Static Program Partitioning for Limited Memory Embedded Processors

Bageshri Sathe
bageshri@cse.iitb.ac.in
Department of Computer Science & Engg., IIT Bombay.

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 addressable RAM. Increased predictability of execution times is another advantage of the latter.

1 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.

1.1 Memory 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 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.

1.2 Execution Model

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 1. The instructions may be loaded into the local memory before the execution or during the execution if the size of local memory is small.

### 1.3 Why Static Partitioning?

From Table 1 it is clear that in general automatic allocation is performed during the execution and pre-execution allocation is almost always manual. Manual allocation requires programmers to partition the programs by counting memory requirements and by inserting instructions to transfer program fragments from global memory into local memory. Clearly, this is a tedious, laborious, and error-prone process.

If automatic allocation could be performed before execution rather than 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.

### 1.4 Related Work

We have not come across any work which addresses the above kind of partitioning and the state of art indicates that this partitioning is currently done manually. However, with the increasing use of language processing tools such as compilers, programmer has less control over the generated object code. Our work presents a simple heuristic for appropriately partitioning programs.

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. In terms of automatic allocation, the work that addresses the closest variation of our problem of partitioning is [24]. Although there is a similarity in the concept of partitioning—[24] also describes partitioning as dividing the program into smaller fragments and loading them on demand—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.

[22] present a branch alignment algorithm to reduce the pipeline penalties occurring due to control transfer instructions. The problem of achieving near optimal branch alignment does not include considering the memory size constraints when determining the program layout. Also [22] consider only the intraprocedural branch alignments.

Most of the other works related to partitioning address other issues. Partitioning of programs in multiprocessor systems for assigning different segments to different processors is discussed in [7, 23]. [21]
describes partitioning of large programs written in C
or VHDL into hardware modules, hardware and soft-
ware processors in order to improve performance.
Data partitioning ([5, 17, 20]) addresses the prob-
lem of distributing the application data into various
memory modules such that the most frequently ac-
assed data is readily available. Several techniques
([9, 12, 14, 19]) have been suggested to exploit the
program/data cache in most profitable way. Run-
time memory management has been discussed in
context of data memory in [6, 18, 10]. Code size
reduction is discussed in [13, 16, 25]

2 Defining Program Partitioning

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

2.1 Program Representations

We use a two-level representation of programs. 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 \( \text{entry} \) 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 ex-
clusive 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 exe-
cution of the fork node, i.e. control transfers in a
CFG are context-sensitive. Thus, every graph theo-
retic 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 (in the order of the
call statements in the 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
re-traversed for both call returns, as well as calls in
different call contexts even if they were visited ear-
lier.

In a CG, cyclic paths represent recursive function
calls whereas in a CFG, cyclic paths represent iter-
ative control structures. In presence of cycles, exe-
cution 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 sepa-
ately by giving higher weight to edges which appear
in a cyclic path. The effect of call returns is incorpo-
rated by multiplying the edge weights by 2 in a CG.

2.2 What is Program Partitioning?

Program partitioning involves the following steps:

1. Identify maximal groups of adjacent nodes
   which should be loaded together in local mem-
   ory. 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 appropri-
       ate 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.

2.3 An Optimality Criterion

Let \( c_i \) denote the code size of node \( i \in N \) and \( \text{MAX} \)
denote the size of the local memory. Let \( \pi \) be a par-
tition of \( N \) consisting of \( V_1, V_2, \ldots, V_m \) such that

\[
\sum_{j \in V_i} c_j \leq \text{MAX}, \ 1 \leq i \leq m \tag{1}
\]
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 $e = i \rightarrow j$ 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 $p \in \text{Paths}$, let $PathProb(p)$ and $PathWt(p)$ denote the path probability and the path weight respectively.

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

$$PathWt(p) = \sum_{e \text{ is an edge in } p} CutWt(e) \times EdgeWt(e)$$ (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 $ExpOcc(e)$.

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

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

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

$$O_T^{CFG} = \sum_{p \in \text{Paths}} PathProb(p) \times PathWt(p)$$ (7)

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

3 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 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 where 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 that load balancing constraints may produce sub-optimal partition in terms of the criterion (7).

4 The Proposed Algorithm

The optimality criteria defined in section 2.3 cover both a call graphs and control flow graphs for functions in a call graph. The granularity of top level partitioning is a procedure while the granularity of lower level of partitioning is a basic block. Thus, we first partition the call graph. If the size of a procedure exceeds the size of local memory, the control flow graph of such a procedure is partitioned.

4.1 Partitioning a Call Graph

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.

Procedure $\text{Partition}_{CG}$ (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 could be loaded at the same address...
in the local memory. This is similar to register allocation: If two variables are not live simultaneously, they could be given the same register. Thus, a call graph could also be viewed as an interference graph used for graph colouring based register allocation.

The state of art register allocators use the graph coloring approach in [8] which decides the order of coloring nodes in an interference graph based on the increasing degree of of nodes with the minimum degree node being considered first. We use the same heuristic as a secondary key for sorting the edges.

4.2 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 2.1, a CG represents a single context-insensitive 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 is made sensitive to this difference.

4.2.1 Constructing Initial Regions

Procedure Partition_CFG (Figure 2) invokes Init_CFG (Figure 6) 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 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 same region. Thus, a loop is isolated from the rest of the graph and if a loop can fit into the local memory, our algorithm will always put the entire loop in the same region.

4.2.2 Region Expansion

Initial regions are then expanded in a lazy manner by applying the following three heuristics (Figure 8) in the order of decreasing preference. In case of a tie, the heuristics defer the decisions as much as possible.

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

2. **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.

3. **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 all alternatives are exhausted.

### 4.2.3 Examples of CFG Partitioning

We explain CFG partitioning with the help of three examples. In each example, the original CFG is annotated with the local edge probabilities in parenthesis and the global probabilities outside the parenthesis. The code size of each node is 1 KB. For the first example in Figure 3 we assume the size of the local memory ($MAX$) to be 4 KB. $Init\_CFG$ identifies edges $1 \rightarrow 3$ and $4 \rightarrow 5$ as critical edges and creates regions $R_1$ and $R_2$ around them. When applying the minimum inclusion crite-
For the next example (Figure 4), let the total memory size be 5 KB. *Init_CFG* identifies edges 1 → 2 and 6 → 7 as critical edges and constructs regions $R_1$ and $R_2$. When $R_1$ is considered for expansions, it has three neighbours: nodes 3, 4, and 5. If node 3 is included, edges 3 → 4, 2 → 4 and 1 → 5 are cut and the inclusion overhead is 1.0. If node 4 is included, edges 2 → 3, 3 → 4, 4 → 6 and 1 → 5 are cut and the inclusion overhead is 1.32. If node 5 is included, edges 2 → 3, 2 → 4, and 5 → 6 are cut and the inclusion overhead is 1.0. Thus there is a tie between node 3 and node 5. Hence we defer the decision of merging and try to find out which node is costlier to be kept out. The exclusion overhead for node 3 is .32, for node 4 it is 1.6 while for node 5 it is 0.4. Thus we include node 4 in $R_1$. Then $R_1$ is further expanded to include $R_2$.

Our final example (Figure 5) is a modified version of the previous example. Here a back edge from node 4 to node 2 has been added. If we assume that a loop is expected to be executed 8 times, the global probabilities of edges 2 → 3 and 2 → 4 get multiplied by 8 and they have been labeled with the the resulting edge weights in the following CFG. Also, the edge weight of the back edge is computed as 8. Now the two critical edges are: 4 → 2 and 6 → 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.

4.2.4 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 back edge is identified as a critical edge and an initial region consisting of the source and the target of a back edge is created. A greedy algorithm might include this initial region into an outer region without giving a chance to the loop region to include other nodes in the loop thereby dividing the nodes in a loop into different regions. This increases the overhead significantly.

5 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 worrying about partitioning in the case of addressable local memory. For the architectures intending to use cache, this approach could provide useful inputs in architecture design exploration or increased predictability of execution times. Further work requires extensive empirical measurements to validate the above.

References


1. \textit{Init}_CFG(G \equiv (N, E, \text{entry}))
2. \{ 
3. \quad C_E = \text{Find}_\text{Critical}_\text{Edges}(G) 
4. \quad \text{for each edge } e \equiv n \rightarrow s \text{ in } C_E 
5. \quad \quad \text{if } (s \text{ is in a region } R \text{ and } (\text{size}_n + \text{size}_R \leq \text{MAX})) \text{ then include } n \text{ in } R 
6. \quad \quad \text{else if } (n \text{ is in a region } R \text{ and } (\text{size}_s + \text{size}_R \leq \text{MAX})) \text{ then include } s \text{ in } R 
7. \quad \quad \text{else if } (\text{size}_n + \text{size}_s \leq \text{MAX}) 
8. \quad \quad \quad \text{Construct a new region } R \text{ consisting of } n \text{ and } s \text{ and add } R \text{ to the list of regions } R_L 
9. \quad \text{return } R_L 
10. \} 
11. \text{Find}_\text{Critical}_\text{Edges}(G \equiv (N, E, \text{entry}))
12. \{ 
13. \quad \text{Compute}_\text{Edge}_\text{Weights}(G) 
14. \quad C_E = \emptyset 
15. \quad \text{for each edge } e \equiv n \rightarrow s \text{ in } E 
16. \quad \quad \text{if } e \text{ is the edge with the highest } \text{EdgeWt} \text{ among all edges of } n \text{ and } s \text{ then } C_E = C_E \cup \{e\} 
17. \quad \text{return } C_E 
18. \} 

Figure 6: Constructing the Initial Regions in A Control Flow Graph


1. `Compute_Edge_Weights(G \equiv (N,E,entry))`
2. /* nodevalue\_n represents the number of (acyclic) paths reaching \( n \) for CG */
3. /* and the global execution probability of \( n \) for CFG. Similarly, edgevalue\_e */
4. /* represents the number of acyclic paths ending at \( e \) for CG and the */
5. /* global execution probability of \( e \) for CFG. */
6. { for each node \( n \in N \) /* Visited in reverse post order */
7.   { if \( n == \text{entry} \) then nodevalue\_n = 1
8.   else nodevalue\_n = 0
9.     for each \( p \in \text{pred}(n) \) such that \( p \rightarrow n \) is not a back edge
10.        nodevalue\_n = nodevalue\_n + edgevalue\_p\rightarrow n
11.     for each \( s \in \text{succ}(n) \)
12.       { if \( n \rightarrow s \) is a back edge then edgevalue\_n\rightarrow s = 1
13.         else edgevalue\_n\rightarrow s = nodevalue\_n \times \text{LocalProb}_{n\rightarrow s} /* \text{LocalProb}_e \) is 1 for CG. */
14.         EdgeWt\_n\rightarrow s = edgevalue\_n\rightarrow s \times \text{ExpIterDepth}_{n\rightarrow s}
15.       }
16.   }
17. }

Figure 7: Computing Edge Weights

1. `Expand_R_With_Min_Inclusion_Overhead(R,R_L)`
2. { if \( R \) has only one neighbour \( r \) such that \( \text{size}_r + \text{size}_R \leq \text{MAX} \) then return False
3.   else for each neighbour \( r \) of \( R \) such that \( \text{size}_r + \text{size}_R \leq \text{MAX} \)
4.     Overhead\_r = \sum EdgeWt\_e \text{ where } e \text{ is an in edge or an out edge of } R \text{ with } r \text{ included}
5.     if there is a unique minimum Overhead\_r then
6.       { Include \( r \) in \( R \)
7.         if \( (r \) is a region) then remove \( r \) from \( R_L \)
8.         \text{Expand}_R_With_Min_Inclusion_Overhead(R,R_L)
9.       return True
10.   }
11. else return False /* Defer the inclusion decision */
12. }
13. `Expand_R_With_Max_Exclusion_Overhead(R,R_L)`
14. { if \( R \) has only one neighbour \( r \) such that \( \text{size}_r + \text{size}_R \leq \text{MAX} \) then return False
15.   else for each neighbour \( r \) of \( R \) such that \( \text{size}_r + \text{size}_R \leq \text{MAX} \) then
16.     Overhead\_r = \sum EdgeWt\_e \text{ where } e \text{ is an edge between } R \text{ and } r
17.     if there is a unique maximum Overhead\_r then
18.       { Include \( r \) in \( R \)
19.         if \( (r \) is a region) then remove \( r \) from \( R_L \)
20.       return True
21.     }
22. else return False /* Defer the inclusion decision */
23. }

Figure 8: Region Expansion Heuristics