DEFINING NEIGHBORHOOD RELATIONS FOR FAST SPATIAL-TEMPORAL PARTITIONING OF APPLICATIONS ON RECONFIGURABLE ARCHITECTURES

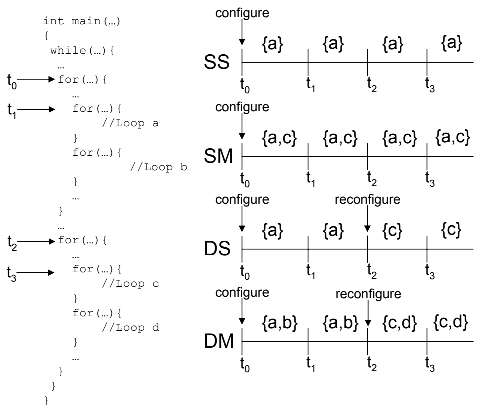
Joon Edward Sim, Tulika Mitra, Weng-Fai Wong
School Of Computing
National University of Singapore
{esim, tulika, wongwf}@comp.nus.edu.sg

Abstract

Considering both spatial and temporal partitioning, though potentially profitable, increases the complexity of the design space of applications for run-time reconfigurable architectures. In particular, the number of ways to partition is exponential and dynamic reconfiguration cost is difficult to estimate. These difficulties are particularly challenging for the implementation of neighborhood searches over the design space, such as the sheer amount of design space to be searched and time taken to evaluate each design point accurately. In order to address these challenges, this paper presents a framework that enables fast navigation of the design space using any *neighborhood search schemes. The key is a neighborhood relation which spans the entire spatial and temporal partitioning design space. Computed over a SEQUITUR compressed loop trace structure, this relation enables the fast estimation of neighboring design points. We implemented two neighborhood searches, Hill-Climb and Tabu search, to evaluate our technique. On four non-trivial benchmarks, these searches are accelerated by up to two orders of magnitude when using our proposed technique while finding optimal results most of the time.*

1. INTRODUCTION

Dynamically reconfigurable system-on-chip (SoC) solutions where an embedded processor core is tightly coupled with programmable logic (e.g., FPGAs, PLDs etc.) have recently become commercially viable. Examples include Xilinx II-Pro, Virtex 4 FX [1]. Typically, the programmable logic may be subjected to two forms of partitioning: spatial and temporal. Spatial partitioning allows multiple kernels to share the hardware at any single time instance while temporal partitioning allows different configurations to be reconfigured at different time instances (thus employing dynamic run-time reconfiguration). The former has the advantage of high resource utilization but limits the speed up obtainable from each kernel. The latter has the advantage of resource virtualization while incurring the penalty of reconfiguration overhead. This paper seeks to combine the two by enabling the fast exploration of a design space that encompasses both approaches.



```
int main(…)
{
 while(…){
  …
  for(…){
    …
    for(…){
        //Loop a
    }
    for(…){
            //Loop b
    }
    …
  }
  …
  for(…){
    …
    for(…){
        //Loop c
    }
    for(…){
        //Loop d
    }
    …
  }
 }
}
```

Fig. 1: Four partitioning strategies.

When considering both spatial and temporal partitioning, a design point may fall into one of the following categories, as shown in Figure 1. Static Single-kernel(SS) is the case where a single kernel is implemented in hardware without dynamic reconfiguration. Neither spatial nor temporal partitioning is required. In Figure 1, only loop `a` is selected to be realized in hardware. Static Multi-kernel(SM) is the case where the hardware is spatially partitioned among multiple kernels. There is no temporal partitioning. In Figure 1, loops `a` and `c` share the hardware. the rest are executed in software. Dynamic Single-kernel(DS) is the case where the hardware is temporally partitioned such that at any point exactly one kernel occupies the entire hardware. There is no spatial partitioning in this case. In Figure 1 loop a and loop c occupy the hardware at time t_0 and t_2, respectively. Loop b and d are executed in software. Finally, Dynamic Multi-kernel(DM) is the case where the Hardware is both spatially and temporally partitioned. In Figure 1, loop a,b share the hardware between time t_0 and t_2 and they are swapped out by loops c, d at time t_2.

Factors such as the application's runtime behavior, the available hardware resource and the reconfiguration overhead determines which of the four will yield better results. Restricting the search space to any one of these categories may result in sub-optimal results. The entire design space has to be explored in order to achieve the optimal solution. Furthermore, the estimation of the reconfiguration cost is non-trivial due to the non-deterministic nature of control-flow applications. These factors complicate the evaluation of the design point.

Neighborhood searches such as GRASP[5] and Tabu[6] have been used to solve complex combinatorial problems effectively. Thus one way to traverse the design space is to use some form of neighborhood search. A key insight in speeding up such searches is that all these techniques involve evaluating the neighbors of the current design point. Such evaluations are often time-consuming. In this paper, we will propose a way of speeding up such neighborhood searches. In particular, the main contributions of this paper are the following. Firstly, We define a neighborhood relationship among the design points that facilitates the navigation of the entire design space, both spatial and temporal. Secondly, We propose an efficient way of computing reconfiguration cost for the neighboring design points. Last but not least, Our experiments show that by employing our neighborhood estimation over a SEQUITUR-compressed loop trace, speeds up neighborhood searches by up to *two orders of magnitude*. Furthermore, it reports the optimal design point most of the time.

2. PRELIMINARIES

In this section, we formally define notions used in the description of our technique in Section 3.

2.1. The design space

The design space in the context of this paper is defined in terms of the following parameters.

- $K_1 \dots K_N$: Candidate kernels (loops)
- $k_{i,1} \dots k_{i,m_i}$: Different hardware implementation instances of kernel K_i with varying area and performance
- $a(k_{i,j})$: Area required by kernel instance $k_{i,j}$
- $s(k_{i,j})$: Savings in execution time due to choosing $k_{i,j}$ of kernel K_i over its software execution
- Loop trace indicating the run-time execution sequence of the candidate kernels
- A: Total hardware area constraint
- ρ: Time to perform one reconfiguration

The loop trace and candidate loops can be obtained through profiling [11]. The savings and area estimates of alternate hardware implementations can be obtained through behavioral synthesis and other methods of estimation. The details of profiling and estimation are beyond the scope of this paper and are orthogonal to its contribution. For particular architectures, we assume that ρ and A are constants.

2.2. Configurations and partitions

We define a *configuration* to be a non-empty set of kernels. A *configuration instance* is a particular implementation of a configuration. A configuration instance is obtained from a configuration by choosing particular instances corresponding to each member kernel. Using the example in Figure 1, a configuration can be of the form $\{K_a, K_b\}$ but configuration instance will be of the form $\{k_{a.i}, k_{b.j}\}$, where $k_{a.i}, k_{b.j}$ are hardware instances of loops a and b, respectively. The total area required by all the kernel instances in a configuration instance must not exceed the hardware area constraint. Given a configuration, selecting an optimal configuration instance is a sub-problem of the entire design-space exploration problem. Switching from one configuration to another incurs a reconfiguration cost.

A set of configurations is called a *partition*. Similarly, a *partition instance* is a particular implementation of a partition. A partition instance is obtained from a partition by choosing particular instances corresponding to each member configuration. A partition consisting of a single configuration corresponds to static configuration. This is SS when the configuration is a singleton, and SM otherwise. A partition with more than one configuration implies dynamic reconfiguration. This is DS when all the configuration are singletons, and DM otherwise. An empty partition implies that no kernel was chosen for hardware implementation. It should be noted that a chosen partition implicitly implies that the other kernels have been designated for software implementation. For a partition P, we refer the set of kernels designated for hardware implementation as $HW(P)$ and the set of the kernels designated for software implementation as $SW(P) = \{K_1, \dots, K_N\} - HW(P)$.

We have chosen to enforce a constraint that a loop kernel can appear in at most one configuration in a partition. The reason for such a constraint is if a loop is allowed to have multiple hardware versions, then it becomes necessary to dynamically infer the context under which a particular hardware version of the loop should be loaded, which further complicates the problem.

We now define the *savings* (in execution time) for configuration and partition instances.

$$s(C) = \sum_{k_{i,j} \in C} s(k_{i,j}) \qquad (1)$$

$$s(P) = \sum_{C \in P} s(C) - n(P) \times \rho \qquad (2)$$

Equation 1 shows the savings of a configuration instance. The savings of a configuration instance is simply the sum of the savings of its member hardware kernel instances. We compute the total savings of a partition instance in Equation 2 by offsetting the total reconfiguration time against the total savings of the member configuration instances. $n(P)$ is the expected number of reconfigurations for partition instance P and ρ is the time to perform one reconfiguration.

We shall now describe how our neighbors of a given current point in the design space can be evaluated over a SEQUITUR-compressed trace of loops.

3. FAST EVALUATION OF NEIGHBORING DESIGN POINTS

We consider the exploration of the design space described above using some neighborhood search scheme. One of the key components that is common among these search strategies is the evaluation of the design points within a certain neighborhood. We shall now describe a neighborhood relationship between partitions that both complete in coverage of the partitioning space and does not recompute unnecessarily when evaluating the neighbors of an evaluated partition. The necessary components of our techniques are 1)Loop traces encoded using SEQUITUR grammar, 2)Evaluation of a single partition (without any evaluated neighbors), 3)The neighborhood relationship proper and 4)Evaluation of a partition's neighboring points. We shall now describe each of these.

3.1. Evaluating a partition

We evaluate a partition by determining the optimal way to implement the partition. The savings of a partition instance depends on the savings of its member configuration instances and the number of reconfigurations. However, all the partition instances corresponding to a partition requires the same number of reconfigurations for a given loop trace. In the example shown in Figure 1, if loops a and b are put in one configuration, and c and d are put in another, there will be only one reconfiguration per iteration of the outer while loop, regardless of the instances of the loops chosen to be implemented in hardware. Therefore, *an optimal partition instance can be obtained by simply choosing optimal configuration instances.* Given this insight, we need both a method for choosing an optimal configuration instance and a method for calculating the number of reconfigurations. These are described in the following subsections.

3.1.1. Computing optimal configuration instance

Each loop kernel is associated with a number of alternative hardware implementations. A naive approach to find the optimal instance corresponding to a configuration would be to enumerate all feasible instances. However, this approach does not scale either with the number of kernels or with the number of instances corresponding to each kernel.

We handle this problem by pruning the number of instances corresponding to a kernel. We only keep the pareto-optimal instances corresponding to each kernel. Intuitively speaking, these instances are more efficient in terms of area utilization, giving better speedups with less area. After this pruning, the optimal configuration instance is found by an exhaustive enumeration of the remaining feasible configuration instances. We do not synthesize the configuration instances at this stage. Rather, the savings of a particular configuration instance is estimated using Equation 1 along with the area requirement.

3.1.2. Loop trace compression using SEQUITUR graph

We can compute the reconfiguration cost of any given partition by going through the entire trace. However, this step could be costly in terms of computation due to the size of the traces. Therefore, we compress the loop trace using SEQUITUR, in a format amenable for reconfiguration cost computation, as shown in the later subsections.

The SEQUITUR algorithm developed by Nevill-Manning [12] compresses a sequences of symbols (loop ids) by building hierarchical structures of frequently repeated sub-sequences. The SEQUITUR algorithm represents a finite string σ as a context free grammar. whose language is a singleton set $\{\sigma\}$. The SEQUITUR grammar can be represented as a directed acyclic graph. Each leaf vertex in the DAG corresponds to a candidate loop. Each intermediate vertex in the DAG represents a sub-trace and the root vertex represents the entire loop trace. An in-order traversal of the sub-graph rooted at a vertex retrieves the corresponding sub-trace. For example, an in-order traversal of the graph shown in Figure 3 generates the sequence ababacacbcbcababacacbcbcd.

3.1.3. Computing the number of reconfigurations

We can efficiently compute the number of reconfiguration of a partition through a single bottom-up traversal of the SEQUITUR DAG $G = (V, E)$ where V is the set of vertices and E the set of edges with complexity O(V+E). During the traversal for a particular partition, each vertex v in the DAG is labeled with the following: (1) the first and last hardware kernel in the the loop sub-trace represented by v, and (2) total number of reconfigurations for the loop sub-trace represented by v. During the same bottom-up traversal, we can compute the labels corresponding to an intermediate vertex by looking at the labels of its children as follows. Let v be an intermediate vertex with children $v_1 \dots v_k$. Let $n(v), f(v)$, and $l(v)$ represent the number of reconfigurations, first and last configuration of vertex v. Then $n(v) = \sum_{i=1}^{k} n(v_i) - \sum_{i=1}^{k-1} x_i$, where x_i is equal to 1 if $l(v_i) = f(v_{i+1})$ and 0 otherwise. The leaf vertices would be the base case where the loop sub-trace consists of only one candidate loop corresponding to the leaf vertex. Let v be a leaf vertex. $n(v)$ would be 1 if the candidate loop has been designated for hardware, 0 otherwise. $f(v)$ and $l(v)$ would be the candidate loop if the candidate loop has been designated for hardware, null otherwise. At the end of the traversal, the label at the root vertex yields the number of reconfigurations corresponding to the entire loop trace.

3.2. The neighborhood relationship

The neighbor of a partition (in the design space) is obtained by either (1) removing a hardware kernel from any of the member configurations (removing the entire configuration if the configuration becomes empty) or (2) adding a kernel currently in software into the partition(thus designating it for hardware implementation), either into one of the existing configurations or as a new configuration containing only this new kernel.

Figure 2 shows a partition $\{\{a\},\{b,c\}\}$ with all of its neighboring partitions. The removal of kernel c from the partition gives us the neighboring partition $\{\{a\},\{b\}\}$. There are 3 ways to add kernel d into the partition. Thus, adding d gives us partitions $\{\{a\},\{b,c,d\}\}$, $\{\{a,d\},\{b,c\}\}$ and $\{\{a\},\{b,c\},\{d\}\}$. Removal of kernel b gives us partition $\{\{a\},\{c\}\}$ and removing kernel a leaves us with $\{b,c\}$. There are 6 neighbors in all. The partition $\{\{c\}\}$ cannot be $\{\{a\},\{b,c\}$'s neighbor because they differ by more than one kernel.

A partition $\{\{K_c\}\}$ cannot be P's neighbor because they differ by more than one kernel. In general, a partition P has $|SW(P)| \times (|P|+1) + |HW(P)|$ neighbors, where $HW(P)$ and $SW(P)$ are the set of hardware and software kernels for partition P, respectively. $|P|$ is the number of configurations in partition P. This relationship is complete in the sense that any partition may be constructed starting from an empty partition (by adding the kernels one by one) and the empty partition may be reached by deconstructing any partition as well (by removing the kernels one by one).

From Figure 2, we observe that the reconfiguration cost of the neighboring design points cannot be computed simply by adding or subtracting the number of occurrence of the kernel added or removed to the design point. For example, when kernel c is removed, the reconfiguration cost does not decrease by 2 even though c's occurrences in the loop trace is 2. In the next section, we propose a way to compute

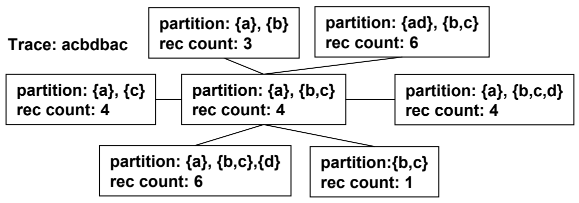


Fig. 2: Neighborhood relations example

the reconfiguration cost of neighbors efficiently by making use of the SEQUITUR graph.

3.3. Evaluating the neighbors simultaneously

In Section 3.1.3, we have shown how to compute the number of reconfigurations of a partition efficiently using a compressed loop trace. However, the number of neighbors of a partition can be quite large. Therefore, traversing the SEQUITUR graph for each neighbor can be quite expensive. Instead, given a partition P, we propose a method to compute the reconfiguration cost of *all* its neighbors through a single bottom-up traversal of the SEQUITUR graph.

Our method is based on the observation that only certain sequences in the loop trace need to be considered in order to compute the reconfiguration cost of a neighboring partition. Let K be an arbitrary kernel in configuration C of partition P, i.e., $K \in C \in P$. The loop trace contains many sequences of the form of $< K_x, S, K, S', K_y >$ where $K_x, K_y \in HW(P)$ and S, S' are (possibly empty) sequences of software kernels. In each of these sequences, there are three mutually exclusive possibilities:

1. K_x or K_y is in the same configuration as K. In this case, removing K has no effect on the number of reconfigurations.
2. K_x and K_y are in the same configuration, but not in the same one as K. In this case, removing K results in the savings of two reconfigurations.
3. K_x, K_y and K are in distinct configurations. In this case, removing K results in the saving of one reconfiguration.

The effect of removing a kernel K can thus be computed after identifying all distinct sequences of the form $s = < K_x, S, K, S', K_y >$ and the number of times, $w(s)$, each of these sequences occurs in the trace. The decrease in number of reconfigurations $d(s)$ can then be computed based on the three cases above. The total savings in number of reconfigurations is $\sum_s d(s) \times w(s)$. The effect of adding kernels can be computed in a similar way.

Therefore, given a partition P, we need to enumerate all sequences of the form $< K_x, S, K_i, S', K_y >$ and their frequency for each candidate kernel K_i. This will allow us to

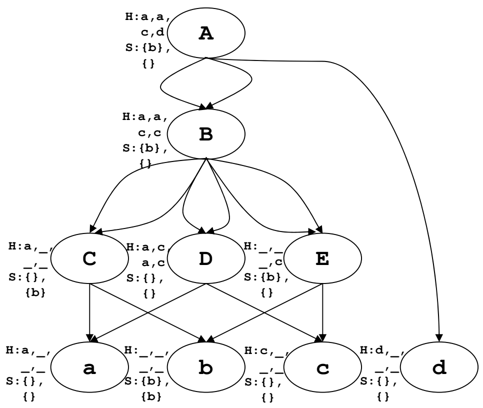


Fig. 3: A `SEQUITUR` graph labeled with H and S tags

compute the number of reconfigurations of a partition obtained through addition (if in software) or removal (if in hardware) of kernel K_i from partition P. By employing an extension of the labeling proposed in Section 3.1.3, this can be performed through one bottom-up traversal of the `SEQUITUR` graph.

3.3.1. Extending the labeling and enumerating the sequences

We observe that these sequences $< K_x, S, K_i, S', K_y >$ will always span two consecutive sub-traces. The extreme case of these sub-traces would be one sub-trace having one loop and the other sub-trace having two loops. Given that the each vertex in the `SEQUITUR` graph represents a sub-trace, we need to label the vertices in a way that allows such sequences to be identified easily.

Consider sub-traces represented by (not necessarily distinct) vertices v_i and v_{i+1} that are next to each other in the original trace (i.e. v_i and v_{i+1} would be children of the same parent vertex direct siblings). Assume further that the sub-trace represented by v_i to be $< \ldots, K_1, S_1, K_2, S_2 >$ and the sub-trace represented by v_{i+1} to be $< S_3, K_3, S_4, K_4, \ldots >$ where $K_1, K_2, K_3, K_4 \in HW(P)$ and S_1, S_2, S_3, S_4 represents (possibly empty) sequences of software kernels. In order to enumerate the $< K_x, S, K, S', K_y >$ sequence that spans these 2 sub-traces, we need to consider two cases. If K is in hardware, then both K_2 and K_3 are candidates for K. If K is in software, then all kernels occurring in S_2 and S_3 are candidates for K. We further note that once K_i is identified, K_x and K_y can be easily identified by finding the nearest preceding and subsequent hardware kernel.

The above consideration leads to the conclusion that both the first two and the last two hardware kernels of the sub-traces are needed to identify K_x, K and K_y. All software kernels in the sub-trace occurring before the first hardware kernel and after the last hardware kernel are also needed. Thus, we label each vertex, v_i, with a H tag and an S tag, as shown in Figure 3, where kernels `a`, `c` and `d` have been chosen for hardware implementation.

The H tag consists of two pairs of indices The first pair would be the first two hardware kernels of the sub-trace represented by v_i. The second pair would be the last two hardware kernels in the same sub-trace. In cases when the sub-trace represented by the v_i does not contain at least 2 hardware kernels (e.g., in the case of leaf vertices), the non-existent hardware kernels would be labeled with '_'.

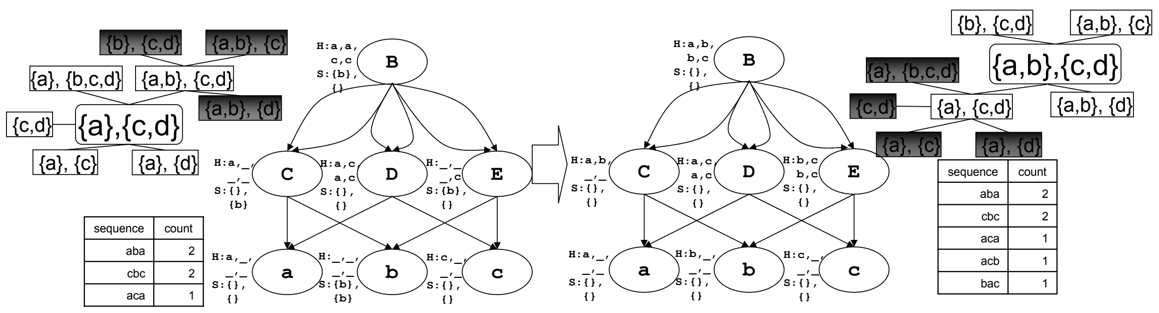
The S tag consists of two (possibly empty) *sets* of indices. The first set contains the software kernels that occur in the sub-trace represented by v_i *before* the first hardware kernel. The second set contains the software kernels that occur in the sub-trace represented by v_i *after* the last hardware kernel. In cases when the sub-trace does not contain any hardware kernels, all the kernels contained in the sub-trace are added to both sets.

This labeling process, i.e., computing the H and S tags, is done in a single bottom-up traversal of the `SEQUITUR` tree. Assuming that all the children vertices of v_i are properly labeled, the H and S tags of v_i can be computed using the H and S tags of the first and last child of v_i.

With the H and S tags in hand, we can now enumerate the $< K_x, S, K_i, S', K_y >$ sequences of v_i. It turns out that this can be done in the same bottom-up traversal by examining the labels of v_i's siblings and concatenating the possible sequences. For example, according to the tags of vertex `C`, there is only one hardware kernel `a` that occurs in the sub-trace represented by vertex `C` and `b` is the only software kernel that occurs after `a`. According to the tags of vertex `D`, the first hardware kernel of the sub-trace represented by vertex `D` is `a`. Thus the the sequence <`a,b,a`> spans the two sub-traces represented by vertex `C` and `D`. In fact, the sequence <`a,b,a`> also spans the sub-traces represented by two consecutive occurrences of vertex `C`. Thus, this sequence occurs twice in the sub-trace represented by vertex `B` and since vertex `B` itself has an occurrence count of 2, the sequence <`a,b,a`> occurs four times in total. With all the necessary sequences enumerated, all the neighbors of a design point can be evaluated easily based on Equation 2.

3.4. Putting it all together

A crucial step during a neighborhood search usually involved the following steps: evaluation of the current design point, comparison with neighboring design points and eventually selecting one particular neighboring design point to be the next step of the search. Figure 4 shows what happens during such a step in a search. It shows a partial view of the relevant design space, enumerated sequences and labeled `SEQUITUR` graph for 2 consecutive steps of a search. The current design point is shown in bold while the ignored



sequence	count
aba	2
cbc	2
aca	1

sequence	count
aba	2
cbc	2
aca	1
acb	1
bac	1

Fig. 4: An example of a move between neighboring design points.

design points during the step are shaded.

Initially, the current partition of the search is $\{\{a\},\{c,d\}\}$. When considering the move of adding kernel b into configuration $\{a\}$ (i.e. move to $\{\{a,b\},\{c,d\}\}$), the occurrence count of enumerated sequences $< a,b,a >$ and $< c,b,c >$ used to compute the change in the reconfiguration cost in such a move. Since kernel a and b would be in the same configuration, the increase in reconfiguration count is 2. Similar computations can be made for the other neighbors and are left as an exercise for the reader. The partition $\{\{a,b\},\{c,d\}\}$ is selected(the criterion depends on the search algorithm) for the next step in the search. Consequently, the SEQUITUR tree needs to be relabeled according to the methods described in section 3.3.1. To complete the move, sequences $< a,c,b >$ and $< b,a,c >$ are enumerated to reflect the case that kernel b is now in hardware. The search can thus continue after the move is completed.

4. EXPERIMENTAL EVALUATION

4.1. Experimental setup

We use four non-trivial benchmarks for our experimental evaluation: a JPEG encoder(cjpeg), a JPEG decoder (djpeg), an encryption key exchange program (dh), and an MPEG encoder (mpegenc). We use the Trimaran compiler infrastructure [7] to generate the input parameters for the design space exploration problem. In particular, we have implemented a loop profiler that selects a loop kernel (both inner and outer) as a candidate if its computation time exceeds more than 1% of the entire application.

In view of a lack of estimation tools, we have to pre-generate the area and timings estimation. We applied loop unrolling with various loop unroll factors to each candidate loop kernel. To obtain hardware performance and area required for each kernel instance, we automatically generate Handel-C code [2] from Trimaran's Elcor intermediate representation. The timings and area estimations of these alternate hardware implementations are subsequently obtained through synthesis using the Celoxica DK design suite and Xilinx ISE tools. The target FPGA for synthesis is the Xilinx 2000E model[1]. To evaluate our framework, we developed three searches: Exhaustive, Hill-Climb and Tabu.

Exhaustive search In a pre-processing phase, we compute the optimal configuration instances corresponding to all possible configurations of the candidate kernels using the method described in section 3.1.1. The main phase then enumerates all possible partitions. The enumeration algorithm used is by Kreher and Stinson [10]. This algorithm ensures the proper enumeration of all the partitions. The savings of a partition is defined as the savings of its optimal instance. Evaluation of the savings of a partition is described in Section 3.1. The partition instance with the maximum savings is chosen as the optimal partition instance. It should be noted that apart from how the optimal configuration instances are chosen, the Exhaustive search algorithm does not make use of the rest of the framework.

Hill-Climb search We start with an empty partition. This ensures that our solution is at least as good as an all-software solution to begin with. We then evaluate all its neighbors using the technique described in the previous subsection. We choose the neighbor with the maximum savings (i.e., minimum execution time). The search then moves to this new design point and examines its neighbors. We always maintain the best partition obtained so far. The search terminates at a design point if we cannot find any partition in the neighborhood that is better than the current best partition. It should be noted that our Hill-Climb search draws heavily on the framework described in Section 3, making full use of the neighborhood relationships and the efficient evaluation of the neighbors.

Tabu search We modify the Hill-Climb search so that the search does not terminate when a local maximum is reached. Instead, we maintain a tabu list of design points which have been visited and the most profitable neighbor is always visited, regardless of whether the neighbor yields more savings than the current design point. If the particular neighbor design point is on the tabu list, the next most profitable neighbor not present on the tabu list is visited. The search terminates when the number of moves made reaches

Benchmark	Num. of Candidate Kernels	Size of Comp. Trace KBytes	Avg. Exhaustive Search Time (sec)	Avg. Hill-Climb Search Time (sec)	Avg. Tabu Search Time (sec)
`cjpeg`	11	1	17719.72	0.24	3.17
`djpeg`	7	4.3	17.34	0.04	1.06
`dh`	7	72	3837.87	3.73	112.04
`mpegenc`	6	74	245.88	0.96	18.44

Table 1: Running times of the algorithms

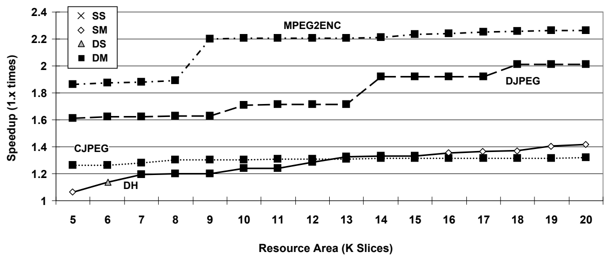


Fig. 5: Speedups plotted against increasing hardware resource

a certain limit. In our experimentation, d the number of entries on the tabu list is 100 and the maximum number of moves is a logarithm of the design space size to base 1.05.

We have implemented the search algorithms in `C++` compiled by `gcc` version 4.1.2. We run the experiments on a 2.8 GHz Pentium 4 machine in the GNU-Linux environment. All the run-time of the search algorithms reported are based on Pentium's hardware cycle counters. The Trimaran framework allows us to define a VLIW machine with 4 integer units, , 1 branch unit and and 1 load/store unit. We obtain the cycle-accurate measure of the all-software solution based on the simulator reports of the Trimaran framework.

Table 1 show the number of candidates kernels for each benchmark and the average running times of the implemented searches for all benchmarks. These values are obtained by running the experiments with varying input parameters described in section 4.2. This table demonstrates the infeasibility of the exhaustive search approach. The number of kernels increases the running exponentially, even though `cjpeg` has the smallest compressed trace among all the benchmarks, the running time was close to 4 hours to run the exhaustive search. Table 1 shows the average running time of the Hill-Climb search and Tabu search as well. It should be noted that the length of the trace dominates the running time when the number of kernels is the same. We can conclude this by observing that the running time of `dh` is longer compared to `djpeg` even though the number of kernels is the same. Our experiments show that Hill-Climb is able to find the optimal design point more than 90% of the cases while Tabu search found the optimal design point in all of our experiments.

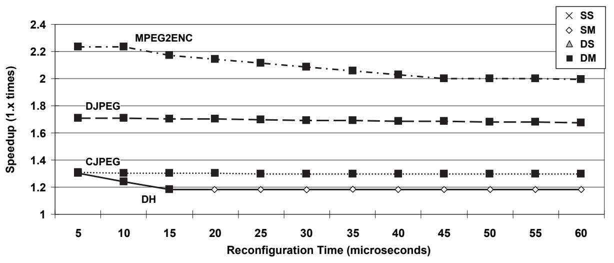


Fig. 6: Speedups plotted against increasing reconfiguration time

4.2. Scaling the hardware resource and reconfiguration time

Both Figure 5 and Figure 6 plots the results of exhaustive search in order to give an idea of the design space. The lines on the graphs have been labelled with the benchmark name and the plotted points with shapes to indicate the type of the solution. For example, following Figure 5, benchmark `dh` yields a SM partition under a resource constraint of 5K slices and then a DS partition under the resource constraint of 6K slices. Beyond, the resource constraint of 16K slices, `dh` is optimally implemented with SM. It should be noted that while many of the plotted points show DM to be the optimal partition, the kernels included in the partition are not uniformly the same for the same benchmark. Sometimes, as resource constraints increase or decrease, certain kernels has to be moved to software or hardware, though the resulting partition is still ostensibly DM.

Figure 5 plots the speedups of the optimal design point through exhaustive search against increasing resource area. The reconfiguration time is set at 10 μseconds. We observe that placing multiple kernels in hardware yields the optimal results in most cases except for `dh`. For the `dh` benchmark, if the resource available goes beyond 15K slices, the SM will give better speedup than DM. This is because when the area is large enough to be shared by all the kernels, we no longer gain from dynamic reconfiguration. If the resources available decrease below 7K slices, DS and SM gives better speedup. This could be because the available resources becomes too small to hold multiple kernel. It should be mentioned that though the graph shows DM to be an optimal design point most of the time, the partition solutions for each benchmark are not the same throughout.

Figures 6 plot speedups of the optimal design point as reconfiguration time increases. The area is fixed at 10,000 logic slices. The speedups of `cjpeg` and `djpeg` remain almost constant because the optimal design point gives a partition which yields quite a small reconfiguration cost while achieving the speedup at the same time. As a result, the change in the reconfiguration cost is insufficient to alter the optimal design point. For the `dh` benchmark, if the reconfig-

Benchmark	tabu-trace	tabu-seq	hc-trace	hc-seq
cjpeg	97.24x	10.34x	10.12x	1.13x
djpeg	21.77x	8.57x	6.20x	13.92x
dh	45.35x	11.06x	31.62x	8.33x
mpegenc	439.18x	18.72x	242.92x	9.29x

Table 2: Slow-down of `trace` and `seq` Tabu and Hill Climb

uration time is small, it will still employ dynamic reconfiguration with multiple kernels. The trade-off between more kernels and reconfiguration cost comes in when the reconfiguration time increases beyond 15μseconds.

4.3. Impact of using `SEQUITUR` and the neighborhood relation

In order to demonstrate the difference made when the `SEQUITUR` compressed trace and neighborhood relationship are used, we implemented `-trace` and `-seq` versions of the Tabu and Hill-Climb search. The `-trace` version traverses the *uncompressed* loop trace to compute the reconfiguration cost of a design point. The `-seq` version traverses the compressed `SEQUITUR` loop trace and calculates reconfiguration cost *without* the neighborhood relationship, i.e. using the technique described in section 3.1.3 every time a design point is evaluated. Table 2 shows the various slow-downs of these implementation compared to the Tabu and Hill-Climb searches that employ *both* the `SEQUITUR` compressed trace and neighborhood relationship. The slow down is significant. Although using the `-seq` version gives about an order of magnitude of speedup compared with the `-trace` version, employing the neighborhood relationship makes the search a further order of magnitude faster in general, except in the case of Hill-Climb search for `cjpeg`.

5. RELATED WORKS

Research work on hardware-software partitioning may be categorized according to the granularity of computation being placed in reconfigurable logic. Many works [4, 3] have focused on partitioning acyclic task graphs (i.e. assigning task nodes to either software or hardware for execution). The main motivation behind such approaches is to exploit task-level parallelism. The reconfiguration cost model considered is simple because each hardware task needs only be reconfigured once.

Other works [11, 13] focused on selecting loops for hardware implementation to exploit instruction level parallelism. Among these, Koch et al's [9] proposal of configuration merging is similar to our work. They proposed a dynamic programming algorithm to determine a way to merge the configurations with minimal number of reconfigurations. However, it should be noted that minimizing reconfigurations does not imply maximal overall performance gain. More recently, Huynh et al.[8] proposed an iterative heuristic approach to partition applications for dynamically reconfigurable custom instruction sets.

6. CONCLUSION

In this paper, we considered the problem of exploring the design space of dynamically reconfigurable SoCs for spatial and temporal partitioning . Specifically, we proposed a means of speeding up neighborhood searches of such design spaces by a novel method of estimating the design points near the current one in a compressed trace. We showed that our technique works for both Hill-Climb and Tabu search. On four benchmarks, we found that using our neighboring design point computation method, the searches were faster by up to two orders of magnitude while reporting near-optimal solution most of the time.

7. REFERENCES

[1] Xilinx Corp. http://www.xilinx.com/products.

[2] Celoxica ltd. Application Note 69 v1.0 http://www.celoxica.com/support, 2001.

[3] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt. Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *DAC*, pages 335–340, 2005.

[4] K. B. Chehida and M. Auguin. Hw/sw partitioning approach for reconfigurable system design. In *CASES*, pages 247–251. ACM Press, 2002.

[5] T. Feo and M. Resende. Greedy randomized adaptive search procedures. In *Journal of Global Optimization*, volume 6, pages 109–133, 1995.

[6] F. Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Oxford, England, 1993. Blackwell Scientific Publishing.

[7] http://www.trimaran.org. Trimaran ILP Research Infrastructure, 1998.

[8] H. P. Huynh, J. E. Sim, and T. Mitra. An efficient framework for dynamic reconfiguration of instruction-set customization. In *CASES*, pages 135–144, 2007.

[9] N. Kasprzyk, J. van der Veen, and A. Koch. Configuration merging for adaptive computer applications. In *FPL*, pages 217–222, 2005.

[10] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms Generation, Enumeration and Search*. CRC Press Inc, 1998.

[11] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. *DAC*, pages 507–512, 2000.

[12] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[13] G. Stitt, F. Vahid, and S. Nematbakhsh. Energy savings and speedups from partitioning critical software loops to hardware in embedded systems. *ACM Transaction on Embedded Computing Sys.*, 3(1):218–232, Feb. 2004.