

WCET Centric Data Allocation to Scratchpad Memory

Vivy Suhendra Tulika Mitra Abhik Roychoudhury Ting Chen
School of Computing, National University of Singapore
{viv,y,tulika,abhik,chent}@comp.nus.edu.sg

Abstract

Scratchpad memory is a popular choice for on-chip storage in real-time embedded systems. The allocation of code/data to scratchpad memory is performed at compile time leading to predictable memory access latencies. Current scratchpad memory allocation techniques improve the average-case execution time of tasks. For hard real-time systems, on the other hand, worst case execution time (WCET) is a key metric. In this paper, we propose scratchpad allocation techniques for data memory that aim to minimize a task's WCET. We first develop an integer linear programming (ILP) based solution which constructs the optimal allocation assuming that all program paths are feasible. Next, we employ branch-and-bound search to more accurately construct the optimal allocation by exploiting infeasible path information. However, the branch-and-bound search is too time-consuming in practice. Therefore, we design fast heuristic searches that achieve near-optimal allocations for all our benchmarks.

1. Introduction

The increasing performance gap between the processor and the off-chip memory has made it essential to include some form of on-chip memory in real-time embedded systems. Traditionally, caches have been used extensively as on-chip memory in high-performance computing systems. The advantage of cache is that the allocation and deallocation of memory blocks from the cache are managed dynamically by hardware. So, caches are completely transparent to the programmer and/or compiler. Unfortunately, this transparency leads to unpredictable timing behavior for real-time software. In real-time systems (especially safety-critical ones), the designer must provide a guaranteed upper bound on the worst case execution time (WCET) of the software. This upper bound is estimated through static program analysis called WCET analysis. The presence of caches (particularly data caches) in the processor adds significant complications to the WCET analysis framework.

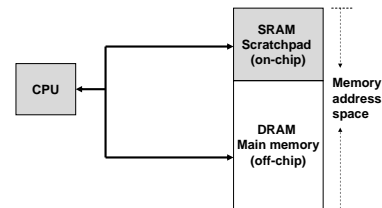


Figure 1. Scratchpad memory.

An alternative to caches for on-chip storage is scratchpad memory, which is inherently more predictable. Scratchpad memories are small on-chip memories that are mapped into the address space of the processor. Whenever the address of a memory access falls within a pre-defined address range, the scratchpad memory is accessed (see Figure 1). As the memory access latencies are predictable, scratchpad memories have become popular for real-time embedded systems. The other advantages of scratchpad memory include reduced area and energy consumption compared to caches [4]. However, now the burden of allocating code/data to scratchpad memory lies with the compiler. In this paper, we concentrate on the allocation of data objects to scratchpad memory with the goal of reducing the WCET of a task. Note that there is no data cache in our memory hierarchy. We consider data objects for allocation as they are more difficult to handle as far as timing predictability of real-time tasks is concerned. Our allocation technique can also be applied to code objects with minimal modification.

Significant research effort has been invested in developing efficient allocation techniques for scratchpad memory [3, 12, 13]. However, all these techniques 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 main difficulty in developing optimal scratchpad memory allocation technique for WCET is the following. An ACET-guided allocation method uses the access frequencies of variables obtained through profiling. In WCET-guided allocation, we are interested in the ac-

cess frequencies of the variables along the worst-case path. As we allocate variables along the worst-case path into the scratchpad memory, a new path may become the worst-case path. This leads to a different access frequency profile of the variables corresponding to the new worst-case path. As a result, locally optimizing the current worst-case path may not lead to the globally optimal solution. The elegant techniques used in ACET-guided optimal allocations, such as 0-1 knapsack are not applicable for WCET-guided allocation.

In this paper, we propose customized optimal and near-optimal allocation techniques that are guided by WCET. Our first optimal allocation technique is based on a simple Integer Linear Programming (ILP) formulation. This solution does not take infeasible path information into account. Therefore, it can potentially allocate objects from a heavy (w.r.t. execution time) but infeasible path. To overcome this drawback, we propose another optimal allocation technique based on branch-and-bound search that does exploit infeasible path information. But branch-and-bound search may be inefficient for large number of variables and large scratchpad sizes. So we also design a greedy heuristic algorithm that is efficient, considers infeasibility information, and produces near-optimal solutions. As we allocate variables using branch-and-bound search or greedy heuristic, the current worst-case path may shift to a new one. Therefore, our allocation techniques require repeated search for the worst-case path and the access frequencies of the variables along that path. In order for our allocation techniques to be efficient, this search should be quite fast. We develop a fast and accurate method for finding the WCET path.

In summary, the main contribution of this work is in developing efficient techniques for allocating program variables to scratchpad memory explicitly guided by the goal of reducing the program's WCET.

2. Related Work

In this section, we review existing literature on ACET-based allocation of code/data to scratchpad memory for embedded systems as well as WCET-based compiler optimizations in different contexts.

Panda et al. have developed a comprehensive allocation strategy for scratchpad memory [12, 13] to improve the average-case program performance. The architecture they use is somewhat different from ours. They assume the presence of data cache on top of the scratchpad memory. Therefore, the goal of their allocation strategy is to minimize the conflict among the variables in the data cache. Avissar et al. [3], on the other hand, assume an architecture that is very similar to ours, i.e., they do not assume the presence of caches. They propose a 0-1 ILP solution to optimally allocate global and stack variables. Their more recent work

extends this approach to heap memory [8]. Sjodin et al. [16] also propose a 0-1 ILP solution for reducing code size via scratchpad allocation. This is achieved by allocating variables to appropriate memory types so that the size of the corresponding pointers and hence the code size can be reduced. Similarly, Steinke et al. [17] formulate an ILP-based allocation strategy to reduce the overall energy consumption of the program. *All these works use profile-guided optimization as their goal is to reduce the average-case execution time (ACET)*. There is a separate body of work [10, 19] that allow the scratchpad contents to change dynamically based on data access patterns; they also use scratchpad for improving average-case program performance or energy consumption.

Compiler techniques to reduce the worst case execution time of a program have started to receive attention very recently. Lee et al. [11] have developed a code generation method for dual instruction set ARM processors to simultaneously reduce the WCET and code size. They use full ARM instruction set along the WCET path to achieve faster execution; the reduced Thumb instructions are used along the non-critical paths to reduce code size. Yu and Mitra [22] perform WCET-guided selection of application-specific instruction set extensions. Bodin and Puaut [5] design a customized static branch prediction scheme for reducing a program's WCET. This work employs a greedy heuristic to design the branch prediction scheme — all branches appearing in the current WCET path are predicted based on their outcomes in the WCET path. Zhao et al. [24] use a greedy heuristic for code positioning that places the basic blocks on WCET paths in contiguous positions whenever possible. Their most recent work [23] reduces WCET by forming superblocks along the WCET path. To the best of our knowledge, there has been no work so far on allocating variables to scratchpad memory specifically to reduce the WCET. In this paper, we perform scratchpad allocation with the explicit goal of WCET reduction.

The work of [21] is probably closest to ours. They use a static allocation strategy for program and data objects in scratchpad memory to reduce the average-case energy consumption of the program. Now using the resulting scratchpad allocation, they compute the WCET of the program. Their results show that a tight WCET bound is obtained in an architecture with scratchpad but no data caches. Using similar sized data cache, however, leads to significant overestimation in WCET, in particular for large cache sizes. As a result, the estimated WCET with scratchpad memory is much better than the estimated WCET with the same sized data cache. The main difference between this work and ours is the following. Instead of using an allocation technique that has been developed to improve average-case execution of an application, we develop *customized* allocation techniques that are guided by WCET reduction. As we al-

locate certain variables to scratchpad, the WCET path and hence the critical variables may change. Therefore, obtaining optimal allocation to reduce WCET is inherently more challenging; the problem is further complicated due to the presence of infeasible program paths.

Conceptually, the works on data cache locking for predictable program execution [18] bear similarities to our work on allocating data variables to scratchpad memory. In particular, [18] combines data cache locking and static cache analysis to enhance the timing predictability of program execution. Unpredictable regions of the program are identified and for each of these regions, average-case execution profile is used to determine which portions of the data memory will be locked into the cache. One can use our WCET-guided allocation strategies to further enhance these techniques – our allocation strategy can accurately determine the locked cache contents for each unpredictable region.

3. ILP Formulation

In this section, we address the problem of allocating data variables to scratchpad memory so as to reduce the WCET of a program. We do not take into account any infeasible path information, that is, all paths in the control flow graph are considered feasible. In the next section, we consider optimal and near-optimal allocation of data variables to scratchpad memory by considering infeasible path information.

Assumptions Our WCET-guided allocation method is static, i.e., the allocation of variables is fixed at compile time. We consider both scalar variables and arrays. An array can be allocated only if the entire array fits into the scratchpad. We consider for allocation the global variables and the stack variables (parameters, local variables, and return variables) corresponding to non-recursive functions. We do not consider stack variables corresponding to recursive functions because multiple instances of these variables may exist during program execution. For non-recursive functions, our method treats the stack variables just like global variables, i.e., these stack variables are allocated for the entire execution of the program. This restriction can be relaxed in a manner similar to [3] by taking into account functions with disjoint lifetimes.

ILP Formulation We now present our allocation method based on Integer Linear Programming (ILP). We use names starting with capital letters for ILP variables and names starting with small letters for constants to clearly highlight the ILP variables in the constraints. First we develop a scheme for allocating data variables appearing in a single

program loop; later we extend the technique to general programs. Let us consider the directed acyclic graph (DAG) capturing the control flow in the loop body (i.e., the control flow graph of the loop body without the loop back-edge). We assume that the DAG has a unique source node and a unique sink node. If there is no unique sink node, then we add a dummy sink node. Each path from the source to the sink in the DAG is an *acyclic path* — a possible path in a loop iteration. For each data variable v in the program we define a 0 – 1 decision variable S_v which indicates whether v is selected for scratchpad allocation. Thus

$$S_v \geq 0 \quad S_v \leq 1$$

$$\sum_{v \in allvars} S_v * area_v \leq scratchpad_size$$

where $allvars$ is the set of program variables, $area_v$ is a constant denoting the scratchpad area to be occupied by v if it is allocated and $scratchpad_size$ is a constant denoting the total size of the scratchpad memory available. We consider the DAG representing the loop body’s control flow and define a variable W_i for each basic block i in the DAG. Variable W_i denotes the cost of the worst-case path in the DAG rooted at basic block i under the allocation captured by the S_v variables. For each outgoing edge $i \rightarrow j$ from basic block i in the DAG, we have the following constraint.

$$W_i \geq W_j + (cost_i - \sum_{v \in vars(i)} S_v * gain_v * n_{v,i})$$

where $cost_i$ is a constant denoting the execution time (in terms of cycles) of basic block i without any allocation. Furthermore, $vars(i)$ denotes the set of program variables appearing in basic block i , $gain_v$ is a constant denoting the gain (in number of cycles) of a single access of v by allocating v to scratchpad memory, and $n_{v,i}$ is the number of occurrences of v in basic block i . The sink node of the DAG has no outgoing edges. For the sink node we define W_{sink} as follows.

$$W_{sink} = cost_{sink} - \sum_{v \in vars(sink)} S_v * gain_v * n_{v,sink}$$

Clearly, the variable W_{src} (for the source node of the DAG) captures the worst-case acyclic path under the allocation given by S_v variables. So, we define the objective function as $W_{src} * lb$ where lb is a known constant denoting the maximum number of loop iterations. The ILP solver finds the assignment of S_v variables (i.e., the scratchpad allocation) which minimizes the loop’s worst-case execution time.

Extension to full programs In the preceding, we determine the optimal scratchpad allocation based on a single

program loop. To extend our formulation to whole programs, we need to generate the constraints for each innermost program loop as mentioned above. Next we transform the program’s control flow graph by converting each innermost loop to a “basic block”; the cost of each innermost loop is given by the “objective function” mentioned above. We can now construct the constraints for loops in the next level of loop nesting. We go on in this fashion until we have reached the topmost level of loop nesting; this gives us all the ILP constraints. The new objective function to be minimized is now W_{entry} where *entry* is the only entry node in the program’s control flow graph.

4. Allocation via Customized Search

In this section, we present search algorithms for generating optimal and near-optimal allocations by taking into account infeasible path information. We have incorporated infeasible path detection into our WCET analysis technique, which we describe in Section 4.3. Here we show how the problem of variable allocation to minimize WCET can be formulated, where the WCET analysis takes into account infeasible path information.

Given the size of the scratchpad memory *scratchpad_size*, we define an allocation as a set $V \in 2^{allvars}$ s.t. $\sum_{v \in V} area_v \leq scratchpad_size$, where the set $2^{allvars}$ denotes the power-set of *allvars*. Let $WCET_V$ be the WCET after allocating the set of variables V into the scratchpad memory. We want to choose the “optimal” allocation, that is, the allocation $V \in 2^{allvars}$ that produces the minimum $WCET_V$. As before, *allvars* denotes the set of all variables accessed in the program, and *area_v* denotes the size of variable v .

Finding the optimal variable allocation for WCET reduction requires the contribution of each variable towards the WCET. However, we cannot define the contribution of a variable towards the WCET as a constant. This is because of the following reasons. First, the contribution of a variable towards the WCET is dependent on the current WCET path. However, allocation of that variable may result in a new WCET path. Therefore, the reduction in WCET due to allocation of a variable is, in general, *not* equal to the contribution of the variable towards the current WCET. Secondly, the reduction in WCET due to allocation of two or more variables is not accumulative. That is, if the WCET reduction by allocating variables v and v' are X and X' , respectively, then the WCET reduction by allocating variables v and v' together can be less than $X + X'$.

To illustrate these two points, consider two paths p and p' having execution times W and W' assuming no variable allocation to scratchpad memory. Suppose $WCET = W = W' + \epsilon$. Consider two variables v and v' where v is accessed in p but not in p' , while v' is accessed in p' but

not in p . Let $cost_v(p) = cost_{v'}(p') > \epsilon$ where $cost_v(p)$ ($cost_{v'}(p')$) is the contribution of variable v (v') towards the execution time of p (p'). Allocating v in the scratchpad memory reduces W by $cost_v(p)$. But p' now becomes the WCET path; thus, the reduced worst-case execution time is W' . The reduction in WCET is ϵ instead of $cost_v(p)$. Furthermore, allocating v and v' reduces both W and W' . As $cost_v(p) = cost_{v'}(p')$, the reduced worst-case execution time is $W - cost_v(p)$. Here, the WCET reduction does not involve contribution of v' .

As we cannot define the contribution of a variable towards the WCET as a constant, the optimal allocation problem for WCET reduction cannot be formulated as a knapsack problem. Furthermore, as the contributions of consecutively allocated variables do not accumulate, the “optimal substructure” property required for dynamic programming is absent. This rules out an optimal dynamic programming solution. We use a branch-and-bound search algorithm to obtain the optimal solution.

4.1. Branch-and-Bound Search

The general paradigm of branch-and-bound deals with optimization problems over a search space that can be presented as the leaves of a search tree. The search is guaranteed to find the optimal solution, but its complexity in the worst case is as high as that of exhaustive search. In our case, the search space consists of the set of all possible allocations $V \in 2^{allvars}$.

Each level k in the branch-and-bound search tree corresponds to the decision of including or excluding a variable $v_k \in allvars$ into the solution set V . Thus, each node m at level k corresponds to a partial allocation $allocation(m)$ with the decision about the variables v_1 up to v_k , i.e., $allocation(m) \subseteq \{v_1, \dots, v_k\} \subseteq allvars$. Whenever we reach a leaf node of the search tree, we have a complete allocation. We then calculate the reduced WCET corresponding to this allocation (see Section 4.3). 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, a **heuristic function** computes an upper bound, $UB(m)$, on the maximum possible WCET reduction at any leaf node in the subtree rooted at m . If $UB(m) < B$, then the search space corresponding to the subtree rooted at m can be pruned. Clearly, the choice of the heuristic function UB is crucial in deciding the amount of search space pruning achieved.

We define the heuristic function for a node m at level k of the search tree as follows. First, we compute the WCET reduction, $reduction(m)$, corresponding to the partial allocation $allocation(m)$ at node m . The upper bound, $UB(m)$,

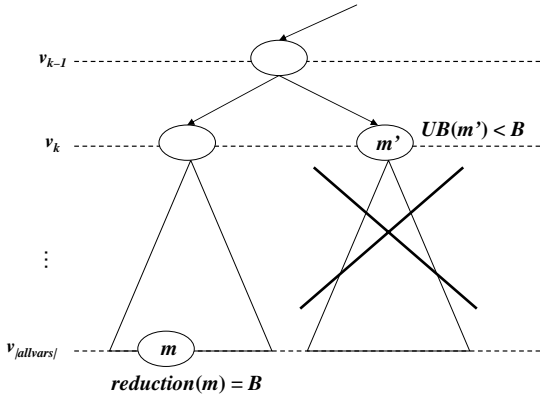


Figure 2. Pruning in the branch-and-bound search tree

is the sum of $reduction(m)$ and the maximum potential reduction in WCET due to the allocation of the variables not yet considered, i.e., $\{v_{k+1}, \dots, v_{|allvars|}\}$. An estimation of the latter is formulated as a simple knapsack problem, which is solved using dynamic programming. The inputs to the knapsack problem are as follows.

1. Variables $v_{k+1}, \dots, v_{|allvars|}$
2. Size of each variable $area_{v_{k+1}}, \dots, area_{v_{|allvars|}}$
3. Size limit defined as the remaining space in the scratchpad $scratchpad_size - \sum_{v \in allocation(m)} area_v$
4. Bound on the maximum WCET reduction due to allocation of each variable $bound_{v_{k+1}}, \dots, bound_{v_{|allvars|}}$. For a variable v , $bound_v$ is defined as the maximum contribution of v towards the execution time of any path. Clearly, the WCET reduction achieved by allocating v to scratchpad memory should be bounded by $bound_v$. These bounds can be estimated once and for all through a single traversal of the control flow graph.

The 0-1 knapsack problem allocates some of the variables $v_{k+1}, \dots, v_{|allvars|}$ to the remaining scratchpad memory space so as to maximize the potential reduction in WCET. Recall that the knapsack problem simply computes the heuristic function, which helps in pruning the branch-and-bound search space.

Figure 2 illustrates the branch-and-bound search process. Suppose that at one point of the search we have constructed a complete allocation at the leaf node m , which achieves the maximum reduction in WCET among all complete allocations encountered so far. We remember $reduction(m)$ as

the bound B — representing the maximum WCET reduction achieved so far. Suppose later in the search we reach node m' at level k . Using the heuristic function described above, we calculate $UB(m')$, an upper bound on the reduction in WCET achieved by extending the partial allocation at m' to a complete allocation. At this point suppose we find that $UB(m') < B$ which means that any complete allocation we may construct by continuing from m' will never outperform the allocation we have constructed at m . Clearly, in this situation the subtree rooted at m' need not be explored further and can be pruned from the search tree (refer Figure 2).

In order to achieve effective pruning of the unexplored nodes, the variables are sorted such that a variable that can potentially reduce the WCET more is explored higher up in the search tree. In other words, we measure the potential WCET reduction of a variable v using its maximum contribution over all execution paths, namely $bound_v$. The ordering is simply a decreasing order of $bound_v$ for $v \in allvars$.

The branch-and-bound formulation as described above yields an optimal solution for global WCET optimization. Unfortunately, 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 (where the scratchpad capacity constraint may prune out a large portion of the search tree).

Algorithm 1 Greedy heuristic for scratchpad allocation to reduce WCET of a program

- 1: $allocation := \emptyset$; $capacity := scratchpad_size$; $change := TRUE$;
 - 2: perform WCET analysis to obtain worst-case path π ;
 - 3: **while** ($capacity > 0$ AND $change = TRUE$) **do**
 - 4: $change := FALSE$;
 - 5: $V := \{v \mid v \text{ is an unallocated variable accessed in path } \pi, area_v \leq capacity\}$;
 - 6: **if** $V \neq \emptyset$ **then**
 - 7: find the variable $v \in V$ with the maximum contribution towards the execution time of π ;
 - 8: $allocation := allocation \cup \{v\}$;
 - 9: $capacity := capacity - area_v$;
 - 10: $change := TRUE$;
 - 11: perform WCET analysis to compute the new worst-case path π ;
 - 12: **end if**
 - 13: **end while**
 - 14: return $allocation$;
-

4.2. Greedy Heuristic

Since the branch-and-bound search is inefficient in running time, we also develop and use a fast heuristic search

based on greedy approaches. This search algorithm, in general, may yield a sub-optimal allocation. But in our experiments we found that the WCET reduction from the resultant allocation is close to the WCET reduction from the optimal allocation found by branch-and-bound search. In our heuristic search, we first find the heaviest path taking into account infeasible path information; let this path be π . The heuristic then allocates one program variable v appearing in π – the one with maximum contribution to the execution time of π . We then again run WCET analysis to find the heaviest path after allocating v and allocate more variables. Of course, we stop whenever the scratchpad is filled. The skeleton of the greedy heuristic appears in Algorithm 1.

We observe that the sub-optimality of the greedy heuristic arises from over-optimization of the first few heaviest paths, so that the scratchpad space is exhausted by the time we get to another relatively heavy path. We thus attempt a more complicated heuristic which balances the allocation among the competing paths by allowing backtracking in allocation. We allow backtracking when the scratchpad is filled, by removing some variables from the existing allocation to make space for variables in the current heaviest path. To guard against unbounded backtracking, we require that the new worst-case path after the replacement is not the same as any of the previously encountered WCET paths. However, this complicated heuristic does not always produce a better reduction in WCET than the greedy heuristic. It is better than the greedy heuristic when we have multiple competing paths with only a few overlapping variables, and the scratchpad size is very small (hence it gets filled up quickly). In our experiments, we found that such a situation occurs rarely. Moreover, allowing for backtracking adds an overhead to the running time of the optimization. Thus, we consider the greedy heuristic to be “better” than the complicated heuristic.

4.3. Finding WCET path

Both the branch-and-bound search as well as the greedy heuristic employ a procedure for finding the WCET path, that is, the path contributing to WCET; this procedure is typically invoked *many times* for generating the scratchpad allocation. We now discuss a simple and efficient calculation method for finding the WCET path. Since the WCET computation itself is not central to our work, we only give the general idea of the method and refer the reader to [6] for the technical issues.

Our method proceeds in two steps. In the first step, we find out certain infeasible path patterns. In general, infeasible path detection involves data-flow analysis. In our work, we avoid the expense of data flow analysis by conservatively identifying pairs of branches and/or assignments which are guaranteed to “conflict”, that is, can never lie

in an execution trace; thus our infeasible path information can be captured as a binary relation. We do not detect conflicts between arbitrary branches and assignments to avoid an inefficient conflict detection procedure. The only conditional branches appearing in our conflict relation are of the form *variable relational_operator constant*. Similarly, the only assignments which appear in our 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) but not with $x < 3$. Similarly $x > 3$ conflicts with $x < 2$ (again with no assignment to x appearing in between) but not with $x > 5$. The reader is referred to [6] for a full discussion on pair-wise conflicts (between assignments and/or branches) and their detection.

After the conflicting pairs of branches and assignments are found, the second step of the method involves WCET calculation. Here we only discuss the WCET calculation for a loop; once this is done, the WCET path of a program can be obtained by composing the WCET paths of individual loops in a manner similar to the timing schema approach [14]. Within a loop, we find the heaviest acyclic path in the loop body by traversing the loop-body’s control flow DAG from sink to source. However, to take into account the infeasible path information, we cannot afford to remember only the “heaviest path so far” at the control flow merge points. This is because the heaviest path may have conflicts with earlier branch-edges or assignment instructions resulting in costly backtracking. Instead, at a basic block i , we maintain a set of paths $paths(i)$ where each $p \in paths(i)$ is a path from block i to the sink node. In fact, $paths(i)$ contains only those paths which when extended from block i up to the source node can potentially become the WCET path. For each path $p \in paths(i)$ we also maintain a “*conflict list*”. The conflict list contains the branch-edges of p that participate in conflict with ancestor nodes and edges of block i . For any basic block i , we never maintain two paths $p, p' \in paths(i)$ where p and p' have the same conflict list; if there are two such paths we maintain the heavier among them. This greatly reduces the number of paths to be maintained and avoids exhaustive path enumeration (see [6]).

5. Experimental Evaluation

In this section, we present the experimental evaluation of our WCET-guided scratchpad allocation.

5.1. Experimental setup

We choose six data/control intensive kernels as benchmarks. The characteristics of these benchmarks are given in Table 1. *lingua* performs language-independent text

Table 1. Characteristics of the Benchmark Programs

Benchmark	Data Memory (Bytes)	Scalars (Bytes)	Arrays (Bytes)	WCET considering infeasibility (cycles)	WCET w/o considering infeasibility (cycles)
lingua	481	141	340	823,305	825,227
statemate	227	163	64	41,578	44,938
susan	36,232	96	36,136	293,989,241	485,328,185
compress	264,006	157	263,849	319,075	390,937
des	1,361	208	1,153	643,270	643,894
fresnel	536	536	0	256,172	256,172

processing [7]. `statemate` and `compress` are benchmarks taken from [20]; `statemate` is a car window lift controller generated automatically from a statechart specification, while `compress` is a data compression program. The `susan` benchmark is taken from MiBench’s automotive application suite [9]; it is a kernel performing edge thinning in an image. Finally, `des` performs Data Encryption Standard, and `fresnel` computes Fresnel integrals. Both are taken from [15].

Most of our benchmarks are compute-intensive kernels processing one or more arrays. This is evident from Table 1 that shows the total data memory size and its division between scalar and array variables. Also, most of our benchmarks have limited numbers of possible paths through any loop iteration. The only exception in this regard is `statemate`, a control-intensive application with very little data manipulation. This benchmark has a large number of possible paths (6.55×10^{16}) for one loop iteration. However, a rough estimate shows that a large number of these paths are infeasible. The number of feasible paths for any loop iteration is 1.09×10^{13} , that is, less than 0.016% of the total number of possible paths. Table 1 shows the estimated WCET of all benchmarks both with and without infeasible path information. This estimation assumes that data variables have *not yet* been allocated to scratchpad. The heaviest path in `fresnel` is feasible, so estimation with and without infeasible paths produce the same WCET.

We use the SimpleScalar tool set [2] for the experiments. The programs are compiled using `gcc 2.7.2.3` targeted for SimpleScalar. As our focus is on allocation of data variables to scratchpad memory, we assume a simple embedded processor with single-issue in-order pipeline and perfect branch prediction. Instructions are accessed from off-chip memory through a perfect instruction cache with 1 clock cycle latency. There is no data cache; a subset of data variables can be allocated to on-chip scratchpad memory. We assume that scratchpad access latency is 1 clock cycle and main memory access latency is 10 clock cycles.

We have developed a prototype analyzer that takes in a binary executable corresponding to a program and disassembles it to generate the control flow graph (CFG). The ex-

ecution time corresponding to each basic block in the CFG is then easily estimated for our simple processor architecture. For more complex micro-architectures, we can use state-of-the-art WCET estimation tools such as aiT [1]. The CFG is then analyzed at assembly level to identify conflicting branch/assignment pairs and this information is used to efficiently calculate the WCET (see Section 4.3). We assume that loop bounds required for WCET calculation are provided through manual annotation. We then run the three different scratchpad allocation techniques (Integer Linear Programming, branch-and-bound, and greedy heuristic) for each benchmark to obtain the corresponding allocation and the reduced WCET. All the experiments have been performed on a 3.0GHz P4 CPU with 1MB cache and 2GB memory.

5.2. Scratchpad Allocation Results

Figure 3 shows the original and reduced WCET due to scratchpad allocation by using ILP, branch-and-bound and greedy heuristic methods. The original WCET assumes that all variables are allocated in off-chip memory, i.e., there is no scratchpad memory. This is quite common for current real-time systems. The reduced WCET is the estimation returned by the different techniques after scratchpad allocation. Note that in Figure 3 the reduced WCET is indicated by the white bars; the difference between the original WCET and reduced WCET is indicated by the black bars stacked on top of the white bars. Thus the total height of each bar (black + white) indicates the original WCET.

We choose three different scratchpad memory sizes for each benchmark corresponding to 5%, 10%, and 20% of the total data memory size (see Table 1). The WCET reduces by 5 – 80% due to allocation for the different benchmarks. As we increase the scratchpad size from 5% to 10%, there is little or no additional reduction in WCET for `compress`, `lingua` and `susan`. This is because these benchmarks have some large arrays and increasing the scratchpad size still cannot accommodate these arrays. In general, allocating *only* 10% of the data memory to scratchpad achieves quite a significant reduction in WCET for all benchmarks.

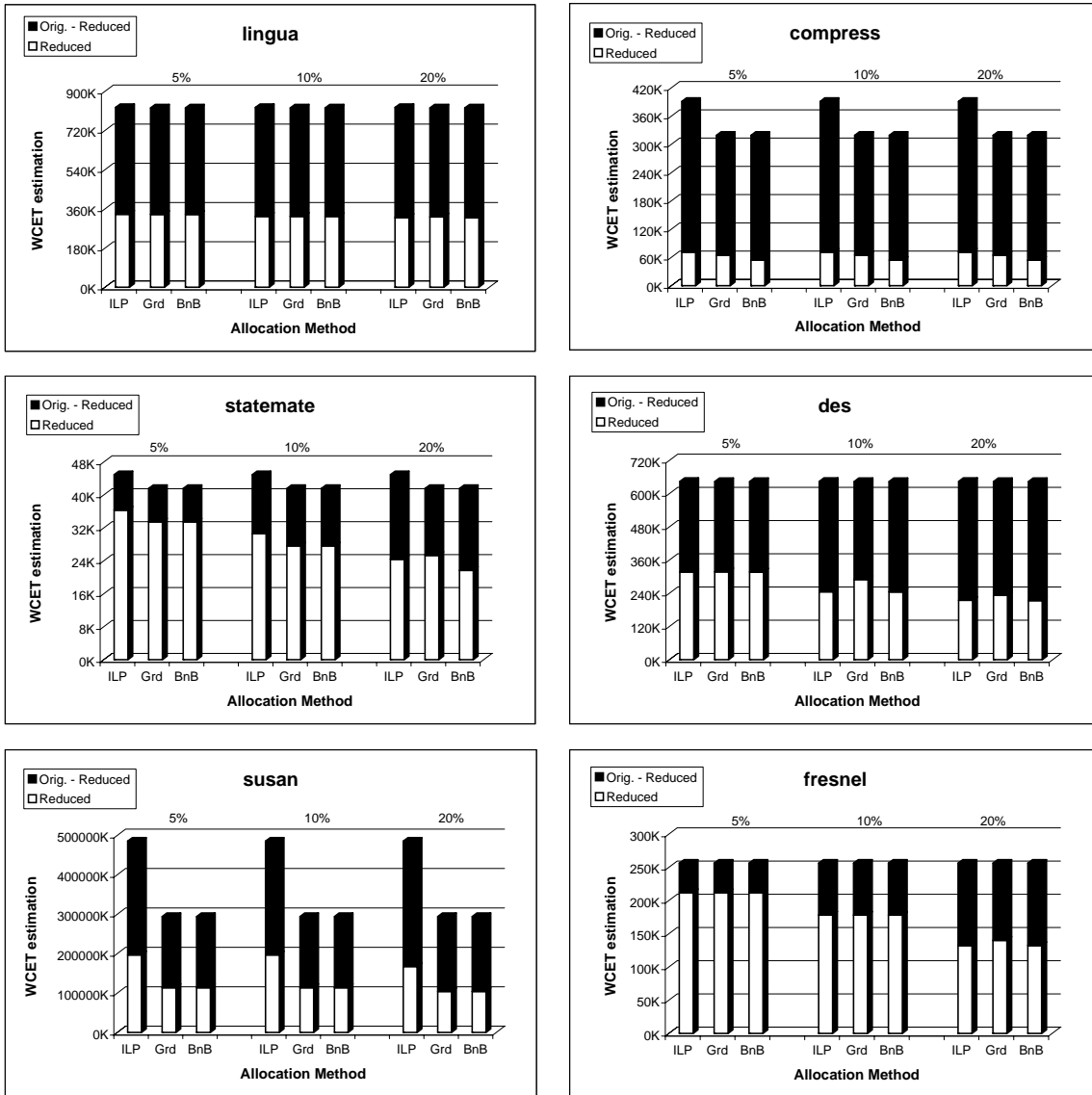


Figure 3. Original and reduced WCET (in terms of execution cycles) for various benchmarks and scratchpad sizes with ILP, greedy (Grd), branch-and-bound (BnB) scratchpad allocation techniques

Notice that the reduced WCET obtained via ILP is typically higher than the reduced WCET with branch-and-bound and/or greedy heuristics. This is because, the ILP-based method cannot take into account the detailed infeasibility information as exploited by our efficient WCET calculation method. This result shows that it is important to take the infeasibility information into account when analyzing and optimizing for WCET. In most cases there is very little or no difference between greedy heuristic and branch-and-bound implying that greedy heuristic achieves near-optimal solutions. Note that for `statemate` with scratchpad size equal to 20% of the data memory, the reduced WCET with ILP is slightly better than the reduced WCET with greedy approach. Even though the greedy approach takes infeasibility information into account, it is still sub-optimal and in this particular case, it performs worse than ILP.

Finally, as mentioned earlier, existing scratchpad allocation techniques use the average-case profile information. However, the optimal allocation for average-case may not be optimal in reducing the worst case execution time. Wehmeyer and Marwedel [21] investigate the effect of scratchpad allocation on WCET. They use average-case profile for determining the allocation. Figure 4 shows experimentally that scratchpad allocation with average-case profile may not lead to optimal reduction in WCET. The average-case data access frequencies are collected by running the benchmark with representative set of inputs. Then, we formulate a 0-1 knapsack problem to find the allocation that optimizes the average-case execution time (ACET); we then compute the reduction in WCET using this allocation. This appears as “Avg” in Figure 4. We plot it against the reduction in WCET using our WCET-guided allocation techniques – ILP, branch-and-bound, and greedy methods. In the `fresnel` benchmark, our WCET-guided allocation via branch-and-bound and greedy methods produces up to 46% reduction in WCET as compared to the allocation produced by the ACET-based technique. Other benchmarks show similar trends but less pronounced WCET reduction. For example, in the `lingua` benchmark our WCET-guided greedy allocation strategy produces up to 22% WCET reduction as compared to the allocation produced by the ACET-based technique. We also observed the effect of our WCET-guided allocation methods on ACET in all the benchmarks. We found that our ILP and branch-and-bound methods achieve similar reduction in ACET as compared to ACET-guided allocation methods.

The running time for the different allocation techniques is shown in Table 2 with scratchpad memory size equal to 10% of the data memory size. It is interesting to note that even though the reduced WCET with ILP method is typically greater than the reduced WCET with greedy heuristic, the running time of the greedy method is comparable

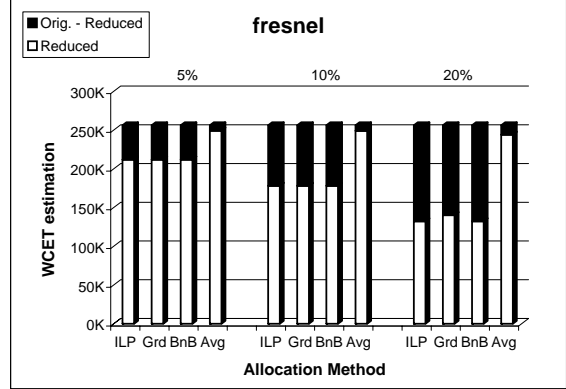


Figure 4. Original and reduced WCET (in terms of execution cycles) for different scratchpad sizes with ILP, greedy (Grd), branch-and-bound (BnB) and ACET based scratchpad allocation techniques for the `fresnel` benchmark.

Table 2. Running time of allocation methods for scratchpad = 10% of data memory

Benchmark	Runtime (ms)			
	ILP		Greedy	Branch-and-bound
	Form'n	Sol'n		
<code>lingua</code>	3	28	16	78
<code>statemate</code>	4	33	12,080	36,616
<code>susan</code>	3	15	18	23,960
<code>compress</code>	3	18	15	346,740
<code>des</code>	3	18	5	19
<code>fresnel</code>	3	16	1	6

to or even less than the running time of the ILP method for all benchmarks except `statemate`. For `statemate`, the running time of the greedy method is substantially more than that of the ILP method. In this benchmark, there are a large number of program paths and hence it is more time consuming to estimate the WCET by taking infeasibility information into account. ILP-based allocation does not take infeasibility information into account and hence is much faster.

6. Discussion

In this paper, we have presented scratchpad memory allocation techniques for data variables with the explicit goal of reducing the WCET of a program. We have proposed both optimal and heuristic allocation techniques. The major difference between our work and existing works on scratch-

pad allocation is that we specifically target WCET reduction instead of using the WCET path (which changes as we fix the allocation) or the ACET path as profiles.

In the future, we plan to include the data cache in the WCET analysis and optimization framework along with the scratchpad memory. Our framework will first identify the variables with predictable access patterns. These variables can be allocated in the main memory and accessed through the data cache as their access times are easily analyzable. The allocation techniques described in this paper can be used to determine the allocation of the remaining variables to scratchpad memory. Another possible direction of future work is WCET-guided, software-controlled, runtime management of scratchpad memory. This will allow us to allocate more variables than the capacity of the scratchpad memory through dynamic overlay.

Acknowledgments

This work was partially supported by NUS research project R252-000-171-112 and InfoComm InfoTech Initiative (ICITI) project R252-000-150-112.

References

- [1] AbsInt. aiT: Worst case execution time analyzer, 2004. <http://www.absint.com/ait/>.
- [2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.
- [3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad based embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 1(1), 2002.
- [4] R. Banakar, S. Steinke, B. S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *CODES*, 2002.
- [5] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.
- [6] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *5th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2005.
- [7] Ctrl Computer Systems. Network bookcase, 2000. <http://www.bookcase.com/library/software/msdos.devel.lang.c.html>.
- [8] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 2005.
- [9] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Annual Workshop on Workload Characterization*, 2001.
- [10] M. Kandemir et al. A compiler based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on CAD (TCAD)*, 23(2), 2004.
- [11] S. Lee et al. A flexible tradeoff between code size and WCET using a dual instruction set processor. In *SCOPES*, 2004.
- [12] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.
- [13] P. R. Panda, N. D. Dutt, and A. Nicolau. On chip vs. off chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3), 2000.
- [14] C. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-time Systems*, 5(1), 1993.
- [15] W. H. Press et al. *Numerical Recipes in C*. Cambridge University Press, 2002.
- [16] J. Sjodin and C. von Platen. Storage allocation for embedded processors. In *ACM International Conference on Compiler, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.
- [17] S. Steinke et al. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, 2002.
- [18] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2003.
- [19] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004.
- [20] WCET benchmarks. Benchmarks from C-LAB and Uppsala University, 2004. <http://www.c-lab.de/home/en/download.html>.
- [21] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, 2005.
- [22] P. Yu and T. Mitra. Satisfying real-time constraints with custom instructions. In *ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2005.
- [23] W. Zhao, W. Krehling, D. Whalley, C. Healy, and F. Mueller. Improving WCET by optimizing worst-case paths. In *Intl. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.
- [24] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET code positioning. In *International Real-Time Systems Symposium (RTSS)*, 2004.