Optimal Program Partitioning for Predictable Performance

Jack Whitham and Neil Audsley
Real-Time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK
jack@cs.york.ac.uk

***Abstract*—Scratchpad memory (SPM) provides a predictable and energy efficient way to store program instructions and data. It would be ideal for embedded real-time systems if not for the practical difficulty that most programs have to be modified in source or binary form in order to use it effectively. This modification process is called *partitioning*, and it splits a large program into sub-units called regions that are small enough to be stored in SPM.**

Earlier papers on this subject have only considered regions formed around program structures, such as loops, methods and even entire tasks. Region formation and SPM allocation are performed in two separate steps. This is an approximation that does not make best use of SPM.

In this paper, we propose a k-partitioning algorithm as a new way to solve the problem. These allow us to carry out region formation and SPM allocation simultaneously. We can generate optimal partitions for programs expressed either as call trees or by a restricted form of control-flow graph (CFG). We show that this approach obtains superior results to the previous two-step approach. We apply our algorithm to various programs and SPM sizes and show that it reduces the execution time cost for executing those programs relative to execution with cache.

I. Introduction

Instruction scratchpads (I-SPMs) are well-known as replacements for instruction cache [1]. k bytes of SPM space take up less physical space and require less energy than k bytes of cache [2]. This is useful for any embedded system, but real-time embedded systems also benefit from the predictability of SPM [3], [4]. With SPM, access time is independent of preceding memory accesses, an advantage for computing a program's *worst-case execution time* (WCET) [5].

However, programs must be modified to use SPM. The program must be analyzed to determine which parts are most frequently executed (in the average or worst case). These are placed in SPM [6], [7]. There is a large performance difference between the execution speed for code stored in external memory and code stored in SPM [5], so it is insufficient to store only the most frequently executed subset of the program in SPM. If the program is too large for SPM, then the SPM contents must change as the program is executed [8], [9]. This means dividing (*partitioning*) the program into sub-units named *regions*, each with its own usage of SPM space [3].

This partitioning process is not as straightforward as it may sound, and a lack of a scalable, systematic way to do it has limited the adoption of SPM technology. This is particularly noticeable in the real-time systems field, where researchers and engineers continue to prefer to analyze the worst-case behavior of cache-based systems, rather than adopt newer technology that is inherently more predictable. The problem is a lack of good tools. While SPM has many benefits which have been demonstrated experimentally, it is not yet a useful building block for further research or commercial products because there are no commercial-grade tools for partitioning programs, allocating SPM space, and subsequently performing WCET analysis. We contrast this with the state of the art in WCET analysis for cache-based systems: several high-quality tools already exist [10].

Good tools are built around good algorithms, and in order to develop high-quality tools for SPM program modification, we must find a good algorithm for SPM allocation. The algorithm must be able to generate a dynamic SPM allocation, i.e. one that can change as the program executes. It must behave consistently, being free of anomalies and pseudorandom behavior that might be triggered unexpectedly by changes in the input program. Ideally, it should be an *optimal* algorithm producing the best SPM allocations possible according to some easily-understood principle. Developers need tools that are well-behaved; these in turn depend on well-behaved algorithms.

SPM allocation algorithms have been proposed before [2], [3], [8], [11]. But these are approximate, suboptimal and heuristic approaches. A common approach is two-step SPM allocation, where regions are formed in one step, and SPM space is allocated in the second, separately for each region. This is an approximation, as is the choice to form regions around program structures such as loops and methods.

The first contribution of this paper is a polynomial-time k-partitioning algorithm for call trees that generates an optimal, dynamic SPM allocation that minimizes the time cost for transferring code into SPM, given a specific execution profile. The second contribution is an extension of that algorithm to support a restricted form of *control-flow graph* (CFG). We compare our new algorithm with two-step SPM allocation and with a cache-based solution.

In this paper we identify dynamic SPM allocation as a graph k-partitioning problem and specify the properties of an optimal solution (section II). We identify a suitable algorithm for k-

partitioning (section III). Unfortunately, this algorithm carries two disadvantages. Firstly, it does not correctly account for the time taken to move from one region to another, so we correct this by proposing a new algorithm in section IV. Secondly, it does not support any program representation other than a tree, so we propose a new program representation to handle trees with back edges, and a further algorithm extension in section V. Section VI compares our new algorithm with previous work by experiment, and section VII concludes.

II. Dynamic SPM Allocation

Dynamic SPM allocation is a *problem* in the formal sense: a statement requiring a *solution* by means of an algorithm. Formally, a solution is some modification of the input program to enable it to make use of SPM. We consider only solutions that are *valid* in the sense that they can be implemented without changing the meaning of the program. There are usually many possible solutions; these form the *solution space*.

A solution has various associated *metrics*, such as its average and worst-case execution time, its total energy consumption, its external memory requirements and the total amount of information transferred between external memory and SPM.

Solutions may consider program *instructions*, program *data* or both. Orthogonally, solutions may be *dynamic* or *static*. A *static* solution specifies which code/data should be loaded into SPM before execution begins; this mapping is not changed during execution. A *dynamic* solution adds code to the program to update the contents of SPM during execution. Each instance of this code is a *reload point*.

An *optimal solution* minimizes one of the metrics, such that no other solution in the solution space reduces the metric any further. An *optimal algorithm* is an algorithm that produces an optimal solution with respect to one metric.

A. Previous Work

An SPM allocation algorithm produces a solution to a problem specified (at a minimum) by an input program and an SPM size k bytes. Development of such algorithms has been a research focus during the past decade. SPM allocation algorithms are an important research issue for real-time systems because of the need to maximize predictable performance [5]. They are also important for low-power embedded systems because of the need to minimize energy while retaining the ability to run complex applications [1].

Recent attempts to specify SPM allocation algorithms have been fairly fruitful. Various metrics have been considered: algorithms have been designed to minimize average execution time [11], energy consumption [6], and worst-case execution time [3]. Static [7] and dynamic [9] solutions have been considered. Solutions have considered program instructions [3] and program data [4]. SPM allocation algorithms have even been specialized to particular forms of data access, such as arrays used by loops [12], and specific sorts of instruction sequence, such as traces [2].

Static SPM allocation algorithms can be optimal. For instance, in [6], a static solution is generated by solving a knapsack problem [13]. The solution generated is optimal with respect to average execution time or energy consumption, given average execution profile data.

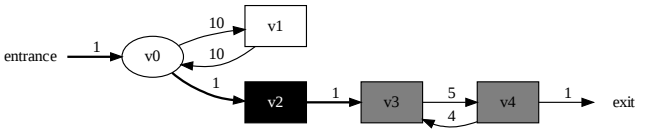


Fig. 1. A simple control-flow graph (CFG) with three reload points. Each edge is marked with its execution frequency when the program is executed. Each of the three regions $[v_0, v_1]$, $[v_2]$ and $[v_3, v_4]$ are shaded differently: edges containing reload points are marked in bold.

However, previous dynamic SPM allocation algorithms are not claimed as optimal. They rely on heuristic algorithms that do not check or prune the entire solution space, instead greedily seeking a solution without backtracking [3], [4], or deferring allocation decisions to runtime heuristics [8]. Some also restrict reload points to specific locations (e.g. loop entry/exit [3], trace entry/exit [2]).

A common design pattern is identified by ourselves as *two-step SPM allocation*. In a two-step process, reload points are determined in a first pass over the program, and then SPM allocation is carried out separately for each sub-unit of the program that is separated from the others by reload points. We call these sub-units *regions*. The two-step process is typical of many previous works [3], [9], [14] and it is suboptimal because the dependency between the two steps (region formation, allocation) is not truly unidirectional. For an optimal solution, the two problems have to be solved together.

B. Formal Description of Problem

A program may be described by a *control-flow graph* (CFG). Figure 1 shows a simple program's CFG with three reload points: at the entrance (the root, v_0), before v_2 and before v_3. These form regions $[v_0, v_1]$, $[v_2]$ and $[v_3, v_4]$.

The CFG $G = (V, E)$ consists of vertexes $v \in V$, and control-flow edges $(v_x, v_y) \in E$. As G represents a program, the vertexes $v_0, v_1, ..., v_{n-1}$ represent blocks of code (typically *basic blocks* [15]). Each vertex v has a size given in bytes, which is represented by the vertex size function $S(v)$.

An edge (v_x, v_y) represents a transition between two blocks v_x and v_y, caused by the control flow in the program. The function $W(v_x, v_y)$ represents the *edge weight* of that edge: the number of times it is taken during execution.

A *set partition* R is a collection of disjoint subsets of V whose union is V [16]. Each of these subsets is known as a *region*. Each region has a different SPM allocation. Reload points are created on every edge (v_x, v_y) that leads from one region to another.

We define R as a mapping from each vertex $v \in V$ to a unique identifier r which is assigned to each region, so that $R : v \to r$. An edge (v_x, v_y) has a reload point if (and only if) $R(v_x) \neq R(v_y)$.

For convenience, we have a shorthand form to represent R. For Figure 1, the partition R can be expressed as $R = [[v_0, v_1], [v_2], [v_3, v_4]]$.

Each region r has a size $S(r)$ given in bytes, calculated as the sum of the sizes of all vertexes within r:

$$S(r) = \sum_{\forall v \in V.R(v)=r} S(v) \quad (1)$$

Any R is a solution to the problem described by the CFG G and the SPM size k bytes. The solution is valid if $\forall r.S(r) \leq k$, i.e. no region is larger than the available SPM space.

C. Graph Partitioning

Dynamic SPM allocation can be represented as a *graph k-partitioning* problem. In general, the solution to a graph partitioning problem is a set partition R that minimizes some criteria (usually the sum of inter-region edge weights) while respecting some bound on the size of each region [17].

However, most graph partitioning algorithms are not suitable for the dynamic SPM allocation problem. They can be divided into two classes. *Balanced* graph partitioning algorithms aim to divide the graph into m regions of approximately the same total size [18], [19]. *Unbalanced* graph partitioning algorithms aim to divide the graph into exactly two regions with a given size ratio [20].

Balanced partitioning produces results that are far from optimal when applied to SPM allocation. Balanced region sizes are unimportant; what matters is the hard limit k on the size of each region.

Unbalanced partitioning produces somewhat better results, as we can repeatedly divide a graph until we reach regions small enough to be placed in SPM, and one paper on dynamic SPM allocation used exactly this technique [11]. Unfortunately the result is only approximate. Each division or *cut* may be *locally optimal*, but in graph problems, repeated locally-optimal decisions may not lead to the global optimum.

A third class of graph partitioning algorithm is required. This is k-partitioning, where each region size is bounded by k. k-partitioning has not been explored as thoroughly as balanced and unbalanced partitioning, but some previous work exists [21]–[23]. As far as we know, k-partitioning has not previously been formally identified as a solution for the problem of SPM allocation.

D. Our Metric

In this paper, we consider a dynamic SPM allocation *problem* as consisting of a program $G = (V, E)$, an edge weight function $W(v_x, v_y)$, an SPM size k, and a bus transfer time function $L(s)$. The bus transfer time function gives the amount of time needed to transfer s bytes from external memory to SPM on the intended platform for G.

We consider an *optimal* solution to this problem to be R, a set partition of V that minimizes the total time spent transferring data from external memory to SPM. We can formally define this as a cost function:

$$\Gamma(R) = \sum_{\forall (v_x,v_y) \in E.R(v_x) \neq R(v_y)} W(v_x, v_y) L(S(R(v_y))) \quad (2)$$

Minimizing $\Gamma(R)$ is equivalent to minimizing the typical execution time of the program, because $\Gamma(R)$ is the only aspect of the program that is changed by partitioning, as we assume that the whole of each region is stored in SPM.

We know there is no value in running any part of the program from external memory; if some part is rarely executed, then that part should be in its own region.

We assume that reload points do not change the size of the program. We also assume that the edge weight function $W(v_x, v_y)$ is sufficient to describe the program. This requires that the program be *bounded*; infinite loops and infinite recursion are not supported. A solution R is generated with respect to a single execution path, so our optimal solutions do *not* minimize the WCET unless the *worst-case execution path* (WC path) is unaffected by partitioning [24]. We do not consider the possibility that the WC path may change as a result of partitioning, though WCET analysis can be performed on the solution, and may guide the generation of a new edge weight function $W(v_x, v_y)$. It is probably not feasible to generate a k-partition that minimizes the WCET. Though hill-climbing approaches can be used to reduce WCET [3], [4], [24], they are two-step allocation algorithms that rely on WCET analysis as part of evaluating $\Gamma(R)$.

E. Further Definitions

The bus transfer time function gives the length of time needed to transfer s bytes across the bus from external memory to SPM. $L(s)$ is commonly defined as follows [3], [5]:

$$L(s) = \textit{setup time} + \lceil \frac{s}{\textit{bus bandwidth}} \rceil \quad (3)$$

or (equivalently) $L(s) \equiv s$ + *setup time* $\times$ *bus bandwidth*.

III. k-Partitioning

A graph k-partitioning algorithm produces a partition R for a program $G = (V, E)$, given an edge weight function $W(v_x, v_y)$ and a vertex size function $S(v)$, in order to minimize the value of a cost function $\Gamma(R)$ while respecting an upper bound k on the size of each region.

Currently, there is no known *general* k-partitioning algorithm which can operate on any graph. It is likely that the problem is NP-hard, as balanced graph partitioning is NP-hard [19]. We note that with k-partitioning the optimal number of regions is initially unknown, whereas balanced graph partitioning assumes a specific number of regions.

We can generate k-partitions by exhaustive search, by assigning a Boolean value to each graph edge to represent whether that edge is a reload point or not, and then searching through all $2^{|E|}$ possible assignments for those Boolean values. Some of the regions generated in this way may be larger than k; others may be invalid in that vertexes on both sides of some reload point are in the same region. Clearly, this will be an $O(2^{|E|})$ search, and intractable for most programs.

Even a simple program such as one of the well-known Mälardalen Real-time Technology Center (MRTC) benchmarks [25] may contain hundreds of edges between basic blocks: far too many for exhaustive search.

Some graph partitioning problems can be expressed as *integer linear programming* (ILP) problems, with binary variables

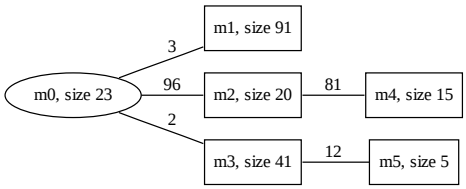


Fig. 2. A call tree containing 6 methods. Vertexes represent methods: each is labeled with its size. Edges represent call/return relationships between methods: each is labeled with its frequency.

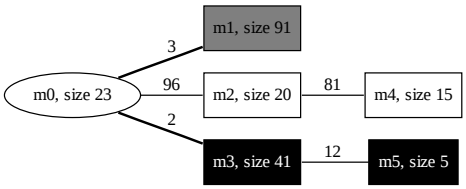


Fig. 3. Figure 2, partitioned for $k = 100$. Each region is shaded differently: reload point edges are marked in bold.

representing region membership [26]. The representation is solved to minimize transition costs using well-known ILP solver algorithms. k-partitioning can be expressed in this way, but the number of variables and constraints is $O(|V|^2)$ because there may be up to $|V|$ regions as well as $|V|$ vertexes.

A. k-Partitioning For Trees

A class of polynomial-time (*P-time*) k-partitioning algorithms exist for specific types of graph, namely *trees*. These are now discussed.

A tree is a graph in which each vertex v_y has exactly one entrance edge, with the exception of the root v_0, which has no entrance edges at all. The source of this entrance edge is called the parent, v_x. As a consequence of this property, trees contain no cycles, and there is exactly one path from the root to any vertex. We shall return to the impact of these restrictions in section V-A. In the meantime, we observe that any program can be represented as a *call tree* in which every vertex is a method and every edge represents a call/return relationship between two methods, i.e. $(m_x, m_y) \in E$ means method m_x calls method m_y.

Figure 2 shows a call tree for a simple program. In this tree, each vertex represents a method. The root m_0 represents the entry method (e.g. main() in a C program). Each vertex is labeled with its size (e.g. method m_0 has size 23, so $S(m_0) = 23$) and each edge is labeled with its weight (e.g. m_1 is called 3 times from m_0, so $W(m_0, m_1) = 3$).

We use m_x, m_y in place of v_x, v_y to highlight the distinction between the call tree representation of the program and a general graph representation. As Figure 2 is a call tree, each edge is bidirectional, with $W(m_x, m_y) = W(m_y, m_x)$.

Kundu and Misra [21] specify an algorithm to produce optimal k-partitions of graphs in $O(n)$ time for an n-vertex tree, provided that each edge weight $W(v_x, v_y) = 1$.

A branch and bound algorithm given by Leupers and Marwedel [23] allows arbitrary edge weights, and prunes the search space by enforcing limits on (a) the region size k, and (b) the value of the best solution found so far. The problem is that the runtime of the algorithm is still $O(2^n)$ in the worst case, which is likely for large k.

B. Lukes' Algorithm

Lukes [22] gives a P-time solution for optimal k-partitioning that allows arbitrary edge weights. The time complexity is $O(k^2 n)$. Like the Kundu and Misra algorithm, Lukes' algorithm requires a program expressed as a tree.

Lukes' algorithm produces an optimal partition R for the tree $G = (V, E)$ which minimizes the number of inter-region transitions, calculated as a sum of the weights of all edges that connect one region to another. This is *related to*, but is *not the same as* Γ (equation 2). The new definition is:

$$\Gamma'(R) = \sum_{\forall (m_x, m_y) \in E . R(m_x) \neq R(m_y)} W(m_x, m_y) \qquad (4)$$

Trees are important to the Kundu, Misra and Lukes algorithms because they solve the partitioning problem by divide-and-conquer. They proceed from the leaves to the root, determining optimal partitions for successively larger subtrees.

Lukes' algorithm allows a large program to be solved quickly but approximately by scaling every method size and k by some constant factor. This is ideal for use within a compiler; the default setting can be a fast but approximate allocation, with a slower, optimal allocation available through command-line parameters to adjust the scale factor.

For $k = 100$ and Figure 2, the output of Lukes' algorithm is $R = [[m_0, m_2, m_4], [m_1], [m_3, m_5]]$. This appears on the graph as shown in Figure 3. There are three regions in the partition. One is $[m_0, m_2, m_4]$, with combined size 58. Another is $[m_1]$, with size 91. The third is $[m_3, m_5]$, with size 46. The partition contains $\Gamma'(R) = 5$ inter-region transitions: 3 between m_0 and m_1, and 2 between m_0 and m_3.

C. Lukes' Algorithm in Detail

Data structures. Lukes' algorithm is based on a data type D, defined as $D : (m, s) \rightarrow R$, i.e. a mapping from a *key* (m, s) onto a *data item* R. The keys consist of a vertex $m \in V$ and a size $s \in [0, k]$. The data item is a partition R for the subtree rooted at m, in which $s = S(R(m))$, i.e. the size of the region containing m is s. There is at most one data item per key, and $R = \varnothing$ for unknown keys.

An *update* procedure is defined which substitutes a new partition R into a variable $d \in D$, but only if that partition has a better value (Γ') than the previous occupant:

```
procedure update(d, m, R):
  s ← S(R(m))
  if (d(m, s) = ∅ or Γ'(d(m, s)) > Γ'(R)):
    d(m, s) ← R
  endif
  if (d(m, 0) = ∅ or Γ'(d(m, 0)) > Γ'(R)):
    d(m, 0) ← R
  endif
endproc
```

Lukes proved that it is only necessary to store the best value R for any particular (m, s) in order to eventually

reach an optimal R for the whole graph. The efficiency of the algorithm depends on its rejection of inferior partitions. The second conditional statement within *update* ensures that $d(m, 0)$ contains the best value R known for m as a whole.

Operations. For each vertex m_x and each child m_y, Lukes' algorithm combines the known partitions for m_x with the known partitions for m_y. This is done in two ways:

Concatenation creates a region boundary on the edge (m_x, m_y). The best known partition containing m_y, $d(m_y, 0)$, is combined with a partition containing m_x, $d(m_x, s)$.

Merging creates a partition $d(m_x, s_x + s_y)$ from two partitions $d(m_x, s_x)$ and $d(m_y, s_y)$. This is done by concatenating the partitions and then merging the region containing m_y with the region containing m_x.

Lukes proved that these two operations were sufficient if they are executed for every possible combination of sizes.

Algorithm. Initially, $d \in D$ is populated with a trivial partition for each vertex:

```
foreach m ∈ V:
  d(m, S(m)) ← [[m]]
endfor
```

Then, each vertex $m_x \in V$ is visited in depth order:

```
foreach m_x ∈ V, deepest first:
  foreach m_y where (m_x, m_y) ∈ E:
    d' ← copy ← d
    foreach s_p in [0, k]:
      update(d', m_x, d(m_x, s_p) + d(m_y, 0))
      foreach s_c in [1, k − s_p]:
        R ← d(m_x, s_p) + d(m_y, s_c)
        merge R(m_x) and R(m_y)
        update(d', m_x, R)
      endfor
    endfor
    d ← copy ← d'
  endfor
endfor
```

The loops visit every vertex m_x, every edge from that vertex (m_x, m_y), and every known partition containing m_x of size s_p. At that point, concatenation is attempted to produce a new $d(m_x, s_p)$ including the m_y subtree. Then, inside the innermost loop, each $d(m_x, s_p)$ is merged with each $d(m_y, s_c)$.

When the algorithm is complete, the optimal partition $R = d(m_0, 0)$, where m_0 is the root.

Time complexity. Lukes' algorithm has $O(k^2 n)$ time complexity. This is derived from three nested loop iterations: foreach s_c (k times), foreach s_p (k times), and the two outermost loops, which have a combined total of $n - 1$ iterations for a tree containing n methods, as such a tree contains $n-1$ edges.

This time complexity requires merging, concatenation and evaluation of $\Gamma'(R)$ to be $O(1)$ operations. We achieve this by representing partitions as binary trees and storing partially-evaluated copies of $\Gamma'(R)$ for successive subtrees. The details of these optimizations can be found within the source code of our experimental implementation (section VI).

IV. Incorporating Loading Costs

Lukes' algorithm solves a problem, but it is not the same problem as dynamic SPM allocation. $\Gamma'(R)$ is minimized, but a solution that minimizes $\Gamma'(R)$ does not necessarily also minimize $\Gamma(R)$.

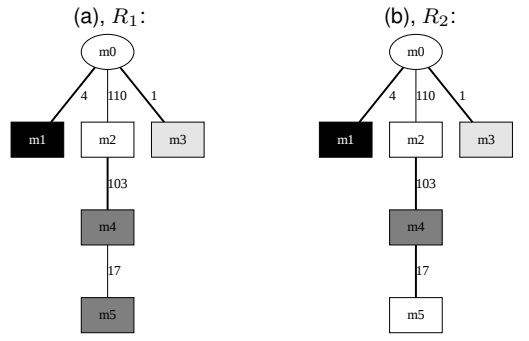


Fig. 4. A call tree partitioned in two different ways. Partition R_1 is chosen to minimize $\Gamma'(R_1)$, while R_2 is chosen to minimize $\Gamma(R_2)$. The size of each vertex is 4, $k = 8$, and $L(s) = s$.

A. Example

Figure 4 shows a call tree partitioned (a) by using Lukes' algorithm to minimize Γ', and (b) by using exhaustive search to minimize Γ. Since Figure 4 is a call tree, each edge is bidirectional, i.e. $W(m_x, m_y) = W(m_y, m_x)$.

Notice that R_1 and R_2 are not the same, and not equivalent in terms of Γ' and Γ. R_1 is better for Γ' while R_2 is better for Γ. If minimizing the *number* of inter-region transitions is most important, then it is best to merge m_4 and m_5. However, if minimizing the *cost* of inter-region transitions is most important, then m_4 and m_5 should be separate. Without this separation, both would need to be loaded together 103 times on the (m_2, m_4) edge.

Lukes' algorithm is therefore *not* optimal with respect to Γ. We must adapt it in order to minimize Γ; the cost of inter-region transitions is not fixed (as in Γ') but dependent on the destination region size (as in Γ).

B. Obvious, Incorrect Solution

The obvious solution is to substitute $\Gamma(R)$ for $\Gamma'(R)$ within the *update* method of Lukes' algorithm (section III-C).

This would require us to take into account the cost of *entering* a region at the root of a subtree m_x, because this cost is not constant when $\Gamma(R)$ is used, being dependent on $S(R(m_x))$. We do that by (1) defining $R(m_w)$ as a null region of zero size for all m_w that are *not* in R, and (2) adding an entry edge for the entire tree, $(\varnothing, m_0)$, with $W(\varnothing, m_0) = 1$.

Unfortunately, though accounting for region entry costs is important, it is not enough for a correct solution. Figure 5(a) is a call tree that has been partitioned using Lukes' algorithm, using $\Gamma(R)$ in place of $\Gamma'(R)$. For this partition, $\Gamma(R) = 6292$. However, this is not the optimal partition. The actual minimum $\Gamma(R') = 6208$ with the partition shown in Figure 5(b).

The problem is that $\Gamma(R)$ is not a single value because the region containing the current subtree root m_x can still grow in size (unless $m_x = m_0$). The size $S(R(m_x))$ affects the value of the partition, but whilst $R(m_x)$ can still be extended, we have only a lower bound for this size.

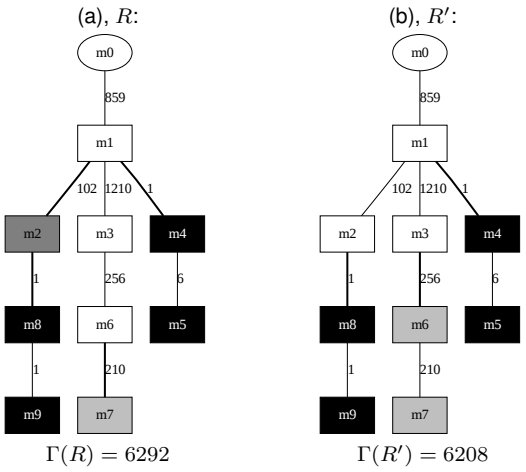


Fig. 5. (a): a non-optimal partition R, generated by substituting Γ into Lukes' algorithm. (b): the partition R' chosen to minimize $\Gamma(R')$. The size of each vertex is 4.

There is an *expansion value* α representing the combined weight of methods that may be added to $R(m_x)$ in future. Figure 5(a) is a result of the assumption that expansion will not change $S(R(m_x))$, and hence the optimal partition for $\alpha = 0$ is the same for all other α. In fact, the optimal partition is highly dependent on α.

Consider the subtree with root m_1. While processing m_1, the algorithm considers two possible partitions, R_a and R_b, starting with regions $[m_1, m_3, m_6]$ and $[m_1, m_2, m_3]$ as seen in Figure 5. The first region $R(m_1)$ has size 12 in each case, so these possibilities are both stored in data structure element $d(m_1, 12)$. At some point, *update* will need to choose between them, and this choice will be based upon the value of $\Gamma(R)$.

Now suppose that the size $S(R(m_1)) = 12 + \alpha$. The values for R_a and R_b are:

$$\Gamma(R_a) = 102(12 + \alpha + 4) + 1(4 + 8) + 210(12 + \alpha + 4) + 1(12 + \alpha + 8) = 5024 + 313\alpha \quad (5)$$

$$\Gamma(R_b) = 1(12 + \alpha + 8) + 256(12 + \alpha + 8) + 1(12 + \alpha + 8) = 5160 + 258\alpha \quad (6)$$

Lukes' algorithm makes the wrong choice because it assumes that $\alpha = 0$, i.e. that there is no expansion. In this situation, R_a is better (5024 versus 5160). But in fact $R(m_1)$ will be expanded to include m_0, so $\alpha = S(m_0) = 4$. In this situation, R_b is better (6192 versus 6276). The choice leads to a suboptimal result. More complex counter-examples involving larger graphs exist, but all follow this general form.

The substitution fails because it assumes that $\alpha = 0$ and therefore that R_1 is always optimal. The failure may be observed whenever two incomplete partitions have the same subtree root (m) and the same size ($S(R(m))$) *but* two or more different optimal partitions for different α. It is therefore essential to store different partitions for different α as well as different first region sizes.

C. Extended Lukes' Algorithm - 1

In this section, we describe an extension to Lukes' Algorithm which we call Extended Lukes Algorithm - 1 (ELA-1) which enables the use of $\Gamma(R)$ as a cost function. The algorithm uses a new data type $D' : (m, s, \alpha) \rightarrow R$. That is, as well as being identified by subtree root (m) and first region size (s), incomplete partitions are identified by expansion value α. The new *update'* procedure takes α into account:

```
procedure update'(d, m, α, R):
  s ← S(R(m))
  S(R(m)) ← s + α
  if (d(m, s, α) = ∅ or Γ(d(m, s, α)) > Γ(R)):
    d(m, s, α) ← R
  endif
  if (d(m, 0, 0) = ∅ or Γ(d(m, 0, 0)) > Γ(R)):
    d(m, 0, 0) ← R
  endif
endproc
```

The new algorithm is as follows. Initially, $d \in D'$ is populated with a trivial partition for each vertex and for each possible expansion value:

```
foreach m ∈ V:
  foreach α in [0, k − S(m)]:
    d(m, S(m), α) ← [[m]]
  endfor
endfor
```

Then, each vertex $m_x \in V$ is visited in depth order. This involves a new loop in which all possible expansion values are tested:

```
foreach m_x ∈ V, deepest first:
  foreach m_y where (m_x, m_y) ∈ E:
    d' ← copy ← d
    foreach s_p in [0, k]:
      foreach α in [0, k − s_p]:
        update'(d', m_x, α, d(m_x, s_p, α) + d(m_y, 0, 0))
        foreach s_c in [1, k − s_p − α]:
          R ← d(m_x, s_p, α) + d(m_y, s_c, α)
          merge R(m_x) and R(m_y)
          update'(d', m_x, α, R)
        endfor
      endfor
    endfor
    d ← copy ← d'
  endfor
endfor
```

Intuition. The difference between Γ and Γ' is the dependence on region sizes. The optimal partition for a subtree may depend on the size of the region at the subtree root, and this region can include the parent of the subtree and other vertexes. The size of this expansion is α. We compute optimal partitions for all possible α values at each subtree and so handle all possible region sizes. No partition is eliminated unless a more valuable alternative has been found.

ELA-1 is equivalent to Lukes' algorithm if $\Gamma'(R)$ is used in place of $\Gamma(R)$. This is because $\Gamma'(R)$ is independent of region size and therefore α.

Time Complexity. ELA-1 has $O(k^3 n)$ time complexity, i.e. k times more than Lukes. This is because of the additional loop iteration (foreach α).

Not all values of $\alpha \in [0, k - s_p]$ are possible for any particular tree and m_x, but it is not easy to work out which α values can occur, because this involves all connected combinations of methods outside of m_x's subtree. In any case, the worst case

is $O(k)$ possible values. Our implementation simply considers all $\alpha \in [0, k - s_p]$ (section VI).

V. Incorporating Program Structure

ELA-1 solves the problem of minimizing $\Gamma(R)$ for a program $G = (V, E)$, but it relies on an approximation, because the program has to be represented as a call tree. The solutions it produces are only optimal for this representation; they are not necessarily optimal for more general forms such as control-flow graphs (CFGs).

A. Representing Programs as Call Trees

Lukes' algorithm and ELA-1 both require the input program graph $G = (V, E)$ to be a tree. In section IV we represented programs as call trees.

All programs can be expressed as call trees, as cycles due to recursion can be converted into loop iteration, and multiple paths from the root m_0 to some method m_i can be removed by cloning a subtree on the path between m_0 and m_i.

The main issue with a call tree representation is that it omits the internal structure of methods. A method is regarded as an indivisible unit. While it may call other methods with a known frequency, there is no information about the order of these calls, nor any information about what parts of the method may be executed before or after each call.

This is an approximation with serious disadvantages. Firstly, a method m may be too big for SPM storage (i.e. $S(m) > k$), in which case Lukes' algorithm and ELA-1 cannot find any solution at all, as there is no way to divide methods into smaller units. Secondly, methods that do fit in SPM are entirely loaded on every entry, even if internal control flow means that parts of the method are rarely executed. Thirdly, parts of a method may be entirely unreachable if the method is entered by a return operation. For instance, consider a method m_x containing three statements:

```
method m_x:
  A
  call (m_y)
  B
endmethod
```

If m_y is in a different region to m_x, then returning from m_y will cause *all three* statements to be reloaded, *including* A, even though A is now unreachable.

The call tree representation is lacking any notion of *sequence*. There is no information about the order of operations. With a call tree, an SPM allocation algorithm cannot make use of knowledge about the order of events in the program.

B. Unsuitability of Syntax Trees

A call tree representation may be approximate, but programs can be represented by other types of trees. One option is a *syntax tree*, which expresses the structure of a method [15]. Figure 6 gives an example of this form.

Syntax trees specify the sequence of operations. They are "executed" in depth-first order from left to right, so B follows A in Figure 6. Certain nodes (e.g. if, while) have special meanings which alter the order of execution for the children.

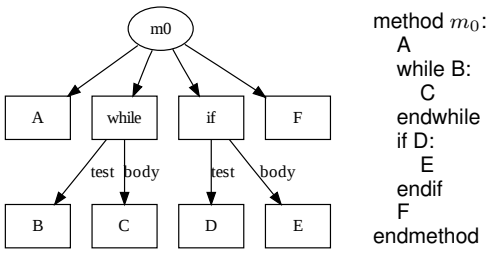


Fig. 6. *Left*: a syntax tree for the method m_0. *Right*: the definition of m_0. The tree is "executed" in depth-first order from left to right.

For instance, E is only executed if D "returns" true, and the sequence B-C repeats until B "returns" false.

By decomposing methods into their constituent primitives, such as loops, conditions and expressions, the syntax tree representation allows ELA-1 to produce a partition which is closer to the optimal. No changes to ELA-1 are required; the syntax tree is just passed to ELA-1 as if it were a call tree. This allows ELA-1 to handle methods that are too large to be stored in SPM.

However, the resulting partition is still only approximately optimal, because ELA-1 is not able to make use of the sequence information, only the structural information (e.g. m_0 contains A, F). It is impossible for ELA-1 to create any partition containing the regions [A, while] and [if, F] because these two regions are not internally connected via the root m_0. But such a partition would be valid within the syntax tree representation, because a reload point could be placed between "endwhile" and "if D" (Figure 6). The algorithm cannot generate all valid solutions so we cannot rely on it to find the optimal solution.

C. Unrolling a Control-Flow Graph

ELA-1 and Lukes' algorithm provide one way to make use of sequence information. The program graph $G = (V, E)$ can be converted to a tree by a process of *unrolling*.

All of the paths through a program form a tree. The program entrance is the root: each conditional branch effectively splits control flow in two directions. Each vertex $v \in V$ contains a small fragment of code (e.g. a basic block) and edges $(v_x, v_y) \in E$ represent transitions between basic blocks. We call this a *control-flow tree* (CFT). It is identical to a CFG where each vertex has one parent (except the root v_0, with no parents). Figure 7 shows the CFT for Figure 6 assuming a maximum of two while loop iterations.

The CFT can be partitioned using ELA-1, and the result R will be a precisely optimal solution (in terms of $\Gamma(R)$) which can be applied to the original program without any approximation because reload points could be introduced before any basic block.

However, the CFT will contain $O(2^\beta)$ vertexes for a program containing β branches. This is not practical. A program containing no branches at all (i.e. a single-path program [27]) might be an exception, but even in that special case, method

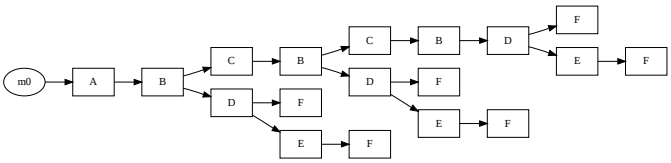
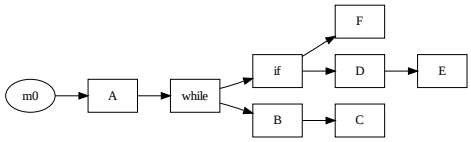


Fig. 7. The CFG for Figure 6, unrolled to produce a *control flow tree* (CFT) assuming a maximum of two while loop iterations. Note that there are six copies of statement F because six different paths from m_0 lead to F.



$B = \{$ (B, while), (C, while), (D, if), (E, if), (F, m_0)$\}$

Fig. 8. *Top*: the CFT for Figure 6. *Bottom*: back edges stored in a separate data structure B.

calls/returns would need to be inlined for the purposes of generating the tree.

D. Back Edges

With the exception of the unrolling technique, tree representations always involve some approximation which means that an optimal solution generated by ELA-1 is not necessarily optimal when applied to the original program.

We propose a new program representation comprising a CFT, $G = (V, E)$ and a set of *back edges* $B : v_y \rightarrow v_x$. Back edges can be used to implement loops, method calls, and (generally) any place where multiple code paths merge together. We define a back edge as a link between a vertex v_y and v_x, where v_x is some *ancestor* of v_y (i.e. parent, grandparent, great-grandparent, etc.). We call this representation CFT + BE: control flow tree, plus back edges.

A CFT is produced as described in section V-C, but every edge (v_x, v_y) in which v_y is an ancestor of v_x is deleted and replaced by a back edge, instead of replicating v_y. This is achieved by storing the contents of the path from v_0 to v_x in some rapidly-searchable data structure such as a hash table. Figure 8 shows the CFT and B for Figure 6.

A back edge $(v_y, v_x) \in B$ has an edge weight $W(v_y, v_x)$ just like a regular edge $(v_x, v_y) \in E$. The cost function $\Gamma(R)$ should consider both types of edge together.

E. Extended Lukes Algorithm - 2

Extended Lukes Algorithm - 2 (ELA-2) adds support for our CFT + BE program representation. It makes use of the functions of ELA-1 described in section IV-C. As ELA-2 processes a non-approximate representation of a program (e.g. Figure 8) it is capable of producing partitions which can be mapped directly onto the machine code of the input program, implementing an optimal dynamic SPM allocation.

We found that it is necessary to decide *a priori* which back edges will be inter-region transitions. Otherwise, the cost of these edges cannot be accurately incorporated by ELA-1. The problem is that the true value of a back edge (v_y, v_x) does not become apparent until region assignment is complete for v_x. Consequently, the *update'* method may reject solutions that only turn out to be good once v_x is processed. There is no straightforward solution for this problem. It is not a matter of adding some sort of expansion value like α because the back edge cost may be zero if the loop is complete.

Algorithm. We assign a Boolean value $b(v_y, v_x)$ to each back edge (v_y, v_x) to represent whether that edge is a reload point or not. We execute ELA-1 for each of the $2^{|B|}$ possible assignments for those Boolean values, producing up to $2^{|B|}$ different partitions R. ELA-2 returns the R with the lowest $\Gamma(R)$, this being the optimal solution. ELA-2 is equivalent to ELA-1 if there are no back edges.

Time Complexity. ELA-2 has $O(2^{|B|}k^3n)$ worst-case time complexity, i.e. $2^{|B|}$ times more than ELA-1.

In practice we can make a few optimizations that make the algorithm manageable. Firstly, some back edges are certainly reload points, because the total size of the vertexes on the path from v_y to its ancestor v_x is greater than k. Secondly, two Boolean values $b(v_{y_1}, v_{x_1})$ and $b(v_{y_2}, v_{x_2})$ are independent if v_{x_1} and v_{y_1} cannot be part of the same region. This independence allows subtrees to be solved separately. We plan to investigate these possibilities in the future.

VI. Implementation and Experiments

ELA-2 produces an optimal partition R of a program that minimizes $\Gamma(R)$, the total time spent transferring data from external memory to SPM. In this section, we demonstrate its capabilities by experiment. Source code for experiments may be downloaded from http://www.cs.york.ac.uk/rts/rtslab/.

Our experimental comparisons consider a local memory size k, an input program $G = (V, E)$, and a definition of the bus transfer time function $L(s)$. Unless stated otherwise, the experiments assume that the time taken to transfer s bytes of data across the bus is $L(s) = 20 + s$. We vary the SPM size $k \in \{128, 256, 512, 1024, 2048, 4096\}$. The input programs are benchmark software taken from the Mälardalen Real-time Technology Center (MRTC) collection [25].

For each experiment, we take one program from the benchmark collection and compile it for ARM using LLVM version 2.6 [28]. Optimization level `-Os` is used. We apply the LLVM `opt -loop-extract` command to the intermediate code in order to exline each loop into a separate method [29]. The ARM binaries are executed using the Gem5 simulator [30] to produce a trace of instruction addresses, accessed by the program as it runs. Only main() and functions called by main() are traced.

A. Two-step dynamic SPM allocation versus ELA-2

In this section, we compare a two-step SPM allocation process with ELA-2. Earlier dynamic SPM allocation algorithms are two-step, so this is a comparison between ELA-2 and a whole class of possible algorithms (section II-A). The instruction address trace for each input program is used to construct a CFT + BE representation as described in section

	64	128	256	512	1024	2048	4096
binarysearch	2.30	1.29	1.22	1.22	1.22	1.22	1.22
bsort100	1.43	1.21	65.9	69.9	69.9	69.9	69.9
crc	1.34	1.22	63.3	64.6	102	102	102
edn	1.28	2.04	2.64	3.81	10.8	16.0	16.1
fir	2.54	1.51	1.23	8.48	8.48	8.48	8.48
insertsort	1.47	1.38	9.07	9.67	9.67	9.67	9.67
jfdctint	2.16	3.24	1.86	1.12	1.17	1.17	1.17
matmult	1.75	1.22	18.0	149	153	153	153

TABLE I
THE RATIO $\frac{\Gamma(R_{\text{TWOSTEP}})}{\Gamma(R_{\text{ELA-2}})}$ FOR VARIOUS PROGRAMS AND VARIOUS k.

	64	128	256	512	1024	2048	4096
binarysearch	-	-	1.04	1.04	1.04	1.04	1.04
bsort100	-	1.24	1.39	1.04	1.04	1.04	1.04
crc	-	-	2.20	1.30	1.18	1.18	1.18
edn	-	-	-	-	5.22	1.75	1.16
fir	-	-	1.30	1.06	1.06	1.06	1.06
insertsort	-	-	10.1	1.04	1.04	1.04	1.04
jfdctint	-	-	-	1.34	1.27	1.11	1.11
matmult	-	1.27	1.15	1.48	1.04	1.04	1.04

TABLE II
THE RATIO $\frac{\Gamma(R_{\text{ELA-1}})}{\Gamma(R_{\text{ELA-2}})}$ FOR VARIOUS PROGRAMS AND VARIOUS k.

V-D. This is passed to ELA-2 with SPM size k. The result is a partition $R_{\text{ELA-2}}$ and a value $\Gamma(R_{\text{ELA-2}})$.

For the first step of two-step allocation, we create one region for each method in the program. Some of these methods are also methods within the C source code; others were originally loops that have been exlined by LLVM's `opt -loop-extract` operation. Thus, our region formations match those used in previous work [3], [8].

For the second step, we allocate SPM space within each method according to a profile captured from the instruction address trace. Instructions are allocated to SPM in descending order of execution frequency. As some instructions may not fit in SPM, the result cannot be expressed as a partition R, but we can nevertheless evaluate the equivalent of $\Gamma(R)$, which we call $\Gamma(R_{\text{TWOSTEP}})$ and define as the total time spent transferring data from external memory to SPM during execution.

Table I shows the ratio of $\Gamma(R_{\text{ELA-2}})$ and its two-step equivalent $\Gamma(R_{\text{TWOSTEP}})$ for two-step allocation, given various benchmarks and various k. Ratios larger than 1.0 mean that ELA-2's partition has a lower time cost. We see that ELA-2 has a large advantage, except for very small k.

ELA-2's advantage is a result of creating regions and allocating SPM space at the same time. ELA-2 finds the optimal region size rather than just the optimal allocation for a region of fixed size. It naturally expands regions to include multiple loops and multiple methods. Two-step allocation could be extended to expand regions by merging, but this would be an awkward and approximate extension, involving a feedback loop between the two steps.

B. ELA-1 versus ELA-2

We compare ELA-1 and ELA-2 as follows. Firstly, the value $\Gamma(R_{\text{ELA-2}})$ is produced for each program as described above. Then, $\Gamma(R_{\text{ELA-1}})$ is produced from a call tree representation of each program, consisting of methods and exlined loops.

$R_{\text{ELA-1}}$ is undefined if the size of some vertex in the call tree is larger than k. When this occurs, Γ is also undefined, and we leave a gap in Table II.

Table II shows the ratio between $\Gamma(R_{\text{ELA-2}})$ and $\Gamma(R_{\text{ELA-1}})$. We see that the two are sometimes very close to each other; this is common for larger values of k. But because methods and loops are indivisible, ELA-1 cannot produce the very best solutions, and cannot operate at all if some vertex is larger than k. Table II is evidence of the call tree disadvantages discussed in section V-A. However, ELA-1 does have the significant advantage of algorithmic efficiency, with P-time complexity.

	64	128	256	512	1024	2048	4096
binarysearch	2.73	2.25	2.21	2.21	2.21	2.21	2.21
bsort100	2.55	0.94	2.09	2.22	2.22	2.22	2.22
crc	1.33	1.08	2.07	1.46	2.05	2.05	2.05
edn	1.17	1.32	1.19	1.82	2.11	1.95	1.69
fir	1.63	1.69	0.81	2.02	2.02	2.02	2.02
insertsort	2.03	1.59	1.95	2.08	2.08	2.08	2.08
jfdctint	1.50	1.48	1.33	1.94	1.91	1.81	1.81
matmult	1.94	1.47	2.75	5.79	1.85	1.85	1.85

TABLE III
THE RATIO $\frac{\Gamma(R_{\text{CACHE}})}{\Gamma(R_{\text{ELA-2}})}$ FOR VARIOUS PROGRAMS AND VARIOUS k.

C. Cache versus ELA-2

In this section, we compare a direct-mapped cache of size k with ELA-2. A cache comparison is worthwhile to put SPM results into a wider context, as caches are a well-understood technology, though we must remember that $\Gamma(R_{\text{CACHE}})$ is very different to $\Gamma(R_{\text{ELA-2}})$. Cache operations are entirely dynamic, so the contents of the cache are not fixed at any point in the program. In contrast, the SPM allocation is fixed for every point in the program. This difference means that a range of execution times are possible for cache simulation.

The size of the cache line c is an additional parameter that affects $\Gamma(R_{\text{CACHE}})$ in two ways. Firstly, the cost of each cache miss is $L(c)$. Secondly, programs that access sequential data may incur fewer misses if the line size is increased. Therefore, we try various cache line sizes $c \in \{4, 8, 16, 32\}$, and choose whichever minimizes $\Gamma(R_{\text{CACHE}})$.

Table III shows the ratio between $\Gamma(R_{\text{ELA-2}})$ and $\Gamma(R_{\text{CACHE}})$. We see that SPM is often significantly better than cache, especially for larger k. This is because the SPM transfers data in blocks of arbitrary size, matching the partition size, whereas the cache transfers data in blocks of size c. Table IV shows that the SPM solution is better if the bus setup cost is increased. If the bus setup cost is very small, or if the SPM block size is small as a result of small k, then cache can hold the advantage, but SPM's large block transfers are preferable for large bus setup costs.

VII. CONCLUSION

Due to its predictability, SPM is an ideal technology for storing program instructions and data within embedded real-

	$L(s) =$				
	$s+0$	$s+10$	$s+20$	$s+50$	$s+100$
binarysearch	1.00	1.89	2.21	3.01	3.94
bsort100	1.03	1.79	2.09	2.81	3.64
crc	0.91	1.78	2.07	2.76	3.63
edn	0.65	1.01	1.19	1.62	2.13
fir	0.29	0.63	0.81	1.22	1.55
insertsort	1.03	1.68	1.95	2.59	3.30
jfdctint	0.75	1.15	1.33	1.74	2.18
matmult	0.85	2.12	2.75	3.85	4.90

TABLE IV
THE RATIO $\frac{\Gamma(R_{\text{CACHE}})}{\Gamma(R_{\text{ELA-2}})}$ FOR VARIOUS PROGRAMS AND L. $k = 256$.

time systems. However, adoption is limited due to the practical difficulty that programs have to be modified in order to use SPM effectively. In order to develop high-quality tools for building programs for systems with SPM, we need high-quality SPM allocation algorithms which behave consistently, being free of anomalies or pseudorandom behavior that can be triggered unexpectedly by changes in the input program. Algorithms should be optimal, or close to optimal, for some well-defined metric.

In this paper, we have developed algorithms to meet these criteria, based upon Lukes' k-partitioning algorithm [22]. ELA-1 and ELA-2 minimize the cost function defined in equation 2, thus minimizing the time spent transferring data from external memory to SPM during execution. ELA-1's optimal solutions are generated by partitioning a call tree representation of the program. ELA-2's optimal solutions are generated by partitioning a restricted form of CFG. The algorithms assume that a provided execution profile is typical.

We have shown that ELA-2 is a significant advance over the previous two-step approach for allocating SPM space. We have shown that the execution time of programs processed using ELA-2 is similar to (or better than) the execution time on a cache-based system. We have also shown that ELA-1 is often close to ELA-2.

Future work. This paper has assumed that reload points do not expand the program, but in reality, each reload point will require some instructions to be added. Though the time cost of executing a reload point is taken into account via equation 3, the space cost is unaccounted for. Future work should address this. It should also consider what optimizations or approximations are possible to remove the $2^{|B|}$ term from the time complexity equation.

REFERENCES

[1] P. Marwedel, *Embedded System Design*. Springer-Verlag New York, Inc., 2006.

[2] M. Verma and P. Marwedel, "Overlay techniques for scratchpad memories in low power embedded processors," *IEEE Trans. VLSI*, vol. 14, no. 8, pp. 802–815, 2006.

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

[4] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET Centric Data Allocation to Scratchpad Memory," in *Proc. RTSS*, 2005, pp. 223–232.

[5] J. Whitham and N. Audsley, "Implementing Time-Predictable Load and Store Operations," in *Proc. EMSOFT*, 2009, pp. 265–274.

[6] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. DATE*. Washington, DC, USA: IEEE Computer Society, 2002, p. 409.

[7] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri, "A post-compiler approach to scratchpad mapping of code," in *Proc. CASES*, 2004, pp. 259–267.

[8] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *J. Embedded Comput.*, vol. 1, pp. 521–540, December 2005.

[9] J.-F. Deverge and I. Puaut, "WCET-Directed Dynamic Scratchpad Memory Allocation of Data," in *Proc. ECRTS*, 2007, pp. 179–190.

[10] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[11] A. Janapsatya, A. Ignjatovic, and S. Parameswaran, "Exploiting statistical information for implementation of instruction scratchpad memory in embedded system," *IEEE Trans. VLSI*, vol. 14, no. 8, pp. 816–829, 2006.

[12] Y. Yang, H. Yan, Z. Shao, and M. Guo, "Compiler-assisted dynamic scratch-pad memory management with space overlapping for embedded systems," *Software: Practice and Experience*, vol. 41, no. 7, pp. 737–752, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1020

[13] E. W. Weisstein, "Knapsack Problem," http://mathworld.wolfram.com/KnapsackProblem.html, 2012.

[14] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *Trans. on Embedded Computing Sys.*, vol. 5, no. 2, pp. 472–511.

[15] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, 1986.

[16] E. W. Weisstein, "Set Partition," http://mathworld.wolfram.com/SetPartition.html, 2012.

[17] C. Walshaw, "Graph Partition Archive: Problem Definition," http://staffweb.cms.gre.ac.uk/~wc06/partition/, 2012.

[18] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," in *Bell Syst. Tech. J.*, vol. 49, 1970, pp. 291–307.

[19] K. Andreev and H. Racke, "Balanced graph partitioning," *Theor. Comp. Sys.*, vol. 39, pp. 929–939, November 2006.

[20] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. DAC*, 1982, pp. 175–181.

[21] S. Kundu and J. Misra, "A linear tree partitioning algorithm," *SIAM J. Comput.*, vol. 6, pp. 151–154, March 1977.

[22] J. A. Lukes, "Efficient algorithm for the partitioning of trees," *IBM J. Res. Dev.*, vol. 18, pp. 217–224, May 1974.

[23] R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," in *Proc. ICCAD*, 1999, pp. 253–256.

[24] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *Proc. CODES+ISSS*, 2007, pp. 143–148.

[25] MRTC, "WCET Benchmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[26] R. H. Bisseling, S. Cerav-erbas, M. Lorenz, R. Pendavingh, C. Reeves, M. Rger, and A. Verhoeven, "Partitioning a call graph," in *Workshop on Combinatorial Scientific Computing*, 2005.

[27] P. Puschner, "Experiments with WCET-oriented programming and the single-path architecture," in *Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 2005.

[28] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. CGO*, 2004, pp. 75–.

[29] R. Spencer and G. Henriksen, "LLVM's Analysis and Transform Passes," http://llvm.org/docs/Passes.html, 2012.

[30] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.