

Static Program Partitioning for Embedded Processors

Bageshri Sathe

Dept. of Computer Science & Engg.,
Indian Institute of Technology, Bombay
Mumbai, India
bageshri@cse.iitb.ac.in

Uday Khedker

Dept. of Computer Science & Engg.,
Indian Institute of Technology, Bombay
Mumbai, India
uday@cse.iitb.ac.in

Abstract—Modern processors have a small on-chip local memory for instructions. Usually it is in the form of a cache but in some cases it is an addressable memory. In the latter, the user is required to partition and arrange the code such that appropriate fragments are loaded into the memory at appropriate times. We explore automatic partitioning by defining an optimality criterion and provide a lazy algorithm which tries to combine procedures which should be loaded together. The procedures which do not fit into local memory are further partitioned. The lazy nature of the algorithm facilitates using multiple heuristics to identify good partitions. Our partitioner can be used to provide the much needed relief to a programmer and could be an important tool in the design space exploration of embedded processor architectures to study the possibility of replacing expensive cache memory by relatively inexpensive and larger RAM.

Index Terms—Embedded systems, Program partitioning

I. INTRODUCTION

This paper explores automatic program partitioning for embedded processors with limited amount of on-chip instruction memory to initiate work along a direction which has received very little attention so far.

A. Memory and Execution Model

We use a simple memory model consisting of a *local* memory and a *global* memory characterized by :

- *Speed*. Accesses from local memory are much faster than the accesses from global memory since local memory is situated closer to the processor and may use more advanced technology.
- *Size*. The local memory is usually much smaller in size due to the cost of technology.
- *Execution*. Program execution requires instructions to be in the local memory.

This model can be applied at different levels of abstractions and a memory component viewed as a local memory may well be looked upon as global memory at a finer level of abstraction.

Cradle’s CRA 20.03 3SoC (Software Scalable System On a Chip) ([1], [3]) consists of 3 compute quads and 1 I/O quad all on the same chip. Each quad has 32 KB of local instruction memory and a DMA based Memory Transfer Engine (MTE) which facilitates accessing common off-chip SDRAM which could be viewed as the global memory. Interestingly, the local memory can be configured to act as either addressable RAM or as cache. Each compute quad has 4 Processing Engines (PEs) based on RISC cores and 8 Digital Signal Engines (DSEs). Each DSE has 512 bytes of very fast local instruction memory. Similarly, each MTE has a 2 KB of local memory.

Intel’s IXS 1000 Media Signal Processor [2] has 1 Control Processor (CP) core and 4 Digital Signal Processor (DSP) cores on the same chip. Each code has a separate SRAM which can be viewed as local memory. The chip has a common global memory available on the same chip (hence IXS 1000 does not require external memory). For the CP core, the local memory is a cache memory while for the DSP core the local memory is addressable SRAM. ADSP series processors from Analog Devices and Xtensa from Tensilica are other examples of the processors with on-chip local instruction memory.

We define the execution model in terms of *allocation* and *loading*. Allocation refers to deciding the actual load address of instructions in the local memory and is characterized by *granularity*, *time*, and *process* as illustrated in Table I. The instructions may be loaded into the local memory before the execution or during the execution if the size of local memory is small.

B. Static Automatic Partitioning

From Table I it is clear that in general automatic allocation is performed during the execution and static, i.e. pre-execution allocation is almost always manual. Automatic partitioning of assembly programs makes partitioning less error-prone and relieves the programmer from the tedious process of counting memory requirements and inserting instructions to transfer program fragments from global memory to local memory.

If allocation could be performed before execution rather than during execution,

TABLE I
INSTANCES OF MEMORY MODELS AND ASSOCIATED EXECUTION MODELS

Instance of a Memory Model	Associated Execution Model			
	Allocation			Loading
	Granularity	Time	Process	
Virtual Memory System	Page	During Execution	Automatic	During Execution
Overlays	Function	Before Execution	Manual	During Execution
Cache and main memory	Few Instructions	During Execution	Automatic	During Execution
PEs in Cradle's 3SoC	Few Instructions	During Execution	Automatic	During Execution
DSEs in Cradle's 3SoC	512 Bytes	Before Execution	Manual	During Execution
MTEs in Cradle's 3SoC	2 KB	Before Execution	Manual	During Execution
CP in Intels's IXS1000	Few Instructions	During Execution	Automatic	During Execution
DSPs in Intels's IXS1000	Unspecified	Before Execution	Manual	During Execution

- The efficiency of execution can be improved. The dynamic overheads of loading program fragments will be more predictable than cache behaviour since dynamic loading requirements are known statically thanks to the pre-execution allocation.
- For ASIPs (Application Specific Instruction set Processors) such an allocation would provide a useful feedback in design-space exploration since it could discover appropriate local memory sizes for typical applications more precisely. Also, it would facilitate use of simpler and inexpensive addressable memory rather than cache, thereby saving cost and silicon area or allowing larger memory and/or more functionality.

C. Related Work

We have not come across any work which addresses the above kind of automatic partitioning; Typically this partitioning is done manually. The traditional approach of using overlays [15] comes close to our approach. As noted earlier, the granularity of overlays is at the level of a function and they are almost always identified manually. [20] describes a three step method of partitioning sequential programs, which involves determining procedures from sequential programs, preclustering them and then partitioning them using N -way partitioning. We use simpler heuristics for call-graph partitioning, and our problem also includes partitioning individual procedures. Another close variation of our problem of partitioning is [23]. The main focus of their work is on creating "tamper-proof" partitions for the application running on devices like *smart card*. Hence their notion of *good* partition differs from ours, and consequently the approach is also different.

Most of the other works related to partitioning address other issues such as branch alignment [21], partitioning in multiprocessor systems ([7], [22]), data partitioning ([5], [17], [19]), efficient cache management ([9], [13], [14]), run-time memory management ([6], [10], [18]) and code size reduction ([12], [16]).

II. DEFINING PROGRAM PARTITIONING

In this section we describe program representations and define an optimality criterion for partitioning.

A. Program Representations

We use a two-level representation of a program. At the top level, the program is represented by a *Call Graph* (CG) $G_c = (N_c, E_c, \text{main})$ where N_c is the set of nodes representing functions in the program and E_c is the set of edges representing function calls. The execution begins from the `main` function. Note that returns are not represented explicitly. At the lower level, each function is represented by a *Control Flow Graph* (CFG) $G_f = (N_f, E_f, \text{entry})$, where N_f is the set of nodes representing *basic blocks* in the function, E_f is the set of edges representing control transfers between basic blocks, and `entry` is the basic block representing the start of the function.

Each CFG appears as a single node in the CG. The semantics of nodes and edges in a CG is different those in a CFG. Traversal of out edges of a fork node (i.e. a node with multiple successors) is mutually exclusive in a CFG but not in a CG. When a fork node is executed in a CFG, the control is transferred to *only one successor* depending upon the result of execution of the fork node, i.e. control transfers in a CFG are *context-sensitive*. Thus, every graph theoretic path in a CFG constitutes a possible execution path and a CFG describes *multiple context-sensitive* execution paths.

In a CG, however, the control may be transferred to *all successors* of a fork node. Due to a higher level of abstraction, control transfers in a CG appear as *context-insensitive*. Every edge in a CG has an implicit return edge with the control finally returning to `main`. Thus there is a *single context-insensitive* execution path. This path is captured by a variant of depth first traversal of the CG in which nodes are retraversed for both call returns, as well as calls in different call contexts even if they were visited earlier.

In a CG, cyclic paths represent recursive function calls whereas in a CFG, cyclic paths represent iterative control

structures. In presence of cycles, execution paths may be unbounded. Since we need to account for *all* paths, we consider only acyclic paths and the effect of loop nesting is incorporated separately by giving higher weight to edges which appear in a cyclic path. The effect of call returns is incorporated by multiplying the edge weights by 2 in a CG.

B. What is Program Partitioning?

Program partitioning involves the following steps :

- 1) Identify maximal groups of adjacent nodes which should be loaded together in local memory. Group boundaries are indicated by *cut edges*; an edge $e \equiv n \rightarrow s$ is a cut edge if n and s are in different groups.
- 2) Insert code fetch instructions for each cut edge to load the next group in the local memory. Note that a group is always loaded from the start of the local memory.
 - For a cut edge $n \rightarrow s$ in a CG, code fetch instructions are inserted
 - At appropriate place in function n to load the group containing function s .
 - At the end of function s to load the group containing function n .
 - For a cut edge $n \rightarrow s$ in a CFG, code fetch instructions are inserted at appropriate place in basic block n to load the group containing basic block s .

After partitioning, appropriate *relocation* will have to be performed on each group.

C. An Optimality Criterion

Let c_i denote the code size of node $i \in N$ and MAX denote the size of the local memory. Let π be a partition of N consisting of V_1, V_2, \dots, V_m of N such that

$$\sum_{j \in V_i} c_j \leq MAX, 1 \leq i \leq m \quad (1)$$

Each subset represents a group and the group boundaries are characterized by cut edges $e = i \rightarrow j$ such that $i \in V_x$ and $j \in V_y$ such that $x \neq y$.

For a forward edge $e \equiv i \rightarrow j$ in a CFG, let the execution probability of e be denoted by $LocalProb(e)$. In CG, $LocalProb(e) = 1$ for each edge. Let $Paths$ denote the set of all acyclic paths. For each path $\rho \in Paths$, let $PathProb(\rho)$ and $PathWt(\rho)$ denote the *path probability* and the *path weight* respectively.

$$PathProb(\rho) = \prod_{\substack{e \text{ is an} \\ \text{edge in } \rho}} LocalProb(e) \quad (2)$$

$$PathWt(\rho) = \sum_{\substack{e \text{ is an} \\ \text{edge in } \rho}} CutWt(e) \times EdgeWt(e) \quad (3)$$

where, $CutWt(e)$ captures the overhead if an edge is a cut edge, and $EdgeWt(e)$ (Eq. 5) captures the effect of multiple traversals of an edge due to

- loops in CFG and recursive invocations in CG by using $Depth(e)$ and $ExpIter$. $Depth(e)$ is the nesting depth of the innermost loop in which e lies whereas $ExpIter$ is the average number of times a loop (or a recursive call) is expected to be executed.
- multiple (non-recursive) executions through the term $ExpOcc(e)$.

$$CutWt(e) = \begin{cases} 0 & \text{if } e \text{ is not a cut edge} \\ 1 & \text{if } e \text{ is a cut edge, } e \in E_f \\ 2 & \text{if } e \text{ is a cut edge, } e \in E_c \end{cases} \quad (4)$$

$$EdgeWt(e) = ExpOcc(e) * ExpIter^{Depth(e)} \quad (5)$$

$$ExpOcc(e) = \begin{cases} 1 & \text{if } e \in E_f \\ \text{No. of Paths} & \text{if } e \in E_c \\ \text{Ending at } e & \end{cases} \quad (6)$$

The optimality criterion for program partitioning is defined as minimizing the total overhead O_T^{CFG} for CFG and O_T^{CG} for CG as defined below:

$$O_T^{CFG} = \sum_{\rho \in Paths} PathProb(\rho) * PathWt(\rho) \quad (7)$$

$$O_T^{CG} = \sum_{\substack{e \text{ is an} \\ \text{edge in } G}} CutWt(e) * EdgeWt(e) \quad (8)$$

III. CAN PROGRAM PARTITIONING BE MODELED AS GRAPH PARTITIONING?

Graph partitioning is a well researched problem in the class of NP-complete problems with many different formulations and several good heuristic solutions [4]. However, modeling our variant of program partitioning as a graph partitioning problem is not possible due to following reasons: (a) Graph partitioning is formulated as a k -way partitioning problem. i.e. a given graph is to be partitioned into k parts where the predetermined constant k depends on the the application. In our case, k depends upon the length of the program as well as the sizes of basic blocks in the program. Thus the number of partitions is a result of partitioning rather than an input to the partitioner. (b) Most of the graph partitioning formulations impose load balancing constraints on the solution. i.e. graph nodes need to be evenly distributed in k parts. For our program partitioning, load balancing is not needed as long as the optimality criterion is met. Our experiments with Metis [11] indicated that load balancing constraints may produce sub-optimal partition in terms of the criterion (7).

IV. CALL GRAPH PARTITIONING

Partitioning a call graph is quite similar to identifying overlays except that overlays consist of procedures that do not co-exist in the memory whereas partitioning identifies procedures which should co-exists in the memory.

```

1  Partition_CG( $G_c \equiv \langle N_c, E_c, \text{main} \rangle$ )
2  { Compute_Edge_Weights( $G_c$ )
3    for each edge  $e \equiv n \rightarrow s$  in  $E_c$ 
4      Insert  $e$  in  $S_E$  sorted on the following
        primary, secondary, and tertiary keys
        { Dec.  $EdgeWt_e$ , Inc.  $OutDegree(s)$ ,
          Inc.  $OutDegree(n)$  }
5    for each edge  $e \equiv n \rightarrow s$  in  $S_E$ 
6      { if (neither  $n$  nor  $s$  is in a region and
7        ( $size_n + size_s \leq MAX$ )) then
8        Construct a new region with  $n, s$ 
9        else if ( $(n \in R_U, s \in R_V)$  and
10       ( $size_U + size_V \leq MAX$ )) then
11       merge  $U$  and  $V$ 
12       else if ( $(n \in R, s$  not in any region) and
13       ( $size_s + size_R \leq MAX$ )) then
14       include  $s$  in  $R$ 
15       else if ( $(s \in R, n$  not in any region) and
16       ( $size_n + size_R \leq MAX$ )) then
17       include  $n$  in  $R$ 
18     for each  $n$  not in any region
19     { Construct a new region  $R_n$ 
20     if ( $size_n > MAX$ ) then
21       Partition_CFG( $G_n$ )
22     }
23   }
24 }

```

Fig. 1. Partitioning A Call Graph

Procedure *Partition_CG* (Figure 1) tries to minimize the edge cut by prioritizing the edges and then combining the source and target of an edge into a region whenever possible and then progressively combining the regions containing the sources and the targets of the remaining edges. For this purpose edges are sorted on $EdgeWt$ (defined in equation 5) and the degrees of the targets and sources.

Using Degrees of Nodes for Grouping

If there is no caller-callee relationship between two procedures they address in local memory. This is similar to register allocation: If two variables are not live simultaneously, they could be given the same register. Graph coloring based register allocation [8] decides the order of coloring nodes in an interference graph based on the increasing degree of nodes with the minimum degree node being considered first. We view a call-graph as an interference graph and use the degree of nodes as a secondary key for sorting edges.

V. A LAZY ALGORITHM FOR CFG PARTITIONING

At an abstract level, partitioning CFGs and CGs could be viewed as being similar. However, the subtle semantic difference between the edges in CFG and CG facilitate a better criteria for CFG partitioning. As observed in Section II-A, a CG represents a single context-insensitive

```

1  Partition_CFG( $G_f \equiv \langle N_f, E_f, \text{entry} \rangle$ )
2  {  $R_L = \text{Init_CFG}$ ( $G_f$ )
3     $Change = True$ 
4    while ( $Change$ )
5      {  $Change = False$ 
6        for each region  $R$  in  $R_L$ 
7          if ( $\text{min\_include}(R, R_L)$ ) then
8             $Change = True$ 
9          if (partitioning is not over and
10         ( $Change == False$ ))
11         for each region  $R$  in  $R_L$ 
12           if ( $\text{max\_exclude}(R, R_L)$ ) then
13             {  $Change = True$ 
14               break
15             }
16         if (partitioning is not over and
17         ( $Change == False$ ))
18         for each region  $R$  in  $R_L$ 
19           if ( $\text{min\_dfs}/(R, R_L)$ ) then
20             {  $Change = True$ 
21               break
22             }
23         if (partitioning is not over and
24         ( $Change == False$ ))
25         { Find a node  $n$  which does not fit
26           in any neighbouring region
27           Create a new region  $R_n$  for  $n$ 
28            $Change = True$ 
29         }
30       }
31   }
32 }

```

Fig. 2. Partitioning A Control Flow Graph

execution path in which out edges of a fork node are treated equally importantly. A CFG has multiple context-sensitive execution paths in which the probability of execution of a fork node distributes over the successor nodes and hence different out edges should be treated with different importance. This importance is reflected by probabilities and the region formation criteria are made sensitive to this difference.

A. Constructing Initial Regions

Procedure *Partition_CFG* (Figure 2) invokes *Init_CFG* (Figure 3) which identifies the critical edges and creates initial regions by merging the source and target nodes of critical edges. The edge weights computed by procedure *Compute_Edge_Weights* are such that back edges are identified as critical edges by procedure *Find_Critical_Edges*. In *Compute_Edge_Weights*, $nvalue_n$ represents the number of acyclic paths reaching n for CG, and the global execution probability of n for CFG. Similarly, $evalue_e$ represents number of acyclic paths ending at e for CG and global execution probability of e for CFG. In nested loops, the inner back edge is a critical edge. This ensures that the source and target of a back edge forming the innermost loop belong to the

```

1  Init_CFG( $G \equiv \langle N, E, \text{entry} \rangle$ )
2  {  $C_E = \text{Find\_Critical\_Edges}(G)$ 
3    for each edge  $e \equiv n \rightarrow s$  in  $C_E$ 
4      if  $((s \in R)$  and
5         $(size_n + size_R \leq MAX))$  then
6        include  $n$  in  $R$ 
7      else if  $((n \in R)$  and
8         $(size_s + size_R \leq MAX))$  then
9        include  $s$  in  $R$ 
10     else if  $(size_n + size_s \leq MAX)$ 
11       Construct a new region  $R$  with  $n, s$ 
12     return  $R_L$ 
13   }
14  Find_Critical_Edges( $G \equiv \langle N, E, \text{entry} \rangle$ )
15  { Compute_Edge_Weights( $G$ )
16     $C_E = \emptyset$ 
17    for each edge  $e \equiv n \rightarrow s$  in  $E$ 
18      if  $\text{EdgeWt}(e)$  is highest among all
19        edges of  $n$  and  $s$  then
20         $C_E = C_E \cup \{e\}$ 
21    return  $C_E$ 
22  }
23  Compute_Edge_Weights( $G \equiv \langle N, E, \text{entry} \rangle$ )
24  { for each  $n \in N$  in reverse post order
25    { if  $n == \text{entry}$  then  $nvalue_n = 1$ 
26      else  $nvalue_n = 0$ 
27      for each  $p \in \text{pred}(n)$  such that
28         $p \rightarrow n$  is not a back edge
29         $nvalue_n = nvalue_n + value_{p \rightarrow n}$ 
30      for each  $s \in \text{succ}(n)$ 
31        { if  $n \rightarrow s$  is a back edge then
32           $value_{n \rightarrow s} = 1$ 
33        else
34           $value_{n \rightarrow s} = nvalue_n * \text{LocalProb}_{n \rightarrow s}$ 
35           $\text{EdgeWt}_{n \rightarrow s} = value_{n \rightarrow s} * \text{ExpIter}^{\text{Depth}_{n \rightarrow s}}$ 
36        }
37    }
38  }

```

Fig. 3. Constructing Initial Regions in CFG

same region. Thus, a loop is isolated from the rest of the graph and if a loop can fit into the local memory, it is put in the same region by the algorithm.

B. Region Expansion

Initial regions are then expanded in a lazy manner by applying the following three heuristics (Figure 4) in the order of decreasing preference. In case of a tie, the heuristics defer the decisions as much as possible.

Expansion based on minimum inclusion overhead: All neighbours of a region R are examined to find a neighbour r which gives the minimum edge cut for the combined region. This edge cut is computed from all in and out edges of the combined region. If there is a clear winner, the region is expanded and the heuristic is applied recursively to the expanded region. If there is no

```

1  min_include( $R, R_L$ )
2  { if  $R$  has a single neighbour  $r$  such that
3     $(size_r + size_R \leq MAX)$  then
4    return False
5  else for each neighbour  $r$  of  $R$  where
6     $(size_r + size_R \leq MAX)$ 
7     $\text{Overhead}_r = \sum \text{EdgeWt}_e$  where  $e$  is
8    an in-edge or out-edge of  $R + r$ 
9    if there is a unique min  $\text{Overhead}_r$  then
10   { Include  $r$  in  $R$ 
11     if  $(r$  is a region) then
12       remove  $r$  from  $R_L$ 
13     min_include( $R, R_L$ )
14   }
15   return True
16 }
17 else return False
18 }
19 max_exclude( $R, R_L$ )
20 { if  $R$  has a single neighbour  $r$  such that
21    $(size_r + size_R \leq MAX)$  then
22   return False
23 else for each neighbour  $r$  of  $R$  where
24    $(size_r + size_R \leq MAX)$  then
25    $\text{Overhead}_r = \sum \text{EdgeWt}_e$  where  $e$  is
26   an edge between  $R$  and  $r$ 
27   if there is a unique max  $\text{Overhead}_r$  then
28   { Include  $r$  in  $R$ 
29     if  $(r$  is a region) then
30       remove  $r$  from  $R_L$ 
31     return True
32   }
33   else return False
34 }

```

Fig. 4. Region Expansion Heuristics

clear winner, a different region R' is considered until all regions are exhausted.

Expansion based on maximum exclusion overhead: If no region can be expanded by the first heuristic, this heuristic examines all neighbours of a region R to find a neighbour r which, if kept out of R , implies maximum edge cut (computed from all in/out edges of neighbour r). If there is a clear winner, the region is expanded and the first heuristic is tried again. Thus, this heuristic works only as a tie breaker and is not applied repeatedly unlike the first heuristic.

Expansion based on minimum DFS number: If the second heuristic also results in a tie, a region is expanded based on DFS numbering. This is quite arbitrary and is used only as a tie breaker because all alternatives are equally good.

If none of the above heuristics breaks the tie, then a node which does not fit in any neighbouring region is converted into a region and the entire process is repeated. The algorithm is lazy in that it defers the merging decision until a clear choice emerges or until

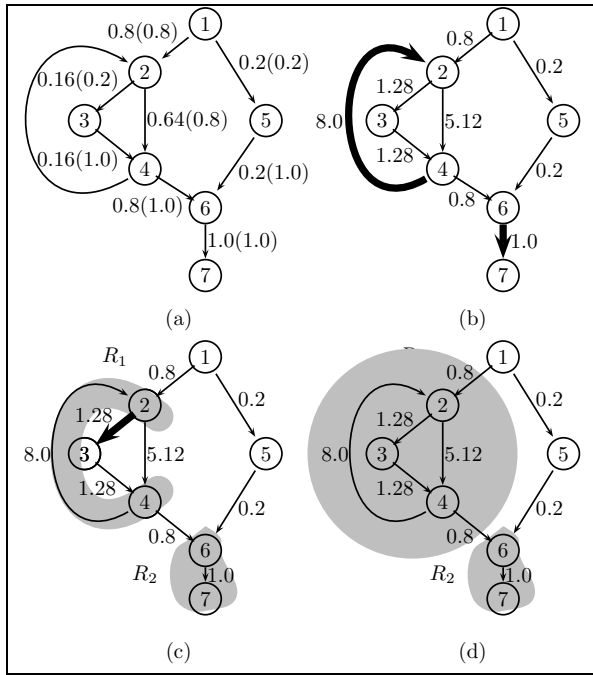


Fig. 5. CFG partitioning example

all alternatives are exhausted.

C. Examples of CFG Partitioning

Consider a CFG in Figure 5(a). The edges are annotated with the global and local edge probabilities as *global(local)*. The code size of each node is 1 KB. If we assume that a loop is expected to be executed 8 times, the global probabilities of edges $2 \rightarrow 3$ and $2 \rightarrow 4$ get multiplied by 8 and they have been labeled with the resulting edge weights in (b). Also, the edge weight of the back edge is computed as 8. the two critical edges $4 \rightarrow 2$ and $6 \rightarrow 7$ resulting in initial regions R_1 and R_2 . In the next step, R_1 is expanded to include node 3. Now either node 1 or region R_3 could be included in R_1 . Their minimum inclusion overheads as well as maximum exclusion overheads are identical (1.0). Hence using the minimum DFS number, node 1 is included in R_1 . Since MAX is 5 KB, this allows even node 5 to be included in R_1 and the total path overhead is 1.0.

D. Lazy Vs. Greedy Partitioning

From the above examples, it might appear that using a greedy algorithm which traverses the highest probability path to form regions might produce similar partitioning. However, this is fraught with the following dangers: Such an algorithm might minimize the overheads along the highest probability path but the edge cuts so reduced get distributed over other paths and the overall path overhead may actually increase. The example programs on which we have tested this algorithm include functions which have over a million paths. In such cases, the

probability distribution over all paths is not very discriminatory and it is preferable to minimize the overall path overhead rather than path overhead for a single path. A greedy algorithm might combine an initial region containing a back edge with nodes/regions outside of the loop. This may prevent the entire loop getting covered in a single region even if it is possible.

VI. EXPERIMENTAL RESULTS

This algorithm has been implemented for partitioning programs for PEs in Cradle’s 3SoC. We have compiled our test programs using `gcc` for Cradle’s PE. The generated assembly language programs are processed to construct a CG for every package and a CFG for every function in the package. For the time being we have ignored the standard library functions. Partitioning is performed for 3 sizes of MAX : 2 KB, 3 KB, and 4 KB. Basic blocks larger than MAX are divided into smaller blocks to fit the local memory and CFG is appropriately modified. Local probabilities required for out edges of fork nodes in a CFG are provided “intuitively” for the current experimentation. The partitioner identifies the cut edges in the programs and generates the statistics.

Table II provides the data for the CGs of the following benchmarks: `ezw`¹, `gsm`², `kexis`³, `arith`⁴. The total edge cut for a program does not exceed 2% of the code size. This figure includes the number of code fetch instructions multiplied by the number of times they are expected to be executed. Thus the actual code expansion is smaller than 2% and the run time overheads of dynamic loading are expected to be well within the acceptable range of 10%. Further experimentation is

¹EZW still image coder: <http://pesona.mmu.edu.my/~msgng/ezw.html>

²GSM speech codec:

<ftp://ftp.cs.cmu.edu/project/fgdata/speech-compression/GSM/>

³Lossless audio file compressor Kexis:

<http://sourceforge.net/projects/kexis/>

⁴Compression using arithmetic coding:

http://www.cs.mu.oz.au/~alastair/arith_coder/

TABLE II
EMPIRICAL MEASUREMENTS ON CALL GRAPHS

Package	Size (Bytes)	#F	#E	FS_{max} (Bytes)	MAX (Bytes)	Edge Cut
arith	36942	62	217	3647	2000	702
					3000	638
					4000	594
ezw	71672	198	607	4399	2000	1432
					3000	1414
					4000	1364
gsm	60146	94	212	4577	2000	384
					3000	366
					4000	356
kexis	15089	27	107	1845	2000	280
					3000	272
					4000	264

TABLE III
EMPIRICAL MEASUREMENTS FOR CFGS

Function	Size	$\#N_f$	$\#E_f$	MAX	$\#CE$	EV
arithmetic_ decode_ target	2301	73	107	2000	24	586
arithmetic_ encode	2705	66	93	2000	7	122
binary_ arithmetic_ encode	2647	69	97	2000	5	129
decode	2175	76	109	2000	10	127
gsm_decode	3489	5	5	2000	2	10
				3000	2	10
				4000	2	10
gsm_explode	4163	6	6	2000	3	10
				3000	2	10
				4000	2	10
RescaleImage	4399	93	128	2000	8	62
				3000	4	84
				4000	4	84

required to configure the PEs in 3SoC to use local addressable memory for these ranges of MAX and compare the performance of the programs with the PEs using comparable sizes of cache.

Table III provides the results of partitioning the functions which are larger than MAX. It can be seen that the number of cut edges is quite small and the code expansion would be around 2%. The number of paths in most of these functions is well over a thousand (and in some cases, well over a million). Hence the probability component of the overhead reduces significantly and as such we have listed only the number of edges which are cut rather than the overheads due to these cuts. Even in these cases, we expect the run time overheads to be well within the acceptable limits of 10%. It should be noted that our partitioner keeps an entire loop within the same region unless the size of the loop body exceeds MAX; in such a case even manual partitioning cannot do better unless the code is rewritten by changing the design.

VII. CONCLUSIONS AND FUTURE WORK

This paper initiates an important direction of work which does not seem to have received much attention in past. We have proposed a static program partitioning algorithm which works at the levels of functions as well as basic blocks. This approach has the potential to relieve the programmer from manual partitioning in case of addressable local memory. It could provide useful inputs in architecture design exploration. Further work requires empirical measurements to validate the above.

REFERENCES

[1] CRA 20.03 Data Sheet, 2002. Cradle Technologies. Available from www.cradle.com.

[2] Optimized High-Density Voice over Packet (VoP) Architecture for Next Generation Networks, 2002. Intel White Paper. Available from www.intel.com.

[3] The Software Scalable System on a Chip (3SoC) architecture, 2002. Cradle White Paper. Available from www.cradle.com.

[4] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning: A survey. *Integration: the VLSI journal*, 19(1-2):1-81, 1995.

[5] Sundaram Anantharaman and Santosh Pande. An efficient data partitioning method for limited memory embedded systems. In *LCDES'98*, 1998.

[6] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM TECS*, 1(1):6-26, 2002.

[7] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 99-108, 2002.

[8] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *PLDI'89*, pages 275-284, 1989.

[9] J. S. Hu, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, H. Saputra, and W. Zhang. Compiler-directed cache polymorphism. In *LCDES'02*, pages 165-174, 2002.

[10] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *38th DAC*, pages 690-695, 2001.

[11] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, 1998.

[12] Dae-Hwan Kim and Hyuk Jae Lee. Iterative procedural abstraction for code size reduction. In *CASES*, pages 277-279, 2002.

[13] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. In *10th ASPLOS*, pages 159-170, 2002.

[14] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A large, fast instruction window for tolerating cache misses. In *29th ISCA*, pages 59-70, 2002.

[15] Noam Levine. Strategies for program code and data overlays in DSP systems. In *ICSPAT*, 1996.

[16] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *LCDES'02*, pages 120-129, 2002.

[17] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *TODAES*, 5(3):682-704, 2000.

[18] Mohamed Shalan and Vincent J. Mooney. A dynamic memory management unit for embedded real-time system-on-a-chip. In *CASES*, pages 180-186, 2000.

[19] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE*, March 2002.

[20] Frank Vahid. Partitioning sequential programs for cad using a three-step approach. *ACM Trans. Des. Autom. Electron. Syst.*, 7(3):413-429, 2002.

[21] Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Improving performance by branch reordering. In *PLDI '98*, pages 130-141, New York, USA, 1998. ACM Press.

[22] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM TOCS*, 20(3):283-328, 2002.

[23] Tao Zhang, Santosh Pande, Andre dos Santos, and Franz Josef Bruecklmayr. Leakage-proof program partitioning. In *CASES*, pages 136-145, 2002.