatically place code and data on scratchpad memories. So far, many studies have been led on allocation of code and data on scratchpad memory for *average* execution time [2] or energy reduction [27]. A study [28] has demonstrated the superiority of scratchpad memory placement on some cache modeling techniques for execution time predictability of hard real-time systems. Recently, algorithms for *static* data allocation on scratchpad memories in [25], and for dynamic *code* allocation in [21] have been specially designed for WCET optimization. As far as we know, no *dynamic* scratchpad memory *data* allocation methods for WCET optimization have been proposed. In this paper, we present an approach to allocate program data to scratchpad memory for WCET reduction.

Our approach determines at compile-time the possible program locations where data will be transferred on and off the scratchpad memory at runtime in a two-steps method. First, memory accesses to data along the worst-case execution path of the program are analyzed. Second, a 0/1 integer linear program (ILP) problem is formulated to select these data for dynamic scratchpad memory allocation. However, 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.

The two steps of the method are described in Section 2 and 3. In Section 2, we propose a compiler technique to determine potential targets of any data memory accesses of a program. These information are employed to estimate the profit for a data allocation. Section 3 describes the approach for dynamic scratchpad memory allocation. Section 4 provides some results and studies the performance improvements of our proposal over previous scratchpad memory allocation techniques. Section 5 overviews related work while Section 6 describes future work and concludes.

2. Determination of load-store instructions targets

On many programs, a large amount of data accesses are dynamic; the target address of load-store instructions may change for each execution. Table 1 gives a two-dimensional classification of data storage and load-store accesses from [18]. The *storage type* defines the location of a given data. Static data, stack data and heap data are respectively stored in global, heap and stack sections of the program memory space layout. Literals are compiler-generated constants stored in the code section; these data are used to reduce the size of the program code.

Storage type	Description
Static	Global and static structures.
Stack	Function stack frame, spilled temporaries and stack allocated structures.
Heap	Dynamically allocated structures on the heap.
Literals	Constants stored in program code section.

Access type	Explanation
Scalar	Only one element.
Regular	Array accessed by regular, stride accesses.
Irregular	Non-regular but still input data independent.
Input dependent	Reference directly depends on input data

Table 1. Data structure classification based on storage type (upper table) and access type (lower table) [18].

The *access type* defines the way a data is accessed. Scalar access types are accesses to a *unique* data address for statics and to a *relative* address to stack frame base addresses for stack data. The access type of a load-store instruction is regular if this instruction is accessing to multiple elements of a unique array with a constant stride (classified as *linear address sequence* accesses in [20]). Irregular accesses include accesses to (possibly *multiple*) data through pointers and are still independent to the input data. Lastly, input dependent accesses include any accesses with addresses computed at runtime from unknown input data (mentioned as *indirect address sequence* accesses in [20]).

In the next section, we will motivate the need for a method to analyze any data memory accesses of programs.

2.1. Quantitative study of data memory accesses by types

Table 2 gives the impact of data by access types and storage types on the worst-case execution path of programs. Benchmark programs are individually described later in Section 4 and these programs *don't* access heap data. Programs are compiled for the StrongARM-110 [22] with loop-related optimizations (loop unrolling, etc) disabled. WCET analyses of programs are performed with the Heptane WCET timing analyser [6].

	Static		Stack		
Benchmark	Scal. or reg.	Irreg. or input dep.	Scal. or reg.	Irreg. or input dep.	Literals
Adpcm	17.0%	60.0%	9.1%	-	13.0%
Engine	16.9%	-	67.3%	3.0%	12.9%
G721	28.5%	8.6%	39.1%	-	23.7%
Histogram	99.9%	-	0.1%	-	-
Lpc	96.1%	-	0.5%	-	3.4%
Pocsag	62.4%	0.4%	13.9%	-	23.3%
Spectral	31.7%	24.5%	37.8%	-	6.1%
Statemate	60.4%	-	6.1%	-	33.5%

Table 2. Impact ratio of load-store instructions by storage types and access types.

Table 2 presents the ratio of accesses to static data, stack data and literals along the worst-case execution path. To illustrate the partition between accesses with or without pointers, we have respectively merged results for scalar/regular and irregular/input dependent accesses into two sub-categories for static and stack data.

The ratio of load-store instructions to literals is up to 33.5% and is important for most programs of the benchmarks set. On the one hand, most programs have a large amount (between 16.9%-99.9%) of data accesses to statics. On the other hand, stack data represent a large part of data accesses for three of the eight benchmarks (between 33.2%-69.8%).

All programs, except Histogram, make use of irregular/input dependent accesses (e.g. memory accesses through pointers). Moreover, two benchmarks programs make intensive (24.5% and 60.0%) use of such accesses. As a conclusion, any accesses to static data and stack data are important. We have to propose a method to calculate the targets for any access types of memory accesses found in programs.

2.2. Calculation of targets of data memory accesses

In programming languages such as C or C++, programs typically employ pointers to arrays elements, dynamic data structures (e.g. linked lists) and procedures parameters. As shown in the previous section, irregular and input dependent accesses types represent a large part of load-store instructions.

Traditionally, pointer analysis has been used in compilers to build aliasing information [4, 11]. In this paper, we propose to reuse *existing* pointer analysis methods of the compiler to determine possible data accessed by any load-store instructions of a program. In order to *exhaustively* associate any memory access to (possibly-multiple) data target(s), we have to apply pointer analysis to (i) all pointers definitions of the program *interprocedurally*, and to (ii) the text of the *whole-program* [11] with its related libraries.

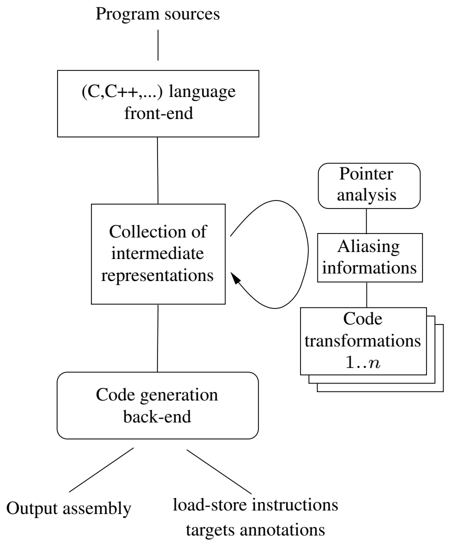

Program sources
(C,C++,...) language front-end
Pointer analysis
Collection of intermediate representations
Aliasing informations
Code transformations $1..n$
Code generation back-end
Output assembly
load-store instructions targets annotations

Figure 1. Pointer analysis in compilation process.

As shown on the Figure 1, a compiler infrastructure typically contains a collection of intermediate representations [10]. A set of code transformations is applied iteratively on intermediate representations. Pointer analyses must be processed on early phases of program transformations; these information are brought through the rest of optimizations phases as annotations to the intermediate representations. Then, the code generation backend phase translates a low-level intermediate representation (similar to [12]) to the output assembly file.

GCC 4.1[1] supports whole-program compilation and it currently provides an *intraprocedural* pointer analysis [4]: targets of pointers passed on procedures parameters are not computed. For the aim of this paper's study, we have slightly modified the compiler infrastructure to apply the pointer analysis interprocedurally ; the compiler keeps results of

[1] GCC – GNU C Compiler: `http://gcc.gnu.org`.

pointer analyses during the whole compilation lifetime. We have also modified the ARM backend to produce the set of possible pointers targets for each generated load-store instruction in the output assembly file.

The pointer analysis applied in this paper supports static and stack storage types. None of our real-time benchmarks make use of dynamic heap allocation. In our study, we don't make the differentiation between individual elements of arrays and between the fields of data structures (whereas such information is computed in the current pointer analysis implementation of the compiler [4]). Moreover, stack data of the whole stack frame of each function are managed as an individual data structure instance.

2.3. Related work on determination of load-store instructions targets

Some approaches have previously succeeded to generate information on some access types. Disassembly of binary files enables extraction of scalar accesses to static data; some dataflow analyses techniques have been applied on assembly code to extend scalar access detection to stack data [3,9,13]. Data dependence analyses techniques and loop induction analyses have been applied on the low-level representation of VPO [19] to determine regular accesses in [29].

[24] uses a processor simulator to generate program memory profile. The profile contains the trace of all memory addresses accessed. The trace must cover all instructions of the program and directly associates an observed target data address for each load-store instruction. This approach enables analysis of any scalar memory accesses. Non-scalar accesses calculation is possible by checking the inclusion of the caught address to any known data's range addresses. This approach guarantee to detect the target of any memory accesses to a *unique* data.

An external module, based on abstract interpretation techniques [5], has been employed on the intermediate representation of the SUIF compiler [30] for pointer analysis. The results of this analysis are re-associated after code generation to the output assembly within the execution of SimpleScalar simulator. Their approach is the most similar to ours. The aim of their study is the impact of memory access aliasing information for scheduling of processor memory request queue [5].

3. Dynamic scratchpad memory allocation

The previous section has presented a complete program analysis framework to determine the targets of load-store instructions of a program. In this section, we employ these information to define at compile-time a data allocation in a single scratchpad memory device. First, we describe the program flowgraph representation considered (Section 3.1). Second, a 0/1 integer linear program (ILP) formulation is given for the allocation of static data (Section 3.2) based on initial knowledge of frequencies along worst-case execution path. We apply this formulation on the considered flowgraph to generate an ILP problem. One solution to this ILP problem provides the location of memory transfer operations on the considered flowgraph. The formulation is later extended to handle stack data (Section 3.2.3). Finally, we describe an iterative algorithm to tackle instability of worst-case execution path of a program. This algorithm incrementally generates the ILP problem for a better WCET optimization.

3.1. Flowgraphs and computation of worst-case execution path information

We have multiple choices for placement of memory transfers operations (e.g. on functions entry-exit, on basic blocks boundaries, etc). We propose to introduce a *generic* graph representation of the program flow. The chosen representation level of the generated program *flowgraph* may lead to different placements of memory transfer operations.

In Figures 2 and 3, the (right-side) flowgraph is generated from the (left-side) original graph representation. The edges in the generated flowgraph are required to describe any possible flows of execution of the program.

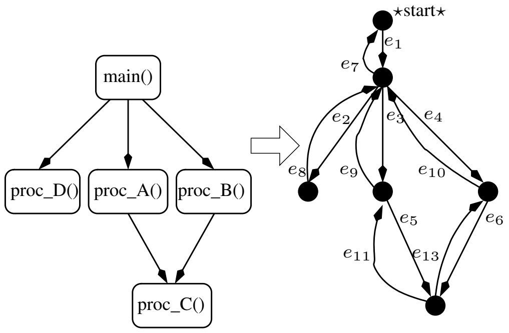
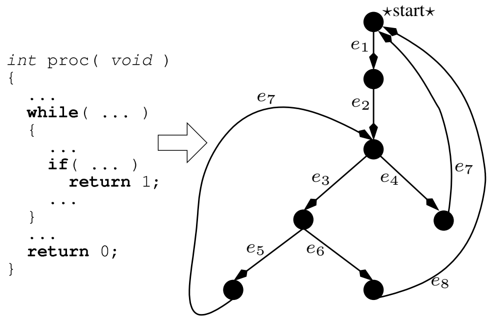


Figure 2. Call graph transformation to a (coarse-grain) flowgraph.

For example, one can build a flowgraph from the original call graph of an application (see Figure 2). There is one node in the flowgraph for each function in the call graph. We can also build a flowgraph from the interprocedural control flow graph of the application (see Figure 3). Here, there is one node in the flowgraph for each basic block in the interprocedural control flow graph. Other levels of representation are possible; one may balance between coarseness and size of the resulting flowgraph. The size of the flowgraph has a practical incidence on the complexity of the future memory allocation problem as shown later in experimental results in Section 4.2.

Previous approaches for dynamic scratchpad allocation [15, 26, 27] have focused on the optimization of the average case. Data accesses statistics are typically computed from the execution profiling with a train input. In order to reduce the WCET of a real-time application, we rely on information of data memory accesses on the worst-case execution path of the program using WCET analysis. Consequently, we apply static timing analysis as an initial step to determine the information as proposed in [21, 25]. Heptane produces information on frequencies of execution of individual basic blocks on the worst-case execution path. Since we are able to compute the set of targets for each load-store instruction (see Section 2.2), we can determine the impact



```
int proc( void )
{
  ...
  while( ... )
  {
    ...
    if( ... )
      return 1;
    ...
  }
  ...
  return 0;
}
```

Figure 3. Control flow graph transformation to a (fine-grain) flowgraph.

of any data for each basic blocks of the worst-case execution path. Moreover, we are able determine if this data is MOD (modified) or USE (used) on the execution of this basic block. The outgoing edges of the generated flowgraph associated with these basic blocks are annotated with these information.

More formally, the flowgraph is a directed graph with the following definitions:

N = Number of nodes in flowgraph;
E = Number of edges in flowgraph;
e_j = jth edge of flowgraph, $j \in [1, E]$;
$C_{e_j}(v)$ = Estimated contribution to WCET reduction for data v scratchpad-allocated on edge e_j;
$U_{e_j}(v)$ = Type of usage of data v on edge e_j where $U_{e_j}(v) \in \{MOD, USE\}$.

Some real-time applications are designed to be activated from multiple entry points. We have added the ⋆start⋆ node to the flowgraph to represent these flows of execution. Some edges are added to link the ⋆start⋆ node with any possible program entry. ⋆start⋆ node is artificially acting as a single entry point for the program. In the same way, all program exits are linked to this ⋆start⋆ node.

3.2. Formulation for static data

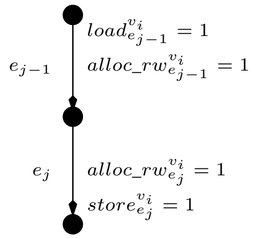
We are considering an initial problem formulation to allocate static data only with the following definitions:

M = Size of scratchpad memory;
G = Number of static data in application;
v_i = ith static data, $i \in [1, G]$;
$S(v_i)$ = Size of variable v_i in bytes;
$X_{copy}(v_i)$ = Time to transfer variable v_i between main memory and scratchpad in cycles;

The optimization problem is formulated as a 0/1 integer linear programming problem. We define the following set of binary variables, $\forall i \in [1, G], \forall j \in [1, E]$:

$$load_{e_j}^{v_i} = \begin{cases} 1 & \text{if data } v_i \text{ is transferred to scratchpad memory at the \textit{beginning} of edge } e_j, \\ 0 & \text{otherwise.} \end{cases}$$

$$store_{e_j}^{v_i} = \begin{cases} 1 & \text{if data } v_i \text{ is transferred back to main memory at the \textit{end} of edge } e_j, \\ 0 & \text{otherwise.} \end{cases}$$



e_{j-1}
$load_{e_{j-1}}^{v_i} = 1$
$alloc\_rw_{e_{j-1}}^{v_i} = 1$
e_j
$alloc\_rw_{e_j}^{v_i} = 1$
$store_{e_j}^{v_i} = 1$

Figure 4.

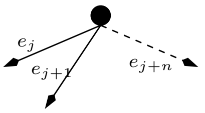


Figure 5.

$$alloc\_rw_{e_j}^{v_i} = \begin{cases} 1 & \text{if \textit{mutable} data } v_i \text{ is allocated on scratchpad memory on edge } e_j, \\ 0 & \text{otherwise.} \end{cases}$$

$$alloc\_ro_{e_j}^{v_i} = \begin{cases} 1 & \text{if read-only data } v_i \text{ is allocated on scratchpad memory on edge } e_j, \\ 0 & \text{otherwise.} \end{cases}$$

Variables $load_{e_j}^{v_i}$ and $store_{e_j}^{v_i}$ determine where data v_i are to be respectively *loaded* and *stored* on scratchpad memory. Variables $alloc\_rw_{e_j}^{v_i}$/$alloc\_ro_{e_j}^{v_i}$ 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.

$$\sum_{i=1}^{G}\sum_{j=1}^{E}\Big(alloc\_rw_{e_j}^{v_i} \times C_{e_j}(v_i) + alloc\_ro_{e_j}^{v_i} \times C_{e_j}(v_i) - load_{e_j}^{v_i} \times X_{copy}(v_i) - store_{e_j}^{v_i} \times X_{copy}(v_i)\Big)$$

Preliminary constraints have to be added to prevent inconsistencies on binary variables. Data v_i is allocated on scratchpad memory with $alloc\_rw_{e_j}^{v_i}$ or $alloc\_ro_{e_j}^{v_i}$ exclusively. $\forall i \in [1, G], \forall j \in [1, E]$:

$$alloc\_rw_{e_j}^{v_i} + alloc\_ro_{e_j}^{v_i} \leq 1 \tag{1}$$

The MOD and USE annotations of the edges of the flowgraph have a direct incidence on the problem formulation. We have to unset $alloc\_ro$ variables for edges that may update this data:

$$alloc\_ro_{e_j}^{v_i} = 0 \quad \text{if } U_{e_j}(v_i) = MOD; \tag{2}$$

3.2.1. Flow constraints

Figure 4 illustrates the need and objective of flow constraints. Let us consider data v_i allocated on scratchpad memory on adjacent and connected edges e_{j-1} and e_j. On this example, this data is loaded on the execution of e_{j-1} and stored back in main memory on the end of e_j's execution. $\forall i \in [1, G], \forall (j-1, j) \in ([1, E], [1, E])$, where e_{j-1} is an incoming edge of e_j:

$$alloc\_rw_{e_j}^{v_i} - alloc\_rw_{e_{j-1}}^{v_i} - alloc\_ro_{e_{j-1}}^{v_i} - load_{e_j}^{v_i} = 0 \tag{3}$$

$$alloc\_rw_{e_{j-1}}^{v_i} - alloc\_rw_{e_j}^{v_i} - store_{e_{j-1}}^{v_i} = 0 \tag{4}$$

$$alloc\_ro_{e_j}^{v_i} - alloc\_ro_{e_{j-1}}^{v_i} - load_{e_j}^{v_i} = 0 \tag{5}$$

Constraint 3 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 4 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 5 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}.

Figure 5 illustrates a node with multiples *outgoing* edges. Constraint 6 guarantees consistent values for variables of outgoing edges of a node in the flowgraph. $\forall i \in [1, G], \forall (j', j'') \in ([1, E], [1, E])$ where edges $e_{j'}$ and $e_{j''}$ are outgoing edges of the same node:

$$alloc\_rw^{v_i}_{e_{j'}} + alloc\_ro^{v_i}_{e_{j'}} - alloc\_rw^{v_i}_{e_{j''}} - alloc\_ro_{e^{v_i}_{j''}} = 0 \qquad (6)$$

Finally, Constraint 7 specifies the upper bound on the sum of the size of all allocated data on each edge, $\forall i \in [1, G], \forall j \in [1, E]$:

$$\sum_{i=1}^{M} \left(alloc\_rw^{v_i}_{e_j} \times S(v_i) + alloc\_ro^{v_i}_{e_j} \times S(v_i) \right) \leq M \qquad (7)$$

3.2.2. Optional support of dynamically scheduled architectures

WCET analysis requires the complete knowledge of instructions executions times. In dynamically scheduled architectures [17], pipeline modeling should take into account all possible timings for each varying timing instruction, increasing the complexity of the WCET analysis [16]. For example, load-store instructions may have multiple executions latencies if possible data targets are stored in different memories with heterogeneous latencies.

In order to reduce the complexity of WCET analysis, we would like to guarantee unique timing for each load-store instruction. Therefore, we have to express allocation of *any* targets data of this load-store instruction to the same level of the memory hierarchy (here, the scratchpad memory or the main memory).

Constraint 8 enforces removal of timing anomalies due to data memory accesses. $\forall j \in [1, E], \forall (i_1, i_2) \in ([1, G], [1, G])$ where v_{i_1} and v_{i_2} are possibly accessed on e_j by the same load-store instruction:

$$alloc\_rw^{v_{i_1}}_{e_j} + alloc\_ro^{v_{i_1}}_{e_j} - alloc\_rw^{v_{i_2}}_{e_j} - alloc\_ro^{v_{i_2}}_{e_j} = 0 \qquad (8)$$

The impact of Constraint 8 may directly depend on the number of possible targets of the pointers in programs. However, the StrongARM-110 [22] is a statically scheduled architecture and does not enable an evaluation of this constraint in the experiments of this paper.

3.2.3. Memory data address assignment

Our formulation provides an *optimistic* solution to data allocation (variables $alloc\_rw$ or $alloc\_rw$) on the edges of the flowgraph. Optimistic in the sense not all data selected by the ILP problem resolution necessary fit on scratchpad memory due to fragmentation.

An address assignment algorithm has been proposed in [27] to place data on scratchpad memory at compile-time. If no free place is found for one data, this data is simply left in main memory. In their approach, each data can be transferred multiple times between main memory and scratchpad memory; however, each data must have only *one* address in the scratchpad memory for the whole program execution.

We propose an improvement to the address assignment algorithm of [27] with the detection of individual data region. Our proposal, detailed in Algorithm 1, may decrease placement conflicts of data on scratchpad memory. A data region is defined by a subgraph of *connected* edges of the flowgraph where data v_i is scratchpad-allocated. Algorithm 1 enables the assignment of a *different* address on scratchpad memory *for each* data region.

Algorithm 1 Address assignment algorithm with detection of data regions

```
1: $data\_regions \leftarrow$ extract data regions from computed allocation
2: sort $data\_regions$ list on their impacting order on WCET
3: for all individual region $(data, edge\_set)$ from $data\_regions$ list do
4:     if $data$ fits in free memory space on edges of $edge\_set$ then
5:         select free placement for $data$ on edges of $edge\_set$ with first-fit policy
6:     else
7:         remove the unallocatable data region
8:     end if
9: end for
10: return $data\_regions$
```

First, Algorithm 1 reads the optimistic allocation computed from the ILP problem, a list of data regions is generated (line 1). This step requires an analysis of the connected components of the flowgraph for each allocated data. This list is then sorted from the impact on the program WCET of individual data region (line 2). We must try to assign a concrete address to each data region. For all edges covered by a data region, find a valid slot to assign the data (lines 3-9). If a data region can not be loaded on scratchpad memory, we ignore this data region and we simply remove all transfer operations for this data in this region.

3.3. Extension for stack data

We are now considering an extension to our previous formulation to support the limited lifetime of stack data. These data do not require initialization nor content backup to the main memory at the end of their lifetime. Similarly to static data, the flowgraph is annotated with MOD or USE for any usage of stack data. The DEF attribute is now defined on function entry and on function exit for stack data associated with the function life span. The $DEAD$ is an attribute set on flowgraph edges to avoid memory transfers for non-live stack data.

F = Number of stack data in the application;

f_i = ith stack data, $i \in [1, F]$;

$U'_{e_j}(f_i)$ = Type of usage of variable f_i on edge e_j where $U'_{e_j}(f_i) \in \{DEF, MOD, USE, DEAD\}$;

$S(f_i)$, $C_{e_j}(f_i)$, $X_{copy}(f_i)$ and the variables $alloc\_rw$, $alloc\_ro$, $load$ and $store$ are similarly defined for stack data.

The general flow Constraints 3, 4, 5 and 6 for static data are directly applicable to stack data.

The objective function to maximize is the contribution for WCET reduction of all accesses to the static data and to the stack data in the application minus all the dynamic transfers of data between main memory and scratchpad memory:

$$\sum_{i=1}^{G}\sum_{j=1}^{E}\Big(alloc\_rw_{e_j}^{v_i} \times C_{e_j}(v_i) + alloc\_ro_{e_j}^{v_i} \times C_{e_j}(v_i)$$
$$- load_{e_j}^{v_i} \times X_{copy}(v_i) - store_{e_j}^{v_i} \times X_{copy}(v_i)\Big)$$
$$+ \sum_{i'=1}^{F}\sum_{j=1}^{E}\Big(alloc\_rw_{e_j}^{f_{i'}} \times C_{e_j}(f_{i'}) + alloc\_ro_{e_j}^{f_{i'}} \times C_{e_j}(f_{i'})$$
$$- load_{e_j}^{f_{i'}} \times X_{copy}(f_{i'}) - store_{e_j}^{f_{i'}} \times X_{copy}(f_{i'})\Big)$$

Stack data are created and destroyed on function entry and on function exit (where $U'_{e_j}(f_i) = DEF$). Consequently these data don't require memory transfer operations (Constraints 9 and 10). On such edges, stack data are initialized with default values and Constraint 11 forbids read-only allocation. Moreover, we enforce (Constraint 12) 0-cost memory transfer operations to scratchpad before and after the stack data lifetime, $\forall i \in [1, F], \forall j \in [1, E]$:

$$load_{e_j}^{f_i} = 0 \quad \text{if } U'_{e_j}(f_i) = DEF \tag{9}$$
$$store_{e_j}^{f_i} = 0 \quad \text{if } U'_{e_j}(f_i) = DEF \tag{10}$$
$$alloc\_ro_{e_j}^{f_i} = 0 \quad \text{if } U'_{e_j}(f_i) = DEF \tag{11}$$
$$X_{copy}(f_i) = 0 \quad \text{if } U'_{e_j}(f_i) = DEAD \tag{12}$$

As described in [2], stack data may have disjoint lifetimes. The program *call graph* is analyzed to provide information on lifetime of stack data:

$\mathcal{L}$ = The set of all leaf nodes in the call graph;

$NP(l)$ = Total number of unique paths to the lth leaf node in the call graph, $l \in [1, \mathcal{L}]$;

$P_t(l)$ = The set of stack data definitions on the tth unique path to lth leaf node in the call graph, $t \in [1, NP(l)]$.

$P_t(l)$ set computes any possible stack data combinations that are simultaneously alive. The size constraint should be formulated as, $\forall j \in [1, E], \forall l \in \mathcal{L}, \forall t \in [1, NP(l)]$:

$$\sum_{i=1}^{G}\Big(alloc\_rw_{e_j}^{v_i} \times S(v_i) + alloc\_ro_{e_j}^{v_i} \times S(v_i)\Big)$$
$$+ \sum_{\forall f'_i \in P_t(l)}\Big(alloc\_rw_{e_j}^{f_{i'}} \times S(v_i) + alloc\_ro_{e_j}^{f_{i'}} \times S(f_{i'})\Big) \leq M \tag{13}$$

An additional extension would be to support heap allocated data as proposed in [8]. DEF machinery is an ideal attribute to define a limited-lifetime data and may be the basis for such an extension of our formulation to heap data. Currently, real-time programs benchmarks rarely employ dynamic heap allocation and won't enable us to lead a complete study on dynamic allocated data.

3.4. Support for instability of the worst-case execution path

For many programs, the worst-case execution path of the program may change after some data allocations. Consequently, it may be needed to evaluate all possible combinations of data allocations to find the optimal reduction of WCET of the program. However, an exhaustive evaluation of all possible combinations would be too time-consuming [25]. A greedy heuristic has been proposed for WCET-centric scratchpad memory static allocation that greatly enhances the quality of allocation in [25]. The method outline is to iteratively allocate one data on scratchpad memory and to (re)-estimate data frequencies information at every iterations.

We propose to adapt their approach to the ILP problem-based scratchpad memory dynamic allocation schemes in Algorithm 2.

Algorithm 2 Iterative dynamic allocation algorithm

```
1:  $allocations \leftarrow empty$
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 additionnal Constraints (14) for $allocations$
8:      $new\_allocations \leftarrow$ call solver on ILP problem
9:      if $new\_allocations \neq empty$ then
10:         $change \leftarrow true$
11:         $allocations \leftarrow allocations \cup$ most impacting allocation
            from $new\_allocations$
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 14 is added to the problem formulation (line 7) to enforce allocation of selected data. $\forall i \in [1, G], \forall j \in [1, E]$ where v_i has been selected for allocation on e_j:

$$alloc\_rw_{e_j}^{v_i} + alloc\_ro_{e_j}^{v_i} = 1 \tag{14}$$

The algorithm selects and allocates the most (non-already 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.

Similarly, we have modified the ILP problem formulation of [2] to obtain an iterative *static* memory allocation algorithm. This algorithm is used in the experiments of this paper. Lines 6-7 of Algorithm 2 are modified (1) to generate the static allocation's ILP problem [2] and (2) to generate additional constraints that enforce allocation of selected data on next iterations. We won't describe in this paper these additional constraints applied to [2]'s ILP problem (line 7) due to the lack of space.

4. Results

The evaluation of the approach for dynamic scratchpad memory allocation is performed for the StrongARM-110 processor [22]. Benchmark programs are compiled using

Benchmark	Source	Lines of code	Static data size	Max. stack size	Load-store inst. ratio	Description
Adpcm	WCET B.	870	2020 bytes	116 bytes	39%	Speech coding
Engine	Powerstone	380	585 bytes	116 bytes	6%	Engine control
G721	Powerstone	1180	730 bytes	284 bytes	16%	Voice compression
Histogram	UTDSP	80	67584 bytes	20 bytes	39%	Image enhancing application
Lpc	UTDSP	450	7388 bytes	72 bytes	22%	Speech coding
Pocsag	Powerstone	1150	1060 bytes	112 bytes	22%	Communication protocol
Spectral	UTDSP	250	2032 bytes	116 bytes	44%	Speech power spectral estimation
Statemate	WCET B.	1290	227 bytes	132 bytes	60%	Car window lift control

Table 3. Informations on benchmarks programs.

a modified GCC 4.1 compiler (see Section 2.2 for the modifications applied to the compiler to compute load-store instructions targets). The compiler generates two files: the output program and an additional file for load-store instructions targets annotations. Second, the Heptane timing analyzer reads the annotation file with load-store's targets instructions to model the complete memory behavior of the program. The program binary is read. The maximum iterations for each program loops are given as annotations to enable determination of possible execution paths in the program.

This study reports results on optimized code with loop-related optimizations disabled. The Heptane timing analyser supports the pipelined execution and the instruction cache of the StrongARM-110. The latency of a word access to main memory is 11 cycles [22]. The latency for accesses to data allocated on the scratchpad memory is 1 cycle. A penalty model for scratchpad memory transfers operations are integrated to the timing analysis of Heptane. We applied a penalty latency of 12 cycles per word of data to transfer. The commercial ILP solver CPLEX 7.1[2] is configured to stop on the first valid solution found. The proposed technique is evaluated on an assorted set of benchmarks from WCET benchmarks[3], Powerstone [23] and UTDSP[4].

4.1. Scratchpad allocation results

We have undertaken a comparison of the impact of our *iterative* dynamic scheme over *non-iterative* static scratchpad memory allocation [2] on programs WCET. In this study, (fine-grained) flowgraphs are generated from the interprocedural control flow graph of benchmarks programs. This gives the maximum latitude for placement of memory transfers in programs. Figure 6 gives the improvement ratio of iterative dynamic scratchpad memory allocation over *non-iterative* static allocation (y-axis) for a range of scratchpad memory sizes (x-axis) computed by $\frac{\text{\#cycles reduction from iterative dynamic allocation}}{\text{\#cycles reduction from non-iterative static allocation}}$. Figure 6 gives in addition the improvement of iterative static allocation over non-iterative static allocation, providing insight on stability of programs worst-case execution paths.

For five out of eight benchmarks, we can remark iterative static allocation may improve non-iterative static scratchpad memory allocation up to 30%, particularly for programs with a large amount of control flow (Engine, Pocsag, Statemate).

Histogram is a typical example of the benefit of the dynamic capability of our scratchpad memory allocation method. This program contains two frequently used arrays of 1024 bytes separately used in two program phases. Static allocation succeeds to place one of these two arrays in a 1024 bytes scratchpad memory. Dynamic allocation moves these two arrays alternatively in the scratchpad memory unit for an improvement of 47% of the original performance enhancement due to a static scratchpad allocation. On a scratchpad memory larger than 2048 bytes, there is enough room to statically place the two arrays. Both schemes yield to identical WCET value.

Major benefits for dynamic scratchpad allocation are achieved for small ratios of scratchpad memory sizes over the whole program data working set. For example, dynamic scratchpad allocation is valuable for scratchpad sizes ratios lower than 10% of the working set for the programs Adpcm, Engine and G721. On these ranges, the method outperforms the static allocation from 12% to 85%. The approach is notably profitable to systems with a scratchpad memory shared among several real-time tasks.

Due to the support of stack data (typically smaller than 32 bytes), our method takes advantage of very small scratchpad memory sizes except programs Histogram, LPC and Spectral. These benchmarks have very few stack data instances (see Figure 4) or few accesses to stack data (see Figure 2).

We have conducted some preliminary evaluations of address assignment algorithms described in Section 3.2.3. In the experiments of [27], the address assignment algorithm is shown to be fairly close to the optimal address assignment. In our experiments, the algorithm with detection of data regions gives marginal performance improvements over the address assignment algorithm of [27]. The iterative allocation algorithm selects data in their performance impact order. Consequently, data with high performance impact have higher chance to get a valid address assignment. Programs Adpcm, G721 and Lpc get a relative performance increase of 3%-7% when the detection of data regions is enabled. Gains are observed for small (less than 300 bytes) scratchpad memories. The scratchpad memory usage is high for such configurations and many data are transferred on scratchpad memory multiple times.

[2] ILOG CPLEX – High-performance software for mathematical programming and optimization: `http://www.ilog.com/products/cplex/`

[3] WCET benchmarks: `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

[4] UTDSP – DSP Benchmark Suite: `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`

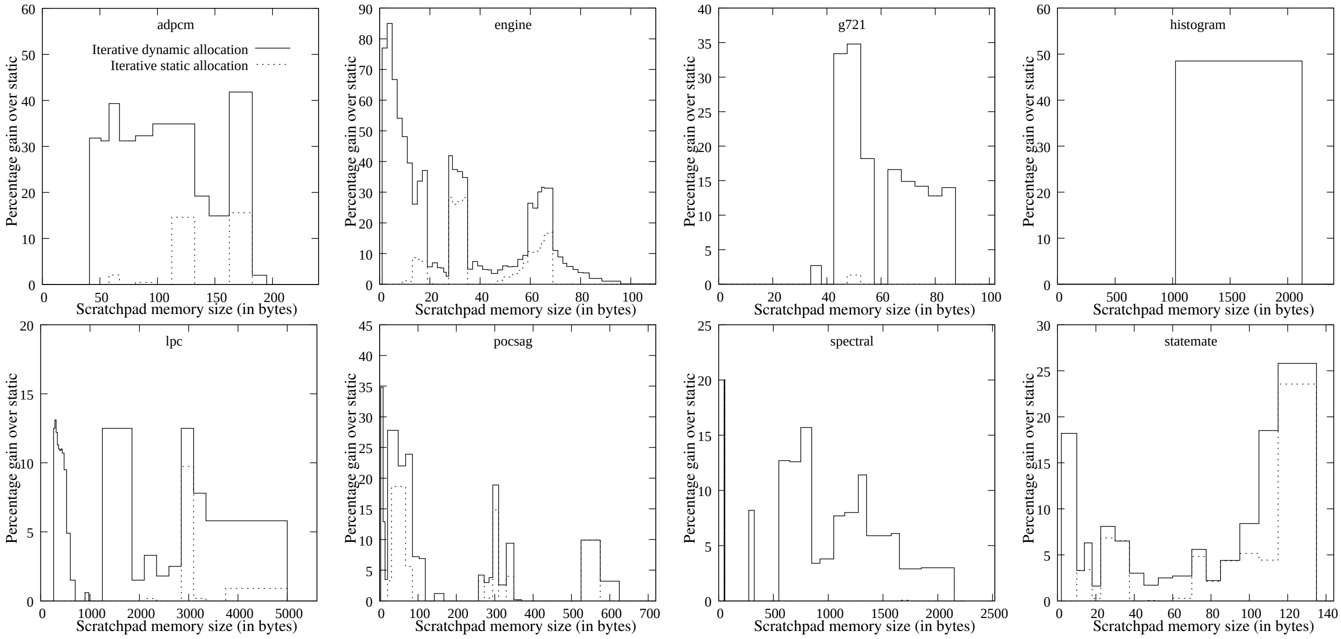


Figure 6. Improvement (in percent) of iterative dynamic and iterative static allocations over non-iterative static allocation.

4.2. Solver execution time

Allocation solving time tightly depends on the number of variables of the ILP problem. The number of variables for static allocation problem is $O(D)$ where D is the number of static data and stack data in the programs. In our experiments, there is implicitly one stack data instance for each defined function. The count of static data and functions in programs of the benchmark set are given in the Table 4. The number of variables for dynamic allocation problem is $O(D \times E)$ where E is the number of edges in the flowgraph. The number of edges depends on the representation level of the flowgraph. Table 4 delivers the number of functions and the number of basic blocks (BBs) of the programs. In this table, the number of functions of a program gives an idea of the size of the (coarse-grain) flowgraph generated from its call graph. In the same way, the number of basic blocks of a program gives an idea of the size of the (fine-grain) flowgraph generated from its interprocedural control flow graph.

In our experiments, we have observed CPLEX running time is the worst for scratchpad memory size configurations where dynamic scratchpad memory allocation gives the best improvement over static scratchpad memory allocation. Table 4 compares maximum observed running time for CPLEX solver to produce a solution for (A) static allocation problem, (B) dynamic allocation problem (coarse-grain) flowgraph (C) with (fine-grain) flowgraph.

Consequently, for the same program, B has less number of possible placements for memory transfer operations and it gives *lower quality* allocation than C. The final column of Table 4 gives the relative allocation quality reduction $\frac{B-A}{C-A}$. A value of 100% means B allocation is as efficient as C and 0% means allocation provides results as low as A static allocation. This ratio is computed for the scratchpad memory size where C does its best over A static allocation.

First of all, the running time of the ILP solver is typically not an issue for any static scratchpad memory allocation problems. Second, programs (Adpcm, G711, Pocsag, Statemate) with an important number of data and a large generated (fine-grain) flowgraph may have huge solver running time. Applying our method to much more benchmarks programs may enable us to draw general conclusions of the number of ILP variables on solver's running time. Unsurprisingly, B dynamic allocation at function granularity produces lower quality results than C dynamic allocation on basic-blocks granularity for most programs. One can remark B is as efficient as C for two of eight benchmarks (Engine, Statemate): even though their respective solving time is shorter. Conversely, Histogram contains only one function and B allocation is strictly equivalent to A static allocation.

The major conclusion of this study is two-fold. First, the practical limitation of our method is the running time to solve ILP problems, which is problematic for the largest benchmarks studied in this paper. Second, approaches exist to scale up the applicability of our method to larger programs. A coarse flowgraph induces smaller ILP problems, potentially leading to a lower allocation quality. An orthogonal approach may be to apply the method to regions of program (i.e. program subgraphs), to generate smaller ILP sub-problems. Moreover, it must be profitable to ignore some non-profitable data within a region in the generated sub-problem, reducing the number of data considered in generated sub-problems.

5. Related work

A main issue for dynamic scratchpad memory allocation is the preliminary selection of possible placements for memory transfer operations. [15] and [27] are considering placement of memory transfer operations at the level of basic

Benchmark	Data	Functions	BBs	Allocation problem solving time			Allocation quality
				(static) A	B	C	improv. ratio
Adpcm	82	11	82	$\leq$ 1s	10s	179s	59.2%
Engine	34	7	81	$\leq$ 1s	$\leq$ 1s	35s	100.0%
G721	34	19	203	$\leq$ 1s	9s	42s	14.3%
Histogram	4	1	15	$\leq$ 1s	$\leq$ 1s	$\leq$ 1s	0.0%
Lpc	19	4	85	$\leq$ 1s	$\leq$ 1s	20s	68.8%
Pocsag	26	11	126	$\leq$ 1s	2s	51s	47.0%
Spectral	10	3	48	$\leq$ 1s	$\leq$ 1s	2s	39.8%
Statemate	106	8	263	$\leq$ 1s	7s	8367s	100.0%

Table 4. Programs sizes vs. problems solving time.

blocks. [26] proposes to restrict memory transfer operations to interesting *program points*, such as functions, conditionals or loops entries/exits with high execution frequencies in a flexible way. Moreover, [26] associates execution *timestamps* to program points in order to capture program execution context. Data accesses statistics are recorded using these timestamps on a profiled execution.

The manageable granularity of the flowgraph enables flexible selection of possible places for memory transfer operations. Moreover, the support of program execution order in our flowgraph seems possible through the replication of subgraphs of the generated flowgraph. However, it is unclear how WCET analysers could generate useful data accesses information in association with execution timestamps.

Formulation for dynamic scratchpad memory allocation introduced in Section 3.2 is an adaptation of [27] to manage read-only and modified data. The main benefit is to avoid useless $store$ memory transfer operations from scratchpad to the reference copy in main memory for non-modified data.

The support of stack data described in Section 3.3 is similar to the work for static memory allocation in [2] applied to our formulation for dynamic memory allocation. [7] studies allocation of spilled data on a small and fast direct addressed memory. Our formulation for dynamic scratchpad memory allocation supports stack data and it supersedes this original work.

[2] contains an interesting study on granularity of allocation of whole stack frame (as applied in this paper's experiments) or individual stack data. Each stack data can be allocated in different memories; hence, the program must have to manage multiple program stack pointers on its execution. Their study concludes individual stack data allocations gives marginal performance increase against whole stack frame allocation due to increased cost of a multiple stack management.

[25] propose an algorithm for greedy static allocation on scratchpad memory. Their algorithm iteratively (i) evaluates the worst-case execution path of the application, (ii) selects and allocates the most impacting (non-already allocated) data to the scratchpad memory then apply (i) and (ii) until no more allocation on free memory space is possible. Our approach differs because the solver is iteratively called on a refined ILP problem and it supports allocation of stack data. Moreover, our approach is portable to both static and dynamic scratchpad memory allocation ILP problems.

In Section 3.2.3, due to scratchpad memory fragmentation, we have proposed to leave unallocatable data in main memory. Memory compaction is an interesting alternative to rearrange data on scratchpad memory. [26] has shown that such a mechanism has a minor impact on program performance. [26] also addresses major implementations issues on static data and stack data relocation for dynamic scratchpad memory allocation.

Fixed-sized scratchpad memory are unable to allocate too large data and are unable to take benefit of temporal locality on access of such data, to the difference with data caches. As studied in [15], program transformations such as tiling of big arrays enable better scratchpad memory usage and increase global effectiveness of the allocation.

6. Conclusion and future work

The main contributions of this paper are two-fold. First, we have described an approach to calculate targets of load-store instructions. Our approach is based on a common compiler infrastructure and relies on the presence of an interprocedural pointer analysis. Exhaustive knowledge of load-store instructions targets in a program requires a whole-program analysis mode, available in our compiler infrastructure.

Second, we have proposed a dynamic scratchpad memory allocation algorithm to support both static data and stack data. Our approach attempts to reduce WCET of real-time programs with the allocation of most impacting data on their worst-case execution paths. Due to the variability of the worst-case execution paths in programs [25], we have applied an iterative scheme for data allocation. This scheme requires multiple iterations of WCET program analysis and it has demonstrated improved results [25].

Our experiments have shown the increasing computational complexity of allocation problem solving with program size. To tackle this issue, we have proposed to limit data transfers to entry and exit of functions, reducing allocation problem size and leading to an absolute decrease of allocation quality.

Scratchpad memory allocation of data provides a fully predictable latency of load-store instructions [1]. Furthermore, some compiler optimizations (e.g. instruction scheduling) could make profit of such information for better code generation.

[28] have compared static scratchpad memory allocation with some instruction cache WCET analyzes. We plan to compare our approach for dynamic scratchpad memory allocation with data cache analyses.

In this paper, we have considered a system with only *one* scratchpad memory device. The optimal static memory allocation [2] supports multiple scratchpad memory devices. This may increase drastically the number of variables of the generated allocation problem. We leave such an extension for dynamic memory allocation as future work.

Acknowledgments. The authors thank Olivier Rochecouste for its comments that helped improve the quality of this paper.

References

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, Austin, TX, Dec. 1993.

[2] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, Nov. 2002.

[3] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23, Barcelona, Spain, Mar. 2004.

[4] D. Berlin. Structure aliasing in GCC. In *Proceedings of the 2005 GCC Developer's Summit*, pages 25–35, Ottawa, Canada, June 2005.

[5] H. Cassé, L. Féraud, C. Rochange, and P. Sainrat. Using Abstract Interpretation Techniques for Static Pointer Analysis . *Computer Architecture News*, 27(1):47–50, Mar. 1999.

[6] A. Colin and I. Puaut. A modular & retargetable framework for tree-based WCET analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[7] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, CA, Oct. 1998.

[8] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4):521–540, July 2005.

[9] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the 1st International Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, Tahoe City, CA, Oct. 2001.

[10] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420, New Haven, CT, Aug. 1992.

[11] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT 2001 Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, Snowbird, UT, June 2001.

[12] R. E. Johnson, C. McConnell, and J. M. Lake. The RTL system: A framework for code optimization. In *Proceedings of the International Workshop on Code Generation*, pages 255–274, Dagstuhl, Germany, May 1991.

[13] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 230–240, Brookline, MA, June 1996.

[14] R. Kirner and P. P. Puschner. Classification of WCET analysis techniques. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 190–199, Seattle, WA, May 2005.

[15] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 329–338, St. Louis, MO, Sept. 2005.

[16] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, Nov. 2006.

[17] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, Phoenix, AZ, Dec. 1999.

[18] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 255–262, Hong Kong, China, Dec. 1999.

[19] J. W. D. Manuel E. Benitez. A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, GA, June 1988.

[20] S. Mehrotra and L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 133–140, Philadelphia, PA, May 1996.

[21] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems, a quantitative comparison. In *Proceedings of the 2007 Conference on Design Automation and Test Europe*, pages 1484–1489, Nice, France, Apr. 2007.

[22] *SA-110 microprocessor timing: an application note*. Digital Equipment Corporation, June 1997.

[23] J. Scott, L. H. Lee, J. Arends, and B. Moyer. Designing the low-power mcore architecture. In *Proceedings of the Workshop on Power Driven Microarchitecture*, pages 145–150, Barcelona, Spain, June 1998.

[24] J. Staschulat and R. Ernst. Worst case timing analysis of input dependent data cache behavior. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 227–236, Dresden, Germany, July 2006.

[25] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 223–232, Miami, FL, Dec. 2005.

[26] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems*, 5(2):472–511, May 2006.

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

[28] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of 2005 Design, Automation and Test in Europe Conference and Exposition*, pages 600–605, Munich, Germany, Mar. 2005.

[29] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2-3):209–233, Nov. 1999.

[30] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, Dec. 1994.