A VLIW architecture
for
GSM benchmark

&

Code Optimization
using
Vex Compiler

Submitted by
Sumit Ahuja
Gurashish Singh Brar

Abstract

In this study, we present the design space exploration for VLIW architecture for the GSM benchmark and code optimization with vex compiler. The study was conducted in two overlapped phases of exploring the best machine configuration and insertion of code optimization compiler directives. We used a greedy algorithm for the design exploration and our results show a considerable improvement in cost and performance. The compiler directives improve the performance by increasing the ILP and thus exploiting the VLIW architecture.

Keywords: *VLIW, Design Space Exploration, ILP, Compiler Optimization, vex.*

I. INTRODUCTION

VLIW architecture permits the runtime ILP extraction to be shifted to the compile time allowing more flexibility and greater scope, thus exposing the compiler directly to the parallel structures of the machine.

The code optimization in compiler attempts to increase the ILP in order to exploit the VLIW architectures exposed parallelism. But if the resources are not there to exploit the parallelism offered by the compiler than this can have negative impact on the performance. However at the same time if the code is not optimized than the extra resources provided by the VLIW architecture adds to the cost and doesn't benefit the performance. This inherent interdependence between the compiler and the VLIW architecture makes the design space exploration trickier than the exploration in other architectures.

We tried to address this problem using an iterative approach. We define iteration in two steps, in the first step we run the design space exploration over a small space, followed by code optimization on the best machine configuration found in the previous step. The iterations are repeated, and the cost function used in the design space exploration is made more conservative on cost as the number of iterations increase.

It can be argued that the code optimization can be performed after estimating an upper limit on the ILP achievable for the benchmark being considered, thus choosing a large VLIW machine with enough resources to cover the ILP ceiling. But this argument has one flaw that is some of the code optimizations, especially concerning the cache, tend to improve the resource utilization. However if the machine model on which we test this code optimization has cache large enough that there are practically no misses, then these optimizations show a negative or no effect by increasing the number of cycles and getting no improvement in the cache behavior. This limits our ability to reduce the machine cost through better resource utilization using compiler optimizations.

In our case we ran three iterations and in the final iteration chose a machine model that is practical to the cost and whose resources are best utilized by the code optimization performed in the third iteration.

In section II, we discuss the design space exploration in further detail, providing the algorithm, results and the analysis. In section III, we discuss the code optimization and compilation techniques used in each of the iteration. We conclude in the section IV.

II. DESIGN SPACE EXPORATION ALGORITHM

As total number of parameter are large (9 for machine and 10 for configurations) so greedy algorithm is applied for getting the best performance in minimum iterations. Appropriate cost for every parameter is assigned on simple assumptions and then greedy algorithm compares the effect of changing the configuration on performance and if we are not getting any change for our cost function it picks the configuration with least cost for the same performance.

Taking into account the effect of all the parameters, relative cost is assigned to every parameter based on simple assumptions. For instance the cost of

IssueWidth and ALU cannot be same as increasing IssueWidth may demand more resources, so different weights are assigned to all the parameters and the cost is calculated as sum of all weights.

The relative cost to every parameter for machine model is as follows:

Parameter	***(Relative cost / unit)***
IssueWidth	5
MemLoad	3
MemStore	3
MemPft	3
Alu.n	1
Mpy.n	3
CopySrc.n	2
CopyDst.n	2
Memory.n	4

Where n corresponds to the cluster, it is 0 for one cluster 1 for 2 clusters, 2 for 3 clusters and 3 for 4 clusters.

Note: for one cluster cost of CopyDst and CopySrc is 0, as there is no inter cluster communication for one cluster.

The relative cost of every parameter for architecture configuration (Architecture units not visible to the compiler) is as follows:

Parameter	***(Relative cost /unit)***
lg2CacheSize	2
lg2LineSize	2
lg2Sets	1
lg2StrSize	2
lg2StrSets	1
lg2StrLineSize	2
lg2ICacheSize	2
lg2ICacheSets	1
lg2ICacheLineSize	2

Increasing the cluster size doesn't imply that the cost function is also increased proportionally. If all the other parameters are fixed and Clusters are increased that implies a lot of overheads are reduced because of long connections with in one cluster, this assumption is also included while determining the cost. Overheads due to more than one cluster are already included in our cost functions in form copyDest and copySource.

So for change in the clusters cost is calculated by the following rule:

if cluster =1
Total cost = sum of all relative costs.
If cluster = 2
Total cost = (sum of all relative costs) 0.8*
if cluster =4
*Total cost = (sum of all relative costs)*0.7*

For finding out the optimal configuration we have taken the following cost function.

Cost function= cycles*scale + cost (area)

We first try to see how much resources are needed by the benchmark and then according to our cost function the algorithm try to provide the configuration corresponding to minimum cost. This cost function takes account of change in cycles as well as change in the cost. Value of scale is chosen in such a way, that effect of cost (i.e. Area) and performance (i.e. Cycles) can equally be taken into account. We have applied greedy algorithm on this cost function.

RESULTS

As we can see from the graph of Performance and cost that by changing the number of clusters we are not getting any improvement in performance for the same cost. This is due to the minimum units per cluster that is enforced by the VEX system. Increasing the clusters thus increases the resources by a large factor. An example is a 4 cluster machine which has to have a minimum of 4 ALU/cluster, or a total of 16 ALUs. The benchmark under consideration doesn't have enough ILP to utilize so many ALUs. Infact we discovered that more than 5 ALUs has no effect on the performance.

So it is quite clear from the graph that in this case changing the clusters is not helpful. The graph shows some points where slight change in cost improves the performance significantly. These improvements can be explained due to change in the size of instruction and data cache.

The changes in cache configuration dominate the performance improvement; the changes in machine model parameters have relatively smaller effect on the performance.

In fig 1., the algorithm tries to find out the optimal vex configuration till the cost 105 and then for that optimal configuration algorithm tries to change the machine model we can see in graph that for those points there is not much change in performance. For the points from cost 120 to 130 we can see that algorithm tries to find out the best machine model for the particular configuration once it is achieved. It again goes to configuration and changes the parameters to find out the best possible match.

ANALYSIS

Our objective is to find out the configuration which provides the best performance/cost. So we identify the points on the Pareto curve. We can see that increasing clusters is not help full in our case as explained earlier due to limitations of the VEX system. The best configuration is obtained with a single cluster machine.

For the cost function discussed in the previous subsection, the best configuration obtained is shown in the following table:

For machine model

Parameter	***Value***
IssueWidth	3
MemLoad	1
MemStore	1
MemPft	1
Alu.0	5
Mpy.0	1
CopySrc	0
CopyDst	0
Mem.0	1

For Configuration

Parameter	***Value***
lg2cacheSize	10
lg2sets	1
lg2LineSize	4
lg2StrSize	4
lg2StrSets	0
lg2StrLineSize	4
lg2ICacheSize	16
lg2ICacheSets	0
Lg2IcacheLineSize	7

The cost for this configuration is 126 and the performance is 27499021 cycles.

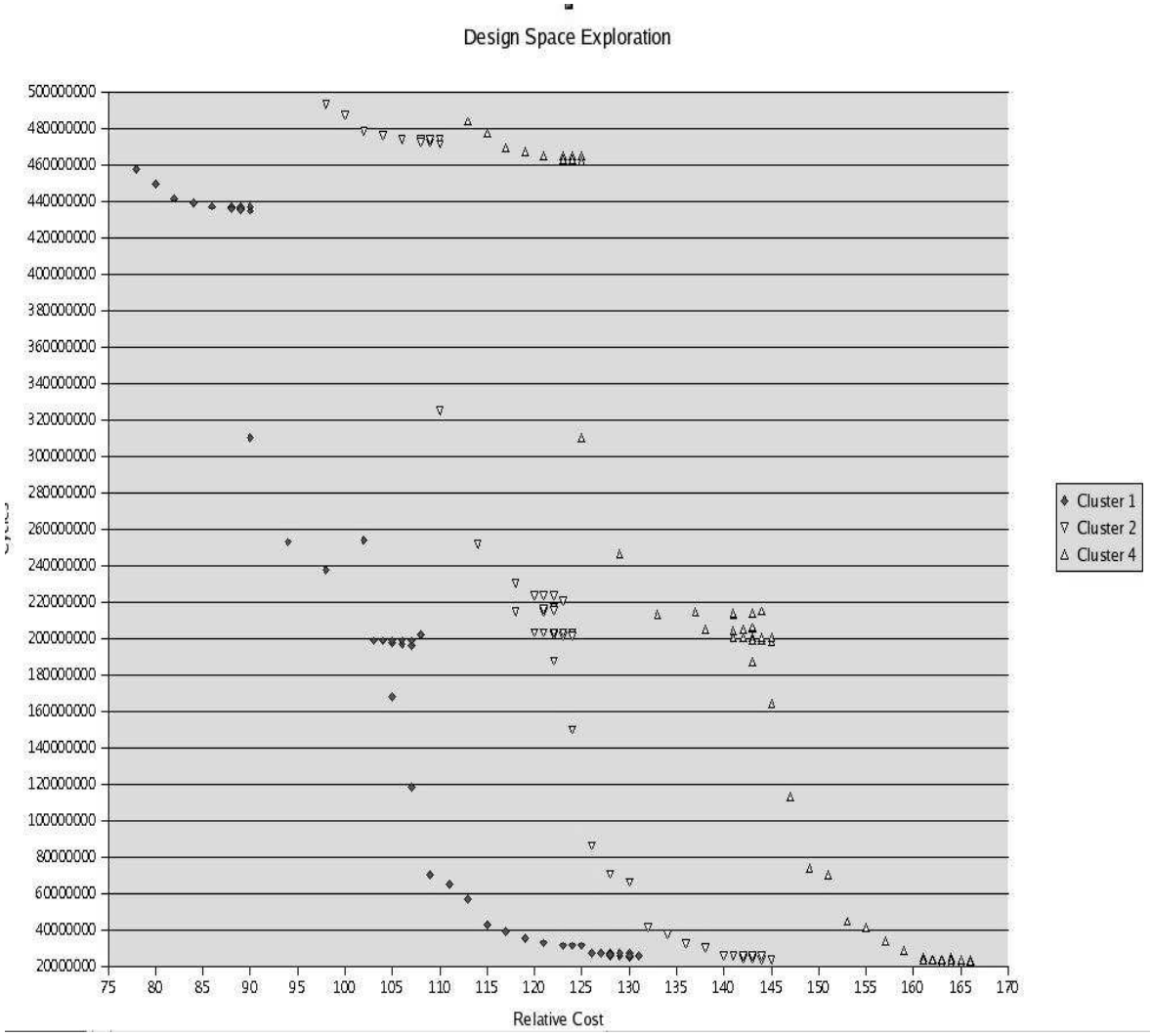

Fig 1: Design Space Exploration for gsm benchmark for VLIW architecture

III CODE OPTIMIZATION

In this section we present some of the techniques we used for optimizing the code.

As we mentioned earlier we did the code optimization and the design space exploration in iterations.

In the first iteration we did the function inlining. This was a good decision as this improves the performance significantly thus saving valuable computation time on the successive iterations. In the second iteration further improvements were made using the custom assembly instructions.

Finally in the third iteration after choosing the best model taking into account practical cost measures, we applied the VEX system's compiler pragmas.

The Vex System provides a set of compiler pragmas that allow a programmer to pass hints, directives and assertions to improve performance.

To inline the functions we had to identify the functions, which were called repeatedly and had relatively small code size. By inlining such functions the performance can be improved as we remove the overhead of a function call. A profile run was done and the functions that could be inlined were identified. Fig 4. shows the cycles distribution among various functions. As can be seen clearly the the basic operation functions performing arithmetic calculations are the ideal candidates for function inlining, as they are relative small in code size and consume the a lot of cycles, thus showing that they are called repeatedly.

Further improvement could be achieved by using the Vex systems custom instructions through the asm intrinsic. The Basic operations were again the ideal candidate for implementing the custom instructions, because of their simplicity.

Finally further code improvement was affected by using the VEX systems compiler pragmas. The loop unrolling was performed especially for the loops where the loop limit was known at compile time through the preprocessor definitions. We made use of the if_prob pragma, which specifies the branch probabilities, in places where a branch was repeatedly executed. The branch probability was computed through test runs and profile data. Memory disambiguation pragmas ivdep were used in places where the memory aliases were clearly non-existent.

RESULTS

Fig 2 shows the performance versus the optimization technique applied. By using the function inlining, over the basic operation functions, the performance is improved almost 4 times. By further using the custom assembly instructions for the basic arithmetic functions we double the performance. Fig 3 shows the relative performance improvement after optimizing a particular function. Fig 4 shows the cycle distribution among the most computationally expensive functions. In fig 3 we show the total cycles consumed, where as the fig 4 shows the number of cycles consumed by a particular function.

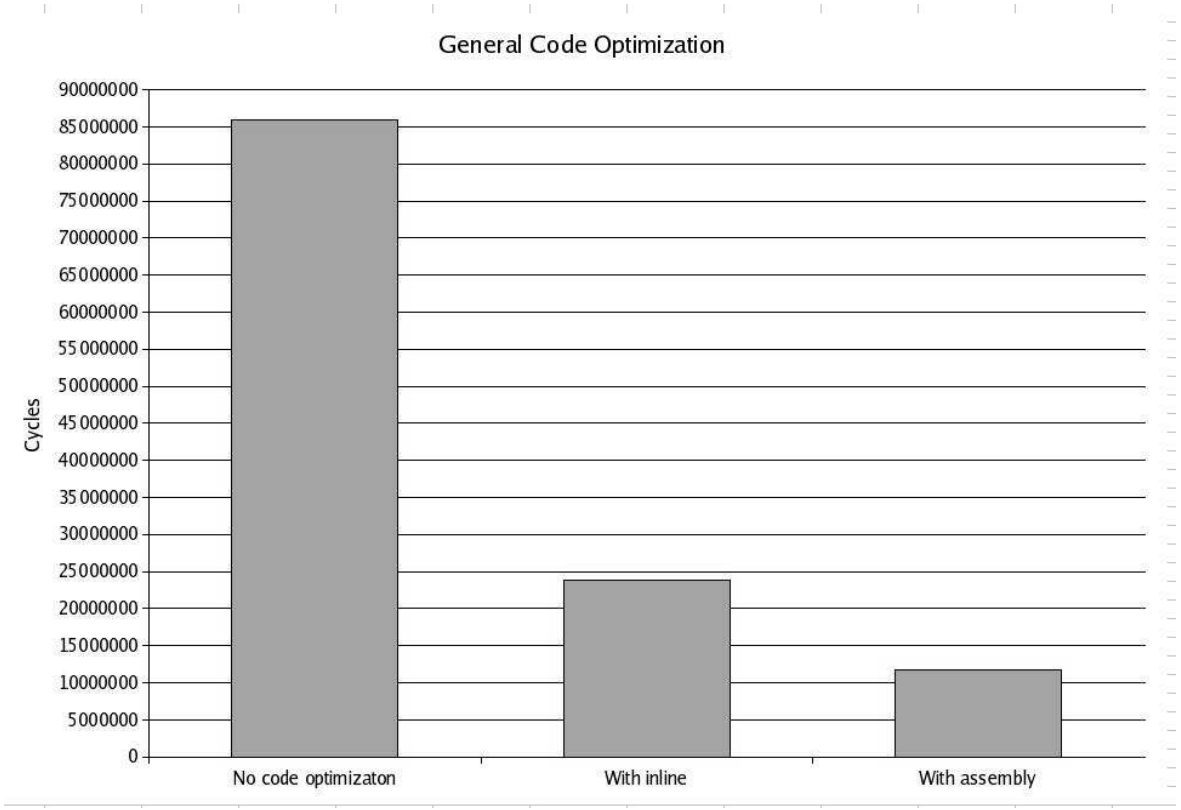

Fig 2: Code optimization in the first two iterations, using function inlining and custom assembly instructions.

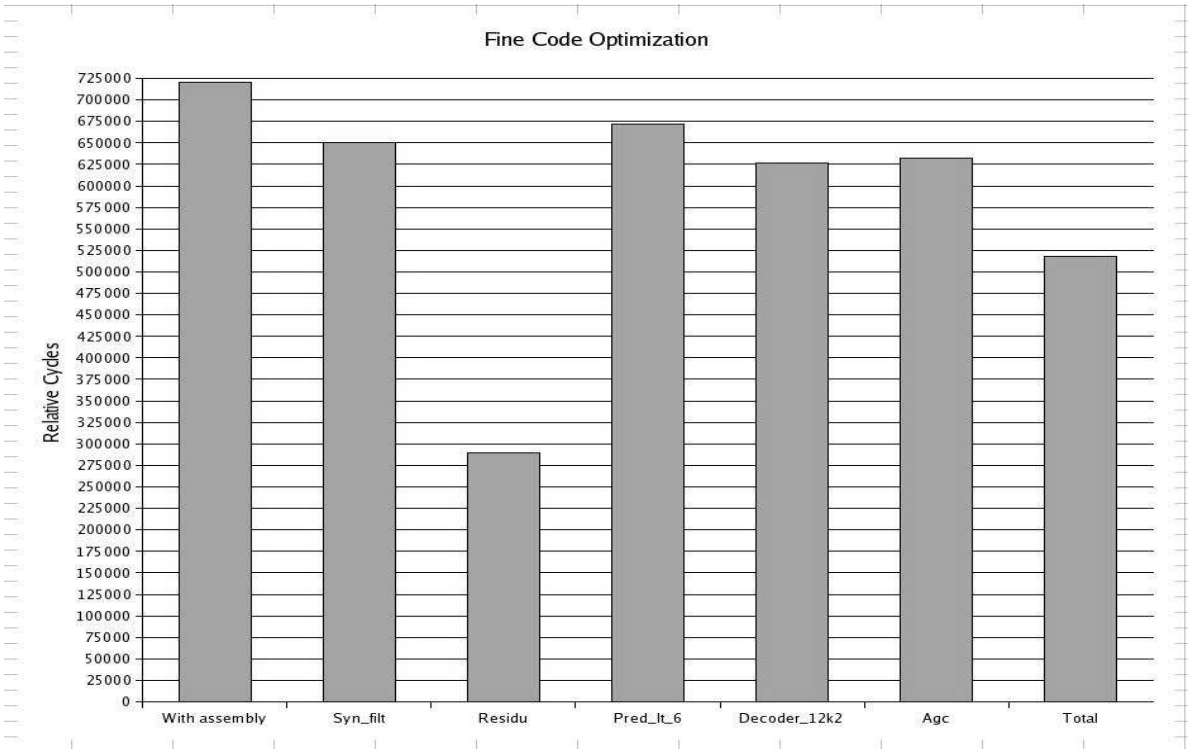

Fig 3: Code optimization in the final iteration, using complier pragmas.

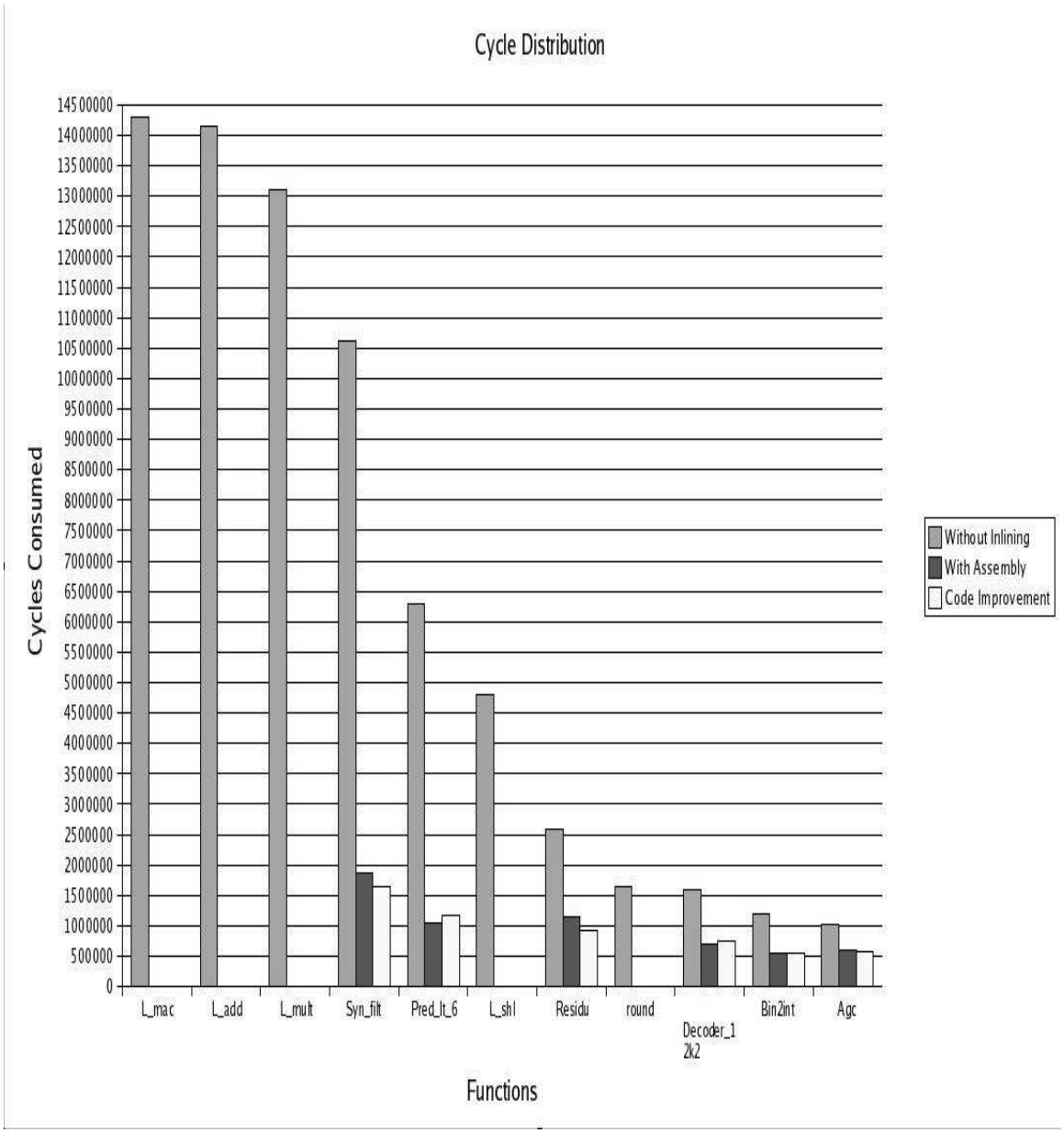

Fig 4: Per function cycle distribution, for various functions and for various levels of optimization.

ANALYSIS

The optimization achieved due to function inlining and implementing the custom assembly instructions is obvious.

The code optimizations performed on the functions identified provide some interesting insight into their behavior.

In the syn_filt function we mainly used aggressive loop unrolling, deciding the amount through profile runs.

In the residu function we identified a variable with temporal locality and used the stream pragma. The total cache misses were reduced by 4%. Loop unrolling was also applied to the inner loop in this function. It was the combined effect of the cache improvement and loop unrolling that brought the significant improvement in this function.

The pred_lt_6 and decoder_12k2 function show an interesting feature. Both of these functions consume more cycles after code optimization as can be seen in fig 4, but in fig 3 we see that the total cycles have reduced. This is because the code optimization applied to these functions improves the cache behavior. In case of pred_lt_6 the cache misses are reduced by 9 % and in case of decoder_12k2 the cache misses are reduced by 4 %. So though the cycles consumed by this particular function increased the total cycles show a reduction due to better cache efficiency.

In Fig 3 we notice that the cycle reduction achieved when we optimize all the functions doesn't follow a linear trend. Infact just optimizing the residu function achieves better results, than in the case when we optimize all. This can be explained due to increased instruction cache misses. The main optimization technique we used in most of the functions is forced loop unrolling over larger amounts, based on profile runs. The loop unrolling however has a disadvantage of increasing code size. Thus when we optimize all the functions the instruction cache misses due to the increased code size out weigh the benefits obtained from the loop unrolling.

IV CONCLUSION

In this study we have used a greedy approach to find the best VLIW machine configuration. We used an iterative approach to fine tune the code optimization and the exploration, and thus also saving valuable computation time. Our results show that VLIW architecture when accompanied with a suitable compiler can extract a significant amount of ILP from the gsm benchmark.