PERFORMANCE OF ON-CHIP MEMORY ARCHITECTURE EXPLORATION OF EMBEDDED SYSTEM ALGORITHM

G Ravi Kumar and S Lakshmi

*Corresponding Author: G Ravi Kumar, ravikumar.g034@gmail.com

Today’s feature-rich multimedia products require embedded system solution with complex System-on-Chip (SoC) to meet market expectations of high performance at low cost and lower energy consumption. SoCs are complex designs with multiple embedded processors, memory subsystems, and application specific peripherals. The memory architecture of embedded SoCs strongly influences the area, power and performance of the entire system. Further, the memory subsystem constitutes a major part (typically up to 70%) of the silicon area for the current day SoC. In this work, we proposed an automated framework for on-chip memory architecture exploration. Our proposed framework integrates memory architecture exploration and data layout to search the design space efficiently. While the memory exploration selects specific memory architectures, the data layout efficiently maps the given application on to the memory architecture under consideration and thus helps in evaluating the memory architecture. The proposed memory exploration framework works at both logical and physical memory architecture level. Our work addresses on-chip memory architecture for DSP processors that is organized as multiple memory banks, with each back can be a single/dual port banks and with non-uniform bank sizes. Further, our work also address memory architecture exploration for on-chip memory architectures that is SPRAM and cache based. Our proposed method is based on multi-objective Genetic Algorithm based and outputs several hundred Pareto-optimal design solutions that are interesting from area, power and performance viewpoints within a few hours of running on a standard desktop configuration

Keywords: SoC, On-Chip Memory Organizer, DSP Processor and Multiple memory banks

INTRODUCTION

Logical Memory Exploration
In this we will focus on memory architecture exploration for a given application in order to obtain memory architecture performance (reduced memory stalls) and memory area. Embedded systems are application specific and hence embedded designers

This article can be downloaded from http://www.ijerst.com/currentissue.php
study the target application to understand the memory architecture requirements. DSP applications are typically data intensive and require very high memory bandwidth to meet real-time requirements. There are two steps to designing an optimal memory architecture for a given application. The first step is to find the right memory architecture parameters that are important for improving target application's performance and the second step is to optimally map the given application on to the memory architecture under consideration. This leads to a two-level optimization problem with multiple objectives. At the first level, an appropriate memory architecture must be chosen which includes determining the number and size of each memory bank, the number of memory ports per bank, the types of memory (scratch pad RAM or cache).

**METHOD OVERVIEW**

**Memory Architecture Parameters**

As discussed in section 2.1.1, the memory architecture of a DSP processor has to support a high bandwidth to satisfy the needs of data memory intensive DSP applications. As shown in the Figure 4.1, the memory architecture of a DSP processor is organized as multiple memory banks, where each bank can be accessed independently to enable parallel accesses. In addition each of the bank can be a single port or a dual port memory. For now we assume that the memory banks with single ports have the same size and similarly the memory banks with dual-ports have the same size. Also, at this point, we only consider a logical view of the memory architecture. How the different (logical) memory banks are realized using different physical memories from a given ASIC design database, and how they impact the power, performance and cost of the memory architecture will be discussed in the next chapter. Choosing the appropriate physical memory architecture is a design space exploration process. We use the terms logical memory exploration and physical memory exploration to clearly distinguish between the two.

**Memory Architecture Parameters**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_p$</td>
<td>Single-port memory bank</td>
</tr>
<tr>
<td>$D_p$</td>
<td>Dual-port memory bank</td>
</tr>
<tr>
<td>$N_s$</td>
<td>Number of SARAM banks</td>
</tr>
<tr>
<td>$B_s$</td>
<td>SARAM bank size</td>
</tr>
<tr>
<td>$N_d$</td>
<td>Number of DARAM banks</td>
</tr>
<tr>
<td>$B_d$</td>
<td>DARAM bank size</td>
</tr>
<tr>
<td>$E_s$</td>
<td>External Memory size</td>
</tr>
<tr>
<td>$D_s$</td>
<td>Total data size</td>
</tr>
<tr>
<td>$W_s$</td>
<td>Normalized weight for SARAM</td>
</tr>
<tr>
<td>$W_d$</td>
<td>Normalized weight for DARAM</td>
</tr>
<tr>
<td>$W_e$</td>
<td>Normalized weight for external memory</td>
</tr>
</tbody>
</table>

**Genetic Algorithm Formulation**

Genetic Algorithms (GAs) have been used to solve hard optimization problems. Genetic algorithms simulate the natural process of evolution using genetic operators such as, natural selection, survival of the fittest, mutation and crossover in order to search the solution space.

To map an optimization problem to the GA framework, we need the following:
chromosomal representation, fitness
computation, selection function, genetic
operators, the creation of the initial population
and the termination criteria.

For the memory layout problem, each
individual chromosome should represent a
memory placement. A chromosome is a vector
of \( d \) elements, where \( d \) is the number of data
sections. Each element of a chromosome can
take a value in \((0..m)\), where \(1..m\) represent
on-chip memory banks (including both
SARAM and DARAM memory banks) and 0
represents off-chip memory. Thus if the
element \( i \) of a chromosome has a value \( k \), then
the \( j \)th data section is placed in memory bank \( k \). Thus a chromosome represents a memory
placement for all data sections. Note that a
chromosome may not always represent a valid
memory placement, as the size of data
sections placed in a memory bank \( k \) may
exceed the size of \( k \). Thus the genetic algorithm
should consider only valid chromosomes for
evolution. This is achieved by giving a low
fitness value for invalid chromosomes. Our
initial experiments demonstrated that the
above chromosome representation (vector of
decimal numbers) is more effective than the
conventional bit vector representation [30] as
the latter will lead to assignment of non-existent
memory banks when the number of memory
banks is not a power of 2.

Genetic operators provide the basic search
mechanism by creating new solutions based
on the solutions that exist. The selection of the
individuals to produce successive generation
plays an extremely important role. The selection
approach assigns a probability of selection to
each individual, depending on its fitness. An
individual with a higher fitness has a higher
probability of contributing one or more
offspring to the next generation. In the selection
process a given individual can be chosen more
than once. Let us denote the size of the
population (number of individuals) as \( P \).
Reproduction is the operation of producing
offspring for the next generation. This is an
iterative process. In every generation, from the
\( P \) individuals of the current generation, \( M \) more
offspring are generated. This results in a total
population of \( P + M \). From this total population
of \( P + M \), \( P \) fittest individuals survive to the next
generation. The remaining \( M \) individuals are
annihilated. In our data layout problem, for each
of the individuals, the fitness function computes
the number of resulting memory conflicts. Since
GAs typically solve a maximization problem,
we change our problem as a maximization
problem by negation and normalization. Recall
that a chromosome may represent an invalid
solution. To discourage invalid individuals, we
associate a very low fitness value to them.

**Genetic Algorithm Formulation for
Memory Architecture Exploration**

To map an optimization problem to the GA
framework, we need the following:
chromosomal representation, fitness
computation, selection function, genetic
operators, the creation of the initial population
and the termination criteria.

For the memory exploration problem, each
individual chromosome represents a memory
architecture. A chromosome is a vector of 5
elements: \( (N_s, B_s, N_{dp}, B_{dp}, E_s) \).

Fitness function computes the fitness for
each of the individual chromosomes. For the
memory exploration problem there are two
objectives Memory cost \( (M_{\text{cost}}) \) and Mem-
Cycles ($M_{cyc}$). For each of the individuals, the fitness function computes $M_{cost}$ and $M_{cyc}$. The memory cost ($M_{cost}$) is computed from equation (4.1) based on the memory architecture parameters ($N_s, B_s, N_d, B_d, E_d$) that defines each of the chromosome.

The memory stall cycles ($M_{cyc}$) is obtained from the data-layout that maps the application data buffers on to the given memory architecture, defined by the chromosome's parameters, with the objective to reduce memory stall cycles. We use the greedy backtracking heuristic algorithm described in Section 3.5 for the data layout. The data layout algorithm estimates the memory stall cycles after placing the application data buffers in the given memory architecture.

Once $M_{cyc}$ and $M_{cost}$ are computed for all the individuals in the population, the individuals need to be ranked. The non-dominated points at the end of the evolution represent a set of solutions which provide interesting trade-offs in terms of one of the objectives in order to annihilate the chromosomes that has a lower fitness. For a single objective optimization problem, the ranking process is straightforward and is proportional to the objective. But for multi-objective optimization problem, the ranking needs to be computed based on all the objectives. We describe how we do this in the following subsection.

**Pareto Optimality and Non-Dominated Sorting**

First we define Pareto optimality. Let ($M^a_{cost}, M^a_{cyc}$) be the memory cost and memory cycles of chromosome A and ($M^b_{cost}, M^b_{cyc}$) be the memory cost and memory cycles B then, A dominates B if the following expression is true.

\[(M^a_{cost} < M^b_{cost}) \land (M^a_{cyc} < M^b_{cyc})\]

\[\lor ((M^a_{cyc} < M^b_{cyc}) \land (M^a_{cost} < M^b_{cost})) \ldots (4.2)\]

The non-dominated points at a given generation are those which are not dominated by any other design points seen so far.

We use the non-dominated sorting process described in [25] for ranking the chromosomes based on the $M_{cyc}$ and $M_{cost}$. The ranking process in multi-objective GA proceeds as follows. All non-dominated individuals in the current population are identified and flagged. These are the best individuals and assigned a rank of 1. These points are then removed from the population and the next set of non-dominated individuals are identified and ranked 2. This process continues until the entire population is ranked. Fitness values are assigned based on the ranks. Higher fitness values are assigned for rank-1 individuals as compared to rank-2 and so on. This fitness is used for the selection probability. The individuals with higher fitness gets a better chance of getting selected for reproduction. Mutation and cross over operations are used to generate offspring’s. These operators are defined in a similar way as in Section 3.4.

One of the common problems in multi-objective optimization is solution diversity. Basically the search path may progress towards only certain objectives resulting in design points favoring those objectives. Hence, solution diversity is very critical in order to get a good distribution of solutions in the Pareto-optimal front. To maintain solution diversity, the fitness value for solution that are in the same neighborhood are given lower values.

This article can be downloaded from http://www.ijerst.com/currentissue.php
Following steps explain the fitness assignment method among the individuals in the same rank. To begin with, all solutions with rank $= 1$ are assigned equal fitness. This becomes the maximum fitness that any solution can have in the population. Given a set of $n_k$ solutions in the $k$-th rank each having a fitness value $f_k$, the following steps reassign the fitness values for solutions based on the number and proximity of neighboring solutions (this is known as the \textit{niche count}).

\textbf{Step 1:} The normalized Euclidean distance for solution $i$ with solution $j$ for all $n_k$

\[ d_{ij} = \frac{M^i_{\text{cost}} - M^j_{\text{cost}}}{M^i_{\text{cost}} - M^j_{\text{cost}}} + \frac{M^{\text{cyc}}_i - M^{\text{cyc}}_j}{M^{\text{cyc}}_i - M^{\text{cyc}}_j} \]  

where $M^i_{\text{cost}}$ and $M^j_{\text{cost}}$ represent respectively the highest and lowest values for $M_{\text{cost}}$ seen across all solutions; similarly $M^{\text{cyc}}_i$ and $M^{\text{cyc}}_j$ denote the highest and lowest number of stall cycles.

Step 2: The distance $d_{ij}$ is compared with a pre-defined parameter $\sigma_{\text{share}}$ and the following sharing function value is computed [25]:

\[ Sh(d_{ij}) = \begin{cases} 1 - \frac{d_{ij}}{\sigma_{\text{share}}} & \text{if } d_{ij} < \sigma_{\text{share}} \\ 0, & \text{otherwise.} \end{cases} \]

Step 3: Calculate niche count for the $i$-th solution in rank $k$ as follows:

\[ m_i = \sum_{j=1}^{n_k} Sh(d_{ij}) \]

Step 4: Reduce the fitness $f_k$ of $i$-th solution in the $k$-th rank as:

\[ f^i_k = \frac{f_k}{m_i} \]

The above steps have to be repeated for all the ranks. Note that the niche count ($m$) will be greater than one for solution $i$ that has many neighboring points. For a lone distant point the niche count will be approximately 1. Thus greater fitness values are assigned to points that do not have many close neighboring solutions, encouraging them to get selected for reproduction. Once all the $n_k$ individuals in rank $k$ are assigned fitness based on the above steps, the minimum of fitness is taken as the starting fitness and assigned to all the individuals in rank $k + 1$.

After some experimentation we fixed the $\sigma_{\text{share}}$ as 0.6 as the initial value and decrease the $\sigma_{\text{share}}$ up to 0.25 based on the number of generations and the number of non-dominated points in rank 1.

The GA must be provided with an initial population that is created randomly. GAs move from generation to generation until a predetermined number of generations is seen or the change in the best fitness value is below certain threshold. In our implementation we have used a fixed number of generations as the termination criterion.

\textbf{Heuristic Algorithm}

As mentioned earlier the data layout problem is NP Complete. Further the ILP and the GA methods described in previous sections consume significant run-time to arrive at a solution and these methods are suitable only for obtaining an optimal data layout for a fixed memory architecture. But to perform memory architecture exploration, this problem is addressed in the following chapters, data layout needs to be performed for 1000s of memory architecture and it is very critical to
have a fast heuristic method for data layout. Using exact solving method such as Integer Linear Programming (ILP) are using an evolutionary approach, such as GA or SA which takes as much as 20 to 25 minutes of computation time for each data layout problem, may be prohibitively expensive for the memory architecture exploration problem. Hence in this section we propose a 3-step heuristic method for data placement.

**Algorithm**: SARAM Placement

1. sort the data sections in data-in-internal-memory in descending order of \( T \ C_i \)
2. for each data section \( i \) in the sorted order do
3. if data section \( i \) is already placed in DARAM
   continue with the next data section
4. else compute min-cost: minimum of \( \text{cost}(i, b) \) for all SARAM banks
5. endif
6. endif
7. find if there is potential gain in placing data \( i \) in DARAM by removing some of already placed sections
8. if there is potential gain in back-tracking
9. identify the data-set-from-daram-to-be-removed
10. find the alternate cost of placing the data-set-from-daram-to-be-removed in SARAM
11. if alternate cost > min-cost(\( i \))
12. continue with the placement of data \( i \) in SARAM bank \( b \)
13. update cost of placement: \( M_{\text{cyc}} = M_{\text{cyc}} + \text{cost}(i, b) \)
14. else // there is gain in backtracking
15. move the data-set-from-daram-to-be-removed to SARAM
16. update cost of placement:
   \( M_{\text{cyc}} = M_{\text{cyc}} + \text{cost}(g, b) \), for all \( g \) in data-set-from-daram-to-be-removed
17. place data \( i \) in DARAM and update cost of placement
18. endif
19. else no gain in backtracking, continue with the normal flow
20. continue with the placement of data \( i \) in SARAM bank \( b \)
21. update cost of placement: \( M_{\text{cyc}} = M_{\text{cyc}} + \text{cost}(i, b) \)
22. endif
23. endfor

Figure: Heuristic Algorithm for Data Layout

**Heuristic Algorithm and Genetic Algorithm Results**

To evaluate the performance of the heuristic and GA, we used the same 4 different embedded DSP applications explained in the previous section. Table 3.4 reports the performance of our Heuristic method and our GA. Column 1 shows the benchmark and column 2 indicates the number of instances of this module in the application. Column 3 shows the number of data sections in the application. Column 4 is the number of conflicts (sum of both parallel and self-conflicts), without any optimization. Column 5 indicates the unresolved conflicts when the heuristic placement
algorithm is used for data layout. Similarly, column 6 shows the number of unresolved conflicts for the genetic algorithm. We observe that both methods eliminate more than 90% the total number of conflicts. In the case of the JPEG decoder, the algorithms resolved all the conflicts and obtained an optimal memory placement. Although the performances, in terms of the unresolved conflicts, of the heuristic and GA method are comparable, the GA method performs better for moderate and large problems. We observe that for large problems the ILP method could not get the optimal solution even after hours of computation. Further, we believe that by tuning some of the parameters of the GA method

<table>
<thead>
<tr>
<th>App</th>
<th># Instances</th>
<th># Sections</th>
<th># Conflicts</th>
<th># Conflicts HP</th>
<th># Conflicts GP</th>
<th># Conflicts ILP</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE</td>
<td>1</td>
<td>12</td>
<td>356813</td>
<td>436</td>
<td>130</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>24</td>
<td>713626</td>
<td>1132</td>
<td>606</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>36</td>
<td>1070439</td>
<td>13449</td>
<td>11883</td>
<td>no result</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>48</td>
<td>1427252</td>
<td>67721</td>
<td>62390</td>
<td>no result</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>60</td>
<td>1784065</td>
<td>129931</td>
<td>126365</td>
<td>no result</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>72</td>
<td>2140878</td>
<td>190134</td>
<td>180591</td>
<td>no result</td>
</tr>
<tr>
<td>JP</td>
<td>1</td>
<td>14</td>
<td>94275</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>28</td>
<td>188440</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>42</td>
<td>282825</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>56</td>
<td>377100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LLP</td>
<td>16</td>
<td>80</td>
<td>556992</td>
<td>19639</td>
<td>19386</td>
<td>no result</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>160</td>
<td>1113984</td>
<td>165865</td>
<td>164395</td>
<td>no result</td>
</tr>
<tr>
<td>2D</td>
<td>4</td>
<td>23</td>
<td>5632707</td>
<td>33368</td>
<td>32768</td>
<td>32768</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>31</td>
<td>6727827</td>
<td>35768</td>
<td>33968</td>
<td>32768</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>39</td>
<td>7821867</td>
<td>37568</td>
<td>36368</td>
<td>32768</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>47</td>
<td>8916447</td>
<td>45638</td>
<td>38168</td>
<td>36368</td>
</tr>
</tbody>
</table>
(e.g., the cross-over and mutation probabilities, the size of the population, and number of generations), GA method can be made to perform significantly better even for large applications, and obtain close to.

REFERENCES


