An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization*

Huynh Phung Huynh, Joon Edward Sim, Tulika Mitra
Department of Computer Science
National University of Singapore
{huynhph1,esim,tulika}@comp.nus.edu.sg

Abstract

We present an efficient framework for dynamic reconfiguration of application-specific custom instructions. A key component of this framework is an iterative algorithm for temporal and spatial partitioning of the loop kernels. Our algorithm maximizes the performance gain of an application while taking into consideration the dynamic reconfiguration cost. It selects the appropriate custom instructions for the loops and clubs them into one or more configurations. We model the temporal partitioning problem as a k-way graph partitioning problem. A dynamic programming based solution is used for the spatial partitioning. Comprehensive experimental results indicate that our iterative partitioning algorithm is highly scalable while producing optimal or near-optimal (99% of the optimal) performance gain.

1 Introduction

Current generation embedded systems designs are characterized by the increasing demand on higher performance under stringent time-to-market constraints. In this context, application-specific customizable processor cores strike the right balance between performance and design efforts. A customizable processor is, in general, configurable with respect to the micro-architectural parameters. More importantly, a customizable processor may support application-specific extensions of the core instruction set. Custom instructions encapsulate the frequently occurring computation patterns in an application. They are implemented as custom functional units (CFU) in the datapath of the existing processor core. CFUs improve performance and energy consumption through parallelization and chaining of operations. Some examples of commercial customizable processors include Lx [14], $\mathrm{ARC}^{\mathrm{TM}}$ core [2], Xtensa [15] and Stretch S5 [3].

*A preliminary version of this paper appeared in the proceedings of the 7th ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2007

However, the total area available for the implementation of the CFUs in a processor is quite limited. Therefore, we may not be able to exploit the full potential of all the custom instructions in an application. This is particularly true if the application consists of a large number of kernels and each kernel requires unique custom instructions — a scenario that is quite common in high-performance embedded systems. Furthermore, it may not be possible to increase the area allocated to the CFUs due to the linear increase in the cost of the associated system. In this context, runtime reconfiguration of the CFU fabric appears quite promising. Here the set of custom instructions implemented in the fabric can change over the lifetime of the application. For multi-kernel applications, runtime reconfiguration is specially attractive, as the fabric can be tailored to implement *only* the custom instructions required by the active kernel(s) at any point of time. Of course, this virtualization of the CFU fabric comes at the cost of reconfiguration delay. The designer has to strike the right balance between the number of configurations and the reconfiguration cost.

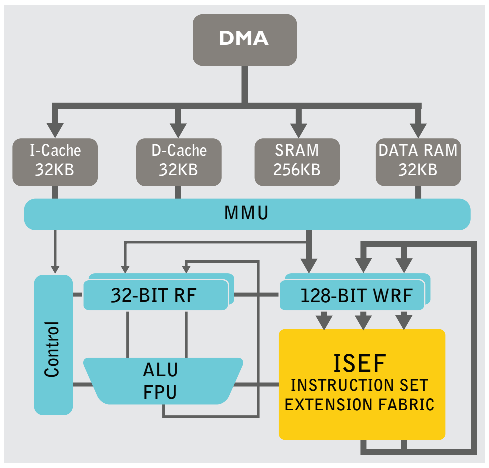


Figure 1: Stretch S5530 datapath [28].

To exploit this performance potential, commercial customizable processors supporting dynamic reconfiguration have been proposed. For example, Figure 1 shows the Stretch S5 engine [28] that incorporates Tensilica Xtensa RISC processor [15] and the Stretch Instruction Set Extension Fabric (ISEF). The ISEF is software-configurable datapath based on programmable logic. It consists of a plane of arithmetic/logic elements (AU) and a plane of multiplier elements (MU) embedded and interlinked in a programmable, hierarchical routing fabric. This configurable fabric acts as a functional unit to the processor. It is built into the processor's datapath, and resides alongside other traditional functional units such as the ALU and the floating point unit. The programmer defined application specific instructions (Extension Instructions) are implemented in this fabric. The core processor issues the Extension Instructions to ISEF, which performs the computation and returns the result.

The distinguishing aspect of ISEF is that it is run-time configurable and reloadable. If the computation resource requirement of the custom instructions exceeds the capacity of ISEF, then the instructions can be partitioned into different *configurations*. When a user-defined instruction is issued, the S5 hardware checks to make sure the corresponding configuration is loaded into the ISEF. If the required configuration is not present in the ISEF, it is automatically loaded prior to the execution of the user-defined instruction. In summary, the ISEF allows the system designers to define new instructions at runtime and thus extend the processor's instruction set.

Currently, it is the programmer's responsibility to manually choose and define the custom instructions and the configurations for architectures such as Stretch. Choosing an appropriate set of custom instructions for an application itself is a difficult problem. Significant research effort has been invested in developing automated selection techniques for custom instructions. Runtime reconfiguration has the additional complication of both *temporal and spatial partitioning* of the set of custom instructions in the reconfigurable fabric. Figure 2 shows how a C code accelerated with different Custom Instruction Sets (CIS) configures the CFU fabric during run-time. A CIS is a set of custom instructions corresponding to a program fragment. When the CFU fabric can accommodate more than one CIS, it is spatially partitioned among them. In Figure 2, configuring the CFU fabric with both `CIS-1` and `CIS-2` at the same time constitutes an example of spatial partitioning. The CFU fabric is temporally partitioned when it is loaded with different configurations during run-time. In our example, the CFU fabric is configured with `CIS-3` after exiting the `for` loop. Therefore, the custom instructions of the entire application are partitioned into two temporal configurations: {`CIS-1, CIS-2`} and {`CIS-3`}.

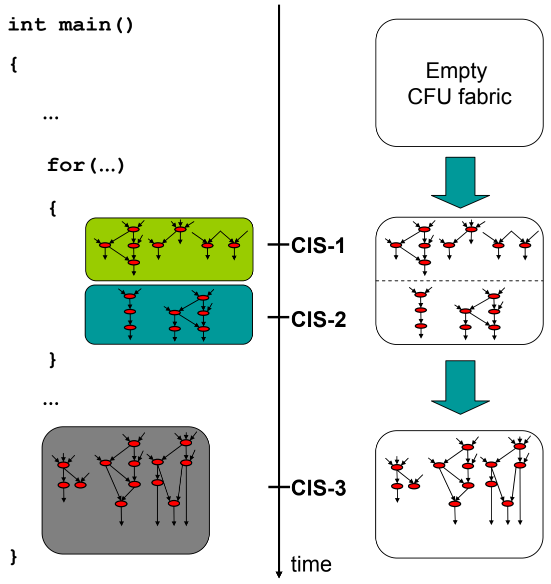


Figure 2: Spatial and temporal partitioning of the custom instructions of an application and the state of the CFU fabric during execution.

In this paper, we develop a framework that starts with an application specified in ANSI-C and automatically selects appropriate custom instructions as well as clubs them into one or more configurations. We first extract a set of compute-intensive candidate loop kernels from the application through profiling. For each candidate loop, we generate one or more custom instruction-set versions differing in performance gain and area tradeoffs in addition to the purely software version. The key component of our framework is an iterative partitioning algorithm. The partitioning algorithm selects appropriate custom instruction-set versions for the loops implemented in fabric and clubs them into suitable configurations to achieve the highest performance gain. We model the temporal partitioning of the custom instructions into different configurations as a k-way graph partitioning problem. We develop a dynamic programming based pseudo-polynomial time algorithm for the spatial partitioning of the custom instructions within a configuration. To the best of our knowledge, this is the first work that attempts automated custom instructions selection in the context of instruction-set extensible processor platforms with dynamic reconfiguration.

Most hardware-software partitioning solutions for FPGAs work at a coarse-grained level (such as task level). However, as we would like to accelerate complete applications specified in high-level programming languages such as ANSI-C, we focus on hot loop kernels instead. Note that the reconfiguration cost model at task level [9, 20] and data flow graph level [27] are simple because the underlying directed acyclic graph representation ensures at most one reconfiguration between any two nodes. In contrast, our dynamic reconfiguration cost model is complex as the number of reconfigurations for one loop depends on temporal partitioning of all the other loops. Furthermore, our methodology allows custom instruction sets corresponding to more than one loop to be placed within a single configuration. Thus spatial partitioning also plays a role in determining the performance gain of the application. The only other loop-level temporal partitioning work that we are aware of [24] considers only one loop per configuration.

The remainder of this paper is structured as follows. Section 2 describes the system design flow. In Section 3, we present the problem formulation and a motivating example. Section 4 details our partitioning algorithm. Experimental setup and evaluation are described in Section 5. The related work are discussed in Section 6. Finally, Section 7 concludes the paper.

2 System Design Flow

Figure 3 shows the system design flow. The input to the design flow is the C source code of the application we want to accelerate. The output is the application accelerated with custom instructions and the synthesized datapath for each configuration. In the following, we describe each component of this design flow.

Hot loops detection Taking our cue from Amdahl's law, we focus on the loops that take up a significant portion of the application's total execution time. In particular, we define a loop with execution time greater than a certain percentage (typically $\geq 1\%$) of the application's overall execution time to be a *hot* loop. The hot loop detector identifies such loops through profiling. Although the total number of loops in an application may be large, we consider only the hot loops to reduce the computation cost of the partitioning algorithm significantly. At the same time, the performance gain we obtain is still comparable to the case where all the loops of the application are

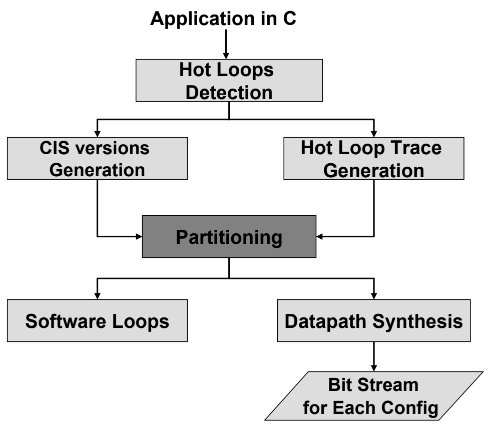


Figure 3: System design flow

considered. This is because the minimal performance gain of the cold loops are more than offset by the high reconfiguration overhead.

Custom instruction-set versions generation We generate multiple custom instruction-set (CIS) versions for each hot loop with a trade-off between hardware area and performance gain. A CIS version consists of a set of custom instructions extracted from the corresponding loop under an area constraint. Each CIS version is characterized by its area and performance gain. In general, the performance gain of a CIS version increases with larger area. To generate the CIS versions for a loop, we first identify [4, 5, 11, 12, 19, 30, 31] a large set of candidate patterns from the loop. Given this library of patterns, in the second step, we select a subset to maximize performance gain under hardware area constraint [4, 10, 11, 12, 23, 29]. As the area increases, a CIS version with higher performance gain will be generated by selecting a larger subset. Moreover, different CIS versions can be generated by loop transformations such as loop unrolling, software pipelining, loop fusion, and others.

Loop Trace The control flow among the hot loops is captured in the form of a loop trace (execution sequence of the loops) obtained through profiling. For typical embedded applications we have profiled, the number of hot loops and the loop trace size are quite small. For longer loop trace, we can use lossless compression techniques (such as SEQUITUR algorithm [26]) to compactly maintain the loop trace.

The hot loops with CIS versions and the loop trace are fed to the partitioning algorithm that decides the appropriate CIS version and configuration for each loop. The selected CIS versions to be implemented in hardware are then input into the datapath synthesis tool. It generates the bit stream corresponding to each configuration (based on the result of temporal partitioning). These bitstreams are used to configure the fabric at runtime. The remaining loops are implemented in software on the core processor. Finally, the source code is modified to exploit the new custom instructions.

3 Partitioning Problem

We now formally define the partitioning problem for dynamic reconfiguration of custom instructions, which is the focus of this paper.

The input to the partitioning step is the set of hot loops $L = \{l_i | i = 1...N\}$. Each loop is associated with multiple custom instruction-set (CIS) versions with a trade-off between hardware area and performance gain. Let $l_{i,j}$ (for $j = 1 \ldots n_i$) be the j^{th} CIS version corresponding to loop l_i where n_i is the number of CIS versions of loop l_i. In addition, let $gain_{i,j}$ and $area_{i,j}$ denote the performance gain and area requirement of $l_{i,j}$. We assume that $l_{i,1}$ corresponds to the software loop without any custom instructions, i.e., $area_{i,1} = 0$ and $gain_{i,1} = 0$. For each loop l_i, only one of its CIS versions will be selected for implementation. For example, if $l_{i,1}$ is selected, loop l_i will be implemented in software without any custom instruction enhancements.

The control flow among the loop kernels is input in the form of a loop trace. Finally, $MaxA$ represents the hardware area available for each configuration and ρ represents the time required for a single reconfiguration. In this work, we do not consider partial reconfiguration, i.e., a configuration is completely replaced by another configuration in the fabric. Hence both $MaxA$ and ρ are constants. Intra-loop reconfiguration incurs high reconfiguration cost. Thus we do not allow custom instructions corresponding to a loop to straddle across configuration boundaries. In other words, the selected CIS version of a loop is completely accommodated within a configuration, i.e., $area_{i,j} \leq MaxA$ (for $i = 1 \ldots N, j = 1 \ldots n_i$). Each configuration, however, consists of CIS versions corresponding to one or more loops. Thus the problem boils down to

1. *Temporal partitioning* of the loops selected for hardware acceleration with CIS into one or more configurations, and

2. *Spatial partitioning* of the loops within a configuration by selecting appropriate CIS version for each loop.

The performance gain of the application is then defined as

$$Performance\ \ gain = \left(\sum_{i=1}^{N} \sum_{j=1}^{n_i} s_{i,j} \times gain_{i,j} \right) - r * \rho \tag{1}$$

$$\sum_{j=1}^{n_i} s_{i,j} \leq 1 \tag{2}$$

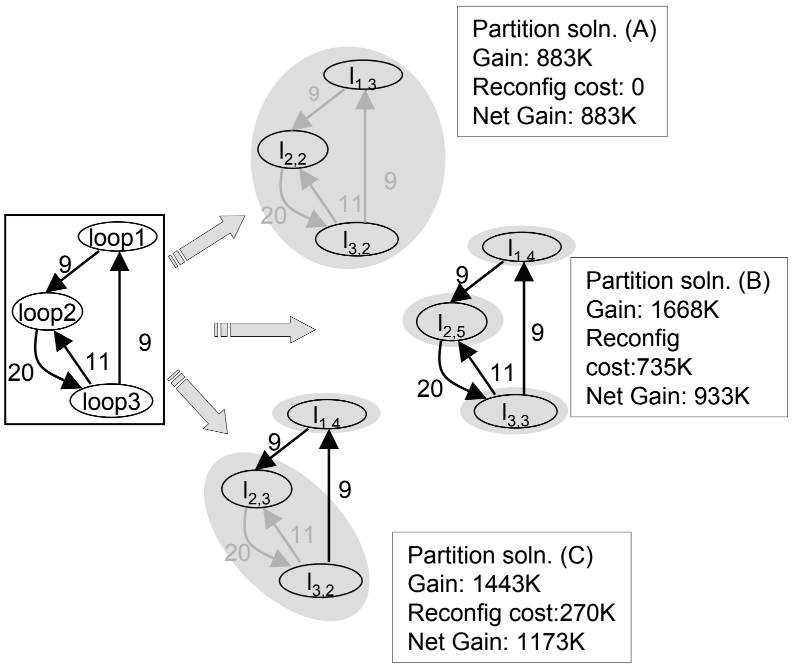
where r is the number of reconfigurations given the partitioning and $s_{i,j}$ is a binary variable equal to 1 if CIS version $l_{i,j}$ is selected and 0 otherwise.

Dynamic reconfiguration through temporal partitioning enlarges the available area for the design by increasing the number of configurations. Therefore, each loop can select better CIS version to be implemented in hardware and better performance gain will be achieved. However, this increase in number of configurations may not result in better overall performance due to the reconfiguration cost. On the other hand, if we minimize the number of configurations, the available area is quite restricted. Consequently, each loop will select its CIS version with smaller area and the performance gain of the application is much smaller, especially when the reconfiguration cost is smaller. Our objective is to maximize the performance gain by selecting an appropriate CIS version for each loop and mapping it into an appropriate configuration.

3.1 Motivating Example

Loop	Version	Area (#AU)	Gain (K cycles)
loop1	1	0	0
	2	257	111
	3	301	160
	4	1612	563
loop2	1	0	0
	2	761	230
	3	1041	387
	4	1321	426
	5	2004	556
loop3	1	0	0
	2	967	493
	3	1249	549

(a) CIS versions for 3 loops



(b) Some partitioning solutions

Figure 4: Motivating Example.

Let us consider an application with three hot loops: `loop1`, `loop2` and `loop3`. Figure 4 (a) shows the performance/silicon area tradeoff of different custom instruction-set versions for each loop. In particular, the table shows the hardware requirement in terms of arithmetic units (AU) and corresponding performance gain in terms of K cycles. For example, `loop3` has three CIS versions. Version 1 of each loop is the software version (without any custom instructions enhancements) with zero area and performance gain. We need to select appropriate CIS versions for the three loops and club them into one or more configurations. Let the hardware area constraint for a single configuration be 2048 AUs. The cost for a single reconfiguration is 15K cycles. The graph on the left-hand side of Figure 4 (b) shows control flow information among the loops for this example. The actual input to our algorithm is the loop trace. We use the graph here (derived from the loop trace) for illustration purposes. We will, however, use a similar graph (called reconfiguration cost graph) later in our temporal partitioning algorithm.

If the system does not support dynamic reconfiguration, the best partitioning solution (solution (A) in Figure 4 (b)) under the hardware area constraint is the selection of version 3 of `loop1`, version 2 of `loop2`, and version 2 of `loop3`. Total performance gain is $160 + 230 + 493 = 883$K cycles and there is no reconfiguration cost.

However, in the presence of dynamic reconfiguration, we can improve the solution. A trivial solution is to put each loop into one configuration (solution (B) in Figure 4 (b)). We can then select the CIS version of a loop with the largest area less than or equal to the area of a configuration: version 4 for `loop1`, version 5 for `loop2` and version 3 for `loop3`. Total performance gain is $563 + 556 + 549 = 1668$K cycles and the total reconfiguration cost is $(20 + 11 + 9 + 9) \times 15 = 735$K cycles. Therefore the resulting net performance gain after subtracting the reconfiguration cost is $1668 - 735 = 933$K cycles. While the net performance gain is better than the case when dynamic configuration is not supported, it is not the optimal solution.

The optimal solution is to put `loop2` and `loop3` into one configuration and `loop1` into a different configuration (solution (C) in Figure 4 (b)). CIS versions 4, 3, and 2 will be selected for `loop1`, `loop2`, and `loop3`, respectively. The performance gain is 1443K cycles, while reconfiguration cost is $(9 + 9) \times 15 = 270$K cycles. Hence, the net performance gain is $1443 - 270 = 1173$K cycles.

4 Partitioning Algorithm

Finding the optimal combination of temporal and spatial partition is a difficult problem. Given N loops, the number of possible configurations is 2^N. However, the number of ways to partition N loops into mutually-exclusive configurations corresponds to the $N + 1^{th}$ Bell number. According to de Brujin [13], asymptotic limits of Bell numbers is $O(e^{N \ln(N)})$.

Our partitioning algorithm needs to makes three choices: (1) optimal number of configurations k, (2) temporal partitioning of the loop kernels into k configurations, and (3) spatial partitioning of the loop kernels in each configuration, i.e., choosing the appropriate custom-instruction set (CIS) version for each loop kernel. Clearly, these choices are inter-dependent. The selection of CIS versions for the loops determines the partitioning solution and vice versa.

4.1 Overview

We propose an iterative algorithm (Algorithm 1) for joint temporal and spatial partitioning of the custom instruction-sets corresponding to the hot loop kernels. The algorithm iterates from a constraint of having exactly 1 configuration (i.e., no reconfiguration) to the upper bound of having $|L|$ configurations where L is the set of hot loops. The solutions (A) and (B) in our motivating example (see Figure 4) represent the two extremes ($k = 1$ and $k = |L|$), while the remaining iterations explore the rest of the design space.

For the iteration with k configurations, we would like to identify the k-way partitioning solution with the optimal net performance gain. Unfortunately, temporal and spatial partitioning are again dependent on each other due to the reconfiguration cost. To break this cycle, we apply a heuristic technique. The heuristic first assumes that we have a continuous area of $k \times MaxA$ available to us where $MaxA$ is the maximum area for a configuration.

Algorithm 1: Iterative Partitioning Algorithm

```
Input: Set of hot loops with custom instruction-set versions: $L$
       Loop Trace: $T$
       Maximum Area of a configuration: $MaxA$
       Reconfiguration Cost: $\rho$

Result: Partition with the best net performance gain

for $k$ = 1 to $|L|$ in steps of 1 do
    $C$ := global_spatial_partition($L$, $k \times MaxA$);
    $P$ := temporal_partition_with_CIS($C$, $T$, $k$);
    $P'$ := temporal_partition_wo_CIS($L$, $T$, $k$);
    $soln$ := local_spatial_partition($L$, $P$, $MaxA$);
    $soln'$ := local_spatial_partition($L$, $P'$, $MaxA$);
    if net_gain($soln'$) $>$ net_gain($soln$) then $soln := soln'$;
    if net_gain($soln$) $>$ net_gain($bestSoln$) then $bestSoln := soln$;
end

return $bestSoln$;
```

The assumption of continuous area allows us to tentatively select optimal CIS versions for the loops in an ideal (but un-realizable) situation where reconfiguration cost is zero. This provides an upper bound on the performance achievable with k configurations. In reality, however, we have k distinct configurations with $MaxA$ area each. So we partition the loop kernels with selected CIS versions into k configurations such that each configuration has *roughly* $MaxA$ area and the reconfiguration cost is minimized. As we break up the continuous area into k distinct areas, some configurations end up being bigger than $MaxA$, while some other configurations are smaller than $MaxA$. To fix this, we have a final patch-up stage that performs spatial partitioning within each configuration to re-distribute $MaxA$ space among the constituent loop kernels.

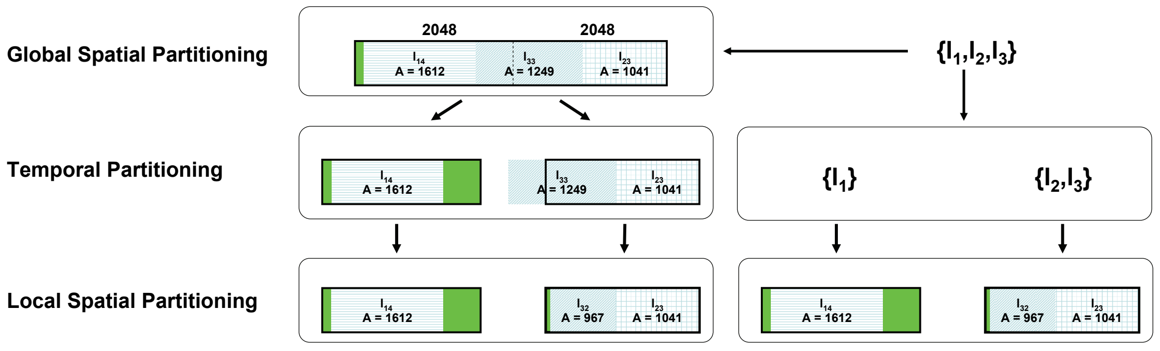


Figure 5: Three phases of iterative partitioning algorithm for number of configurations = 2

Figure 5 illustrates the three phases of the iterative partitioning algorithm corresponding to the iteration with 2 configurations. The input is the three loops in the motivating example and their CIS versions.

The first phase, global_spatial_partition, partitions the area $k \times MaxA$ (where k is the number of configurations for that iteration) among the loops by selecting the CIS versions such that the performance gain is optimal. This phase disregards the reconfiguration cost. It also assumes that a continuous area of size $k \times MaxA$ is available for hardware acceleration of all the loops. We have developed a *dynamic programming* algorithm for this phase. This phase may choose to select the software version for some loops. For our running example, the first phase in Figure 5 chooses CIS versions $l_{1,4}, l_{2,3}, l_{3,3}$.

After the first phase, we have the set of selected CIS versions C for the hot loops. However, we cannot implement this solution as (1) the reconfiguration cost has not been considered, and (2) the loops still need to be partitioned into different configurations. In the second phase temporal_partition_with_CIS, we perform temporal partitioning of the selected loops into k configurations such that the reconfiguration cost is minimized and the partitions are roughly equal in size. This phase returns the partitioning solution P for the set of loops selected for custom instructions enhancements from the first phase.

In the second phase, we also find an alternative partitioning solution P' for the original set of hot loops, i.e., it disregards the results of the first phase. This partitioning, temporal_partition_wo_CIS, only considers the reconfiguration cost and ignores the CIS versions. Partition P is a better choice when performance gain of the CIS versions is high relative to the reconfiguration cost. On the other hand, partition P' is a better choice when the reconfiguration cost is high relative to the performance gain. P and P' complement each other in the search for the best partitioning solution. We model the temporal partitioning as a *k-way weighted graph partitioning problem*, which is well studied [17, 18].

In Figure 5, the left hand side shows the partition P and the right hand side shows the partition P'. For P, the second phase partitions the three loops with selected CIS versions into two configurations: $l_{1,4}$ in the first configuration and $l_{2,3}, l_{3,3}$ in the second configuration. On the other hand, P' simply partitions the three loops based on reconfiguration cost into two configurations. In this example, P and P' return the same temporal partitioning. However, due to the reconfiguration cost, P and P' may be different.

We now have k configurations for each partitioning solution P and P'. The k-way weighted graph partitioning produces partitions with roughly equal size. Therefore for partition P, the area requirement of some of the configurations may exceed the maximum area $MaxA$. Partitioning solution P', on the other hand, does not select any CIS version a-priori. Thus, for each configuration in P and P', the third phase, local_spatial_partition, locally selects the CIS versions for the loops in that configuration to maximize performance gain under area constraint $MaxA$. We again use dynamic programming to perform optimal spatial partitioning for each configuration.

In Figure 5, for partition P, the area requirement of the second configuration exceeds the maximum area budget. Hence phase 3 for this partition replaces CIS version $l_{3,3}$ with $l_{3,2}$. Phase 3 keeps the CIS version for loop l_1 unchanged even though there is additional area available (the green part) as $l_{1,4}$ is the best version for l_1. However, in general, the additional area can lead to the selection of better versions for some loops. The third phase of P' simply selects CIS versions of the loops in each configuration for the first time. Finally, the net performance gains of P and P' are compared to select the best partitioning solution for k configurations.

If the net performance gain of the current solution (with k configurations) is better than the best solution obtained so far (with less than k configurations), we update the best solution. Then we start a new iteration with $k = k+1$. The algorithm terminates when in the current solution, each loop has been assigned its CIS version with the best performance gain. In the worst case, the algorithm runs for $|L|$ iterations. With the motivating example, our algorithm returns the optimal solution, which has two configurations (see Figure 5) and the performance gain is 1173K cycles.

Let us now proceed to describe the spatial and temporal partitioning algorithms.

4.2 Spatial Partitioning

We propose a pseudo-polynomial time dynamic programming algorithm to select the appropriate CIS versions for the loops such that the performance gain is optimal under a hardware area budget. This algorithm is employed in the first phase and the third phase of our iterative solution with different parameters.

Let $G_i(A)$ be the *maximum* performance gain of loops $l_1 \dots l_i$ under an area budget A. Then $G_i(A)$ can be defined recursively.

$$G_i(A) = \max_{\substack{j=1\dots n_i \\ area_{i,j} \leq A}} (gain_{i,j} + G_{i-1}(A - area_{i,j})) \tag{3}$$

That is, given an area A, we explore all possible CIS versions for l_i and choose the one that results in maximum performance gain for loops $l_1 \dots l_i$. The base case for loop l_1 is

$$G_1(A) = \max_{\substack{j=1\dots n_1 \\ area_{1,j} \leq A}} (gain_{1,j}) \tag{4}$$

The maximum performance gain for loops $l_1 \dots l_N$ under area budget $AREA$ then corresponds to $G_N(AREA)$.

Algorithm 2: Spatial Partitioning

```
Input: Set of loops $l_1, l_1, \dots, l_N$ with CIS versions;
    Area constraint: $AREA$

Result: Maximum performance gain

for A = 0 to AREA in steps of $\Delta$ do
    $G_1(A) \leftarrow \max_{\substack{j=1\dots n_1 \\ area_{1,j} \leq A}} (gain_{1,j})$
end
for A = 0 to AREA in steps of $\Delta$ do
    for i=2 to N do
    $G_i(A) \leftarrow \max_{\substack{j=1\dots n_i \\ area_{i,j} \leq A}} (gain_{i,j} + G_{i-1}(A - area_{i,j}))$
end
return $G_N(AREA)$;
```

Algorithm 2 encodes this recursion as a bottom-up dynamical programming algorithm. The step value Δ determines the granularity of area. It is chosen as the greatest common divisor of the area requirements of all CIS versions and AREA. The time complexity of this algorithm is $O(N \times \frac{Area}{\Delta} \times x)$ where $x = max_{i=1\dots N}(n_i)$.

4.3 Temporal Partitioning

We map our temporal partitioning problem to k-way weighted graph partitioning problem. The k-way weighted graph partitioning problem is defined as follows. Given an undirected graph $G = (V, E)$ with weights both on the vertices and the edges, partition V into k subsets $V_1, V_2, \ldots V_k$ such that $V_i \bigcap V_j = \emptyset$ for $i \neq j$, $\bigcup_i V_i = V$, the sum of the vertex-weights in each subset is roughly equal, and the sum of the edge-weights whose incident vertices belong to different subsets (edge-cut weights) is minimized.

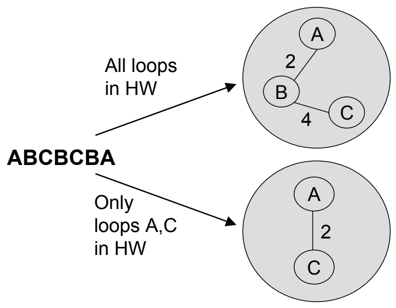


Figure 6: Reconfiguration cost graph from loop trace

We generate a *Reconfiguration Cost Graph (RCG)* from the loop trace to model our temporal partitioning problem as a k-way weighted graph partitioning problem. After the first phase, we have tentatively selected CIS versions for the loops. Each vertex in the RCG represents a hot loop selected for hardware acceleration in the first phase. In other words, we do not consider the loops for which the first phase selects software-only version. Given a vertex v associated with loop l, we assign the area of the CIS version selected for l as the weight of the vertex v. When CIS versions from the first phase are ignored (in temporal_partition_wo_CIS), the RCG includes all the loops and we assume unit hardware cost for each vertex.

The edge weight between vertex v (corresponding to loop l) and v' (corresponding to loop l') is defined as the reconfiguration cost between loop l and loop l' if they are mapped to two different configurations. The edge between v and v' exists if and only if control can flow from loop l to l' or l' to l without passing through any other hot loops. The weight on the edge between v and v' represents the number of times control flows from loop l to l' and l' to l (without passing through any other selected loop). This weight can be derived from the loop trace as follows. If we eliminate the software-only loops from the loop trace, then the weight is the the number of times the string ll' and $l'l$ appear in the loop trace. The time complexity of creating RCG is linear in the size of the hot loop trace.

Figure 6 shows an example of RCG generation from the loop trace. It shows a loop trace $ABCBCBA$ of three hot loops A, B, C. If all the loops are selected to be placed in hardware, then there are 2 reconfiguration points between loops A and B if they are partitioned into different configurations. Similarly, there are 4 reconfiguration points between loops B and C if they are partitioned into different configurations. However, there are no reconfiguration points between loops A and C directly as the control transfers between them always pass through B. However, if we choose to implement B in software in the first phase, then B is eliminated from the RCG. In this case, there are 2 reconfiguration points between loops A and C if they are partitioned into different configurations.

The objective now is to partition the RCG into k configurations such that the configurations have roughly equal area (or the configurations have roughly equal number of loops when area is ignored) and the reconfiguration cost (edge-cut weights) is minimized. If the configurations have roughly equal area, then the loops have higher probability of retaining the optimal CIS versions selected in the first phase regardless of the third phase. As a result, total performance gain (excluding reconfiguration cost) after the third phase is expected to be near the optimal performance gain in the first phase. The rationale behind having roughly equal number of loops in each configuration when CIS versions are ignored (by assigning unit cost to each vertex in the RCG), is to create a balanced temporal partition. It ensures that equal number of loops compete for each configuration space during subsequent spatial partitioning.

We use multilevel k-way partitioning scheme by Karypis and Kumar [18]. The multilevel partitioning scheme consists of three phases: coarsening phase, partitioning phase and uncoarsening phase. During coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$, each with fewer vertices, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_i| < |V_{i-1}|$. The coarsening phase ends when the coarsest graph G_m has a small number of vertices or the reduction in the size of successively coarser graph becomes too small. Then, the partitioning phase computes a k-way partitioning P_m of the coarse graph $G_m = (V_m, E_m)$ such that each partition contains roughly $|V_0|/k$ vertex weight of the original graph. The k-way partitioning of G_m is computed using multilevel bisection algorithm [17]. During the uncoarsening phase, the partitioning P_m of the coarser graph G_m is projected back to the original graph by going through the graphs $G_(m-1), G_(m-2), ..., G_1$. At each intermediate level, the partitioning is refined based on Kernighan-Lin [21] partitioning algorithm and their variants.

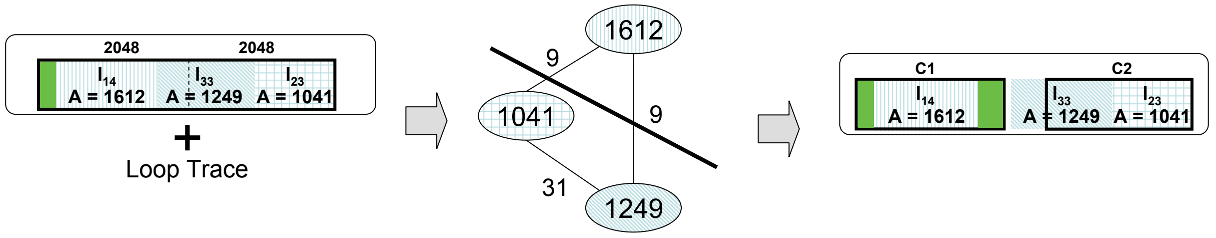


Figure 7: Modeling the temporal partitioning problem as k-way graph partitioning problem.

Figure 7 shows how the temporal partitioning problem is solved by modeling it as a k-way weighted graph partitioning problem for our running example. The edge weights of the RCG are generated from the loop trace. The area of the CIS version selected for each loop in the global partitioning phase is assigned as the weight of the corresponding vertex. Now we perform 2-way partitioning of this graph with minimum edge-cut weights and roughly equal vertex weights in each partition. This gives us the configurations with roughly equal area while minimizing reconfiguration cost.

5 Experimental Evaluation

To compare the accuracy and scalability of our iterative partitioning algorithm, we have developed two other algorithms — exhaustive search and greedy search. The results of the two algorithms are compared with our proposed algorithm in two different sets of experiments. In the first set of experiments, we run the three algorithms using synthetic input to evaluate the scalability and efficiency of the algorithms. We generate input data with 5 to 100 hot loops for this set of experiments. In the second set of experiments, we conduct a case study of the JPEG application with custom instructions implemented on a commercial platform Stretch 5 [28] that supports runtime reconfiguration.

Exhaustive Search The exhaustive search algorithm computes the optimal results by evaluating all possible temporal and spatial partitioning. We use the algorithm described in Kreher and Stinson [22] to enumerate all possible partitioning solutions. We then find the optimal implementation of each configuration in each partitioning solution by choosing CIS versions of the constituent loops through our spatial partitioning algorithm. The net gain of each enumerated partition is then estimated through a brute force computation of the reconfiguration cost by traversing the loop trace. The partition with the maximum net performance gain is then the optimal solution. Our experiments show that the exhaustive search algorithm cannot scale with increasing number of hot loops.

Greedy Search The greedy search algorithm (see Algorithm 3) constructs a solution by building one configuration at a time until no more CIS version can be added without causing a degradation in performance. The input is the set of hot loops with custom instruction-set versions L, loop trace T, area constraint $MaxA$, and single reconfiguration cost ρ. A solution consists of one or more configurations. The algorithm begins with an empty solution and an empty current configuration.

In each iteration, we pre-compute a reconfiguration cost array C. For any unselected loop l_i, the array C gives the expected additional reconfiguration cost if l_i is added to the current configuration. Given C, the current solution and the current configuration, we can now compute the expected performance gain of each CIS version if we add it to the current configuration. For CIS version $l_{i,j}$, this expected performance gain is estimated by subtracting from $gain_{i,j}$, the additional reconfiguration cost for loop l_i (available from array C). We now select the CIS version with the maximum expected *positive* performance gain that can be added to the current configuration without violating the area constraint. The selected CIS version is then added to the current configuration. All the other CIS versions of the same loop are subsequently removed from the set L.

In the event that no CIS version can be selected, there are two possibilities. The first possibility is that no more loops can be added to the current configuration without violating the area constraint ($current$ configuration is not empty in Algorithm 3). In this case, we update the solution with the current configuration and re-start the process

Algorithm 3: Greedy Search Algorithm

```
Input: Set of hot loops with custom instructions: $L$
       Loop Trace: $T$
       Maximum Area of a configuration: MaxA
       Reconfiguration Cost: $\rho$

Result: Partitioning solution

current := new_configuration();
continue := $true$;
while continue = true do
    $C$ := compute_reconfig_cost_for_unselected_loops($L$, $T$, solution, current);
    $l_{i,j}$ := select_most_profitable_feasible_CIS($C$, $L$, $MaxA$, current);
    if $l_{i,j}$ is not found then
        if current is not empty then
            update solution by adding current;
            current := new_configuration();
        else
            continue := $false$;
        end
    else
        update current with $l_{i,j}$;
        remove from L all CIS versions of loop $l_i$;
    end
end

return solution
```

of selecting CIS versions with an empty configuration. The second possibility is that no more loops can be added to the current solution without decreasing its net performance gain ($current$ configuration is empty, i.e., we are trying to select the CIS version under maximum area constraint). In this case, the algorithm stops and returns the solution built so far.

5.1 Efficiency and Scalability of Algorithms

For this set of experiments, we generate synthetic inputs with number of hot loops ranging from 5 to 100. The number of CIS versions for each loop is generated randomly and ranges between 1 to 10. The performance gain of each CIS version ranges between $1,000$ to $10,000$ time units. The hardware area is between 1 to 100 units. The performance gain increases with hardware area for each loop.

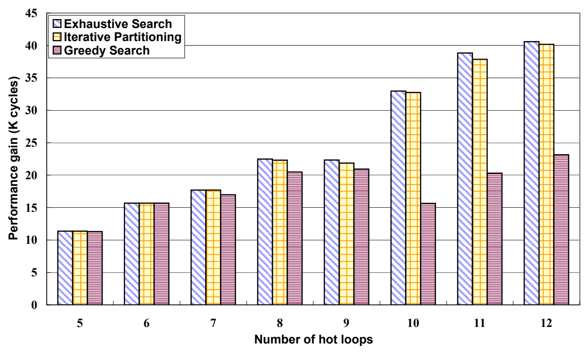
The reconfiguration costs between two loops, if they are assigned to different configurations, are generated randomly. They are in the range 0 to $maxCost$ where $maxCost$ is approximately 40-50% of the average performance gain of all the CIS versions of all the loops $\frac{\sum_{i=1}^{N}\sum_{j=1}^{n_i} gain_{i,j}}{\sum_{i=1}^{N} n_i}$. The value of $maxCost$ ensures that the reconfiguration cost is neither too high nor too low. Both the extremes reduce the search space considerably. If the reconfiguration cost is too high, we should only consider partitions with a small number of configurations. If the reconfiguration cost is too low, then the solution is to simply select the CIS version with the highest speedup for each loop and construct as many configurations as required. The hardware area constraint $MaxA$ is approximately

20-30% of the sum of the average area requirements of the CIS versions of all the loops $\sum_{i=1}^{N} \frac{\sum_{j=1}^{n_i} area_{i,j}}{n_i}$. This ensures that all the loops with their CIS versions cannot fit under the area constraint.

Number of Hot Loops	Running time (sec) Exhaustive search	Greedy search	Iterative partitioning
5	0.26	0.01	0.07
6	1.34	0.02	0.07
7	7.84	0.01	0.07
8	43.91	0.01	0.09
9	283.22	0.04	0.07
10	1788.20	0.01	0.11
11	12604.33	0.01	0.13
12	86338.37	0.01	0.15
20	N.A.	0.02	0.48
40	N.A.	0.04	4.30
60	N.A.	0.07	18.25
80	N.A.	0.11	55.61
100	N.A.	0.16	118.76

Table 1: Running time of the algorithms for synthetic input.

Table 1 shows the running times of the three algorithms for synthetic input with different number of hot loops. The running time of the exhaustive search algorithm, while relatively small with smaller number of loops, increases by almost an order of magnitude each time one more loop is considered. The results of exhaustive search for more than 12 loops cannot be obtained even after waiting for a day. On the other hand, although iterative partitioning algorithm is slower than greedy search in general, its running time is quite acceptable (less than 2 minutes). This demonstrates the scalability of our approach. Moreover, iterative partitioning generates much better quality solutions compared to greedy search as presented in the following.



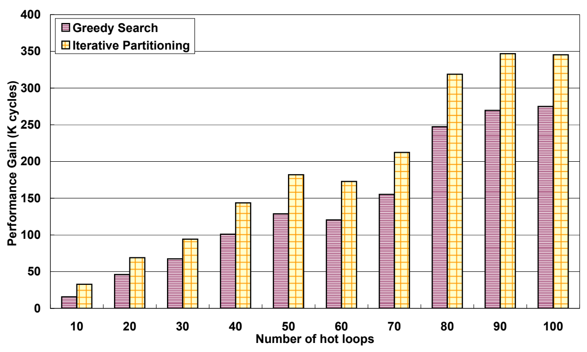


Figure 8: Comparison of the quality of the solutions returned by the algorithms for synthetic input. Exhaustive search fails to return any solution with more than 12 hot loops.

Figure 8(a) compares the quality of the solutions returned by the three different algorithms with number of hot loops varying from 5 to 12. The performance gain obtained using our approach is close to the optimal gain obtained with exhaustive search while greedy search falls far behind. Figure 8(b) presents the comparison between the performance gain of iterative partitioning and greedy search for input with more than 12 hot loops. We cannot report the results for exhaustive search algorithm here as exhaustive search fails to return any solution for more than 12 loops (even after running for more than a day). The iterative algorithm consistently outperforms greedy search in terms of performance gain by a factor of 1.26 to 2.09.

5.2 Case Study of JPEG Application

We present a case study of the JPEG image compression algorithm. In this study, we envision a scenario in which an image is encoded and then decoded subsequently. The hot loops are profiled and the loop trace is generated using an in-house tool based on OpenImpact [1], an open source compiler. The profiling works in two phases. The timing information of each loop is collected by inserting appropriate time stamps at the entry and exit points of the loops. After the first pass, loops which take up more than 1% of the computation time can be detected. During the second pass, the compiler inserts appropriate code to capture the entry point of the hot loops. The resulting application, when executed, generates a trace of the hot loops.

```
for (i = 0; i < num_cols; i++)
{
  r = GETJSAMPLE( in[RGB_RED] );
  g = GETJSAMPLE( in[RGB_GREEN] );
  b = GETJSAMPLE( in [RGB_BLUE] );
  in += RGB_PIXELSIZE;
  y[i]=(JSAMPLE)((ctab[r + R_Y] + tab[g + G_Y]
                + ctab[b + B_Y]) >> SCALEBITS);
  cb[i]=(JSAMPLE)((ctab[r + R_CB] + ctab[g + G_CB]
                + ctab[b + B_CB]) >> SCALEBITS);
  cr[i]=(JSAMPLE)((ctab[r + R_CR] + ctab[g + G_CR]
                + ctab[b + B_CR]) >> SCALEBITS);
}
```

(a) Original Loop

```
SE_FUNC void rgb2ycc(WRA &A, WRB &B)
{
  se_uint<8> r[10],g[10],b[10];
  se_uint<8> y[10],cb[10],cr[10];
  int i,j;
  /* Unpack A, B to RGB data */
  for (i = 0; i < 5; i++) {
    j = i * 3 * 8;
    r[i] = A(j+7,j);
    r[5+i] = B(j+7,j);
    g[i] = A(j+15,j+8);
    g[5+i] = B(j+15,j+8);
    b[i] = A(j+23,j+16);
    b[5+i] = B(j+23,j+16);
  }
  /* Converting 10 pixels */
  for (i = 0; i < 10; i++) {
  //loop body: r[i],g[i],b[i] instead
                of r,g,b
  }
  /* Pack y, cb, cr to A, B */
  A = (cr[4],cb[4],y[4],cr[3],cb[3],
       y[3],cr[2],cb[2],y[2],cr[1],
       cb[1],y[1],cr[0],cb[0],y[0]);
  B = (cr[9],cb[9],y[9],cr[8],cb[8],
       y[8],cr[7],cb[7],y[7],cr[6],
       cb[6],y[6],cr[5],cb[5],y[5]);
}
```

(b) rgb2ycc Custom Instruction

```
WRGET0INIT(0,in); /* GET stream from in */
WRGET0INIT1();
for(i =0; i < num_cols/10; i++)
{
  char out[30];
  WRAGET0I(&A,15); /* A, B GET 10 pixels */
  WRBGET0I(&B,15);
  rgb2ycc(&A, &B); /* Call rgb2ycc instruction */
  WRPUTINIT(0, out); /* PUT stream to out */
  WRPUTI(A,15); /* Put result from A and B */
  WRPUTI(B, 15);
  WRPUTFLUSH();
  .../* Put out to y,cr,cb */
}
```

(c) Loop using Custom Instructions

Figure 9: An example of custom instruction for Stretch processor.

Our loop profiler identifies more than 15 hot loops for the JPEG application. For our experimental purposes, we select the top 10 loops and manually generate custom instruction set versions for each loop on the Stretch S5 platform [28]. Figure 9 shows an example of exploiting custom instructions on Stretch processor for performance enhancement of an application. The original loop is shown on the left of the figure. It performs conversion from RGB color space to YCbCr color space. The original loop converts one pixel at a time. However, the main benefit of custom instructions in Stretch comes from exploiting instruction-level parallelism. Therefore the original loop is unrolled X times ($X = 10$ in our example) to expose more instruction-level parallelism. We can achieve different custom instruction versions (or CIS versions) by changing the unroll factor. The higher unroll factor results in larger hardware area requirement and better performance gain.

Figure 9(b) shows an example of using Stretch C language to define a custom instruction corresponding to the loop body (unrolled 10 times). Stretch C is a variant of C language that allows the designer to define custom instructions. The Stretch C compiler can automatically synthesize the custom instructions into the fabric. The custom instruction, called `rgb2ycc`, has two 128-bit wide registers, A and B, as in-out arguments. A and B contain 10 RGB pixels that will be converted to YCC pixels. First, input data in A and B are unpacked to the local RGB pixel variables. Then RGB pixels are converted to YCC pixels through a `for` loop. The Stretch C compiler, while synthesizing the custom instruction into hardware, will unroll this `for` loop within the custom instruction. That is, the 10 pixels will be converted in parallel in hardware. Finally, YCC pixels are packed into A and B registers as the output. After the new custom instruction is defined, we have to change the source code of the original loop to use the newly defined custom instruction (see Figure 9(c)). The wide register arguments of `rgb2ycc`, A and B, get 10 RGB pixels at a time from stream in. However, constant tables (such as `ctab`) used in `rgb2ycc` can be hard code into the fabric. Finally, we execute `rgb2ycc` and extract the output from A and B. For number of RGB pixels less than 10, we perform normal computation without custom instruction.

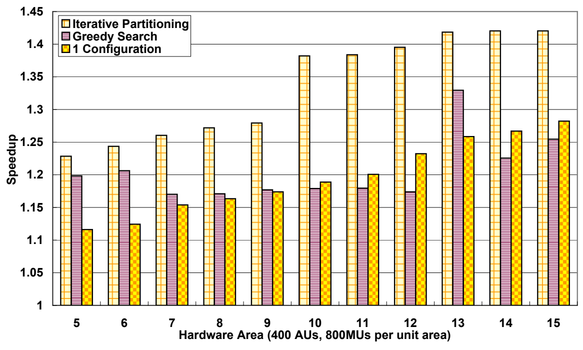
Loop ID	(#AUs, #MUs, Gain (K cycles))		
0	(2249, 4096, 32)		
1	(1612, 2880, 563)	(257, 704, 111)	(389, 2176, 254)
2	(2004, 6272, 556)	(1041, 2048, 387)	(1321, 2592, 426)
	(761, 1504, 230)		
3	(207, 0, 493)	(424, 2, 549)	
4	(2515, 1536, 1094)		
5	(1530, 3584, 1669)	(1300, 3584, 1643)	
6	(981, 4480, 1095)	(491, 2240, 739)	(393, 1792, 590)
7	(1059, 2880, 511)		
8	(1089, 2880, 910)		
9	(1764, 1280, 194)	(1114, 768, 188)	

Table 2: CIS versions for JPEG application.

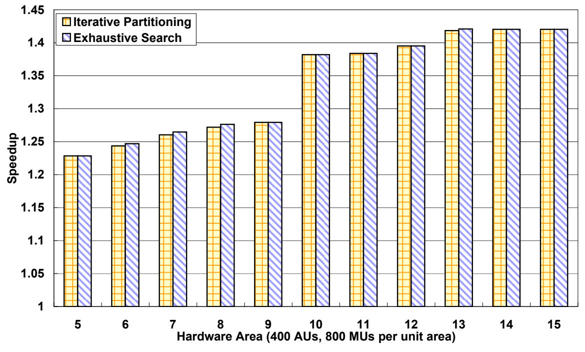
The profiler in Stretch IDE can now provide us the performance gain and hardware area of the CIS versions of each loop. Table 2 shows the various CIS versions for each loop and their respective area requirements and performance gain. It is worth noting that the performance gain of the CIS versions do not commensurate with area increase in general. For example, loop 0 takes up 2249 arithmetic units and 4096 multiplier units but only gives 32K cycles of performance gain. In contrast, the CIS versions of loop 3 use far less area but give much better performance. This is because the parallelism that can be exploited varies from one loop to another.

The configuration time of the whole fabric of Stretch development board, which includes 4096 4-bit arithmetic units (AUs) and 8192 4-bit $\times$ 8-bit multiplier units (MUs) is approximately 100μs. Given that the CPU runs at 300MHz, the configuration time translates to roughly 30K CPU cycles. We define *one hardware area unit* to be a tuple of 400 AUs and 800 MUs. Since the configuration time is proportional to the size of the fabric, configuration time of one hardware area unit is approximately 3K CPU cycles. By scaling the configuration time according to the fabric size, we can easily compute the configuration time for any fabric size.

It is possible to fit CIS versions of all the hot loops from our JPEG application in a suitably-sized fabric. For our experimental purposes, we assume that the hardware area constraint varies from one hardware area unit to 20-30% of the sum of maximum hardware area for all the loops ($5-15$ hardware units for JPEG application). This will lead to the necessity of dynamic reconfiguration. We run all the three algorithms (exhaustive search, greedy search and iterative partitioning) under these different area constraints. Our profiling data indicates that the application takes around 20 million cycles on Stretch CPU without custom instructions enhancements. It should be noted, however, that the speedup we obtain for a particular application depends on the quality of the custom instructions generated in the first place. Our focus in this experiment is to evaluate our proposed algorithm in comparison with greedy search and exhaustive search. That is, we are only concerned about comparing the performance gain obtained using the different algorithms starting with the same set of CIS versions. Our results show that the our proposed algorithm is always optimal or near-optimal and produces much better results than greedy search most of the time.



(a) Comparison of iterative partitioning, greedy search, and static configuration (1 configuration).



(b) Comparison of exhaustive search and iterative partitioning.

Figure 10: Comparison of the quality of solutions for the case study of JPEG application.

In Figure 10(a), we evaluate the performance gain possible if dynamic reconfiguration is exploited. We compare the performance gain obtained using iterative partitioning and greedy search with the case when no reconfiguration is allowed. Clearly, iterative partitioning and greedy search can choose to use more than one configuration. However, the algorithm for static configuration only performs spatial partitioning. If we compare the results of our algorithm with that of static configuration, the advantage of exploiting dynamic reconfiguration decreases as the hardware area increases. This is to be expected, as more custom instructions can fit into the larger area to gain suitable speedup, thus reducing the need to virtualize hardware through run-time reconfiguration. The graph demonstrates that our algorithm increases the performance gain over and above static configuration by at least 34% and as much as 78%.

On the other hand, the simple heuristic of the greedy search algorithm fails to achieve substantial performance gain over static configuration. Often, the greedy search performs as good as the static configuration, and in some cases, even worse. Our proposed iterative partitioning algorithm always performs better than the greedy search, being at least 14% and as much as 91% better than greedy search.

Figure 10(b) measures how closely our proposed algorithm approximates the optimal results obtained through exhaustive search. The graph shows that our algorithm returns solution that coincides with the optimal solution most of the cases, and falls short of the optimal by at most 1% in the remaining cases.

6 Related Works

Custom instruction selection for an application usually consists of two steps. The initial step identifies a large set of candidate patterns from the program's dataflow graph and their frequencies via profiling [4, 5, 11, 12, 19, 30, 31]. Given this library of patterns, the second step selects a subset to maximize the performance under different design constraints. Various approaches proposed for this step include dynamic programming [4], 0-1 Knapsack [12], greedy heuristic [10, 11, 29], and ILP [23, 29]. However, none of these approaches targets applications exploiting dynamic reconfiguration of custom functional units. Recently, [7] proposed rotating instruction set processing platform that can select custom instructions at runtime.

The major part of the research on temporal partitioning comes from the reconfigurable computing community. Usually, the partitioning is done at the task-level [6, 9, 20], though there exist some exceptions. Li et al. [24] partition at the loop level while Purna and Bhatia [27] perform partitions on the data flow graph. When directed acyclic task graphs are used as input, computing reconfiguration costs becomes simple. For example, Banerjee's work [6] is able to reduce the partitioning problem as a scheduling problem because task graphs are used as input. In contrast, it is non-trivial to obtain the reconfiguration cost at the granularity of loops. It should be noted that while Purna and Bhatia's work [27] partitions at the finer granularity of functions and operators, their work uses directed acyclic data flow graph as input as well.

Bondalapati and Prasanna [8] focus on mapping the statements within a loop into configurations to obtain a configuration sequence that gives the least execution time. While dynamic reconfiguration is used as well, their work focuses on intra-loop selection of configurations, i.e., their work operates on one loop only. Our work is different because not only do we consider multiple possible custom instructions set versions per loop, our algorithm allows for multiple loops within a configuration and some loops may remain in software. As such, our work is different from projects that explore the design space for individual loops such as [8].

Hardnett et al. [16] form a framework in which the dynamically reconfigurable architectural design space may be explored for specific applications. In particular, the register allocation problem is adapted to assign reconfigurable units to different custom instructions. An instruction scheduling algorithm for the custom instructions is implemented to minimize overall latency. While their architecture employs dynamically reconfigurable functional units, our work is differentiated from theirs in two specific areas. First, their custom instructions do not share the same functional unit, i.e., no spatial partitioning is required. Secondly, their work does not address the problem of reconfiguration cost directly. Rather, custom instructions are de-selected to relieve resource pressure rather than optimizing overall performance.

The work most related to our work is that of Li et al. [24] in which the Nimble compiler is implemented. Their work focuses on selecting loops from an application for hardware implementation while aiming to reduce dynamic reconfiguration overhead. Their work only considers a single loop in one configuration and they did not consider global reconfiguration cost when selecting loops to put in hardware.

7 Conclusions

We have presented an algorithm to exploit dynamically configurable custom functional units for optimal performance gain. Given an input application, the algorithm selects and partitions the custom instructions corresponding to the loop kernels into different configurations that are reconfigured at run-time. The experimental results show that our algorithm is highly scalable while producing optimal or near-optimal performance gain.

We can extend our work to consider configuration prefetching and partial reconfiguration. Previous work [25] has indicated that how early a configuration can be prefetched depends on several factors, including the relationship between the configurations and the placement of the prefetch instruction. Closely related to configuration prefetching is partial reconfiguration, which allows execution and reconfiguration of the fabric in parallel. In the future, we plan to extend our framework to handle these non-trivial issues.

8 Acknowledgments

This work was partially supported by NUS research project R252-000-292-112 and A*STAR SERC EHS-II project R-252-000-258-305.

References

[1] OpenIMPACT Compiler. `http://www.gelato.uiuc.edu/`.

[2] ARC International. Customizing a soft microprocessor core, 2001.

[3] J. M. Arnold. S5: The architecture and development flow of a software configurable processor. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, pages 121–128, 2005.

[4] M. Arnold and H. Corporaal. Designing domain-specific processors. In *Proceedings of the 9th ACM International Symposium on Hardware/Software Codesign (CODES)*, pages 61–66, 2001.

[5] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th ACM/IEEE Design Automation Conference (DAC)*, pages 256–261, 2003.

[6] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *Proceedings of the 42nd ACM/IEEE Design Automation Conference (DAC)*, pages 335–340, 2005.

[7] L. Bauer, M. Shafique, S. Kramer, and J. Henkel. RISPP: Rotating instruction set processing platform. In *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC)*, pages 791–796, 2007.

[8] K. Bondalapati and V. K. Prasanna. Mapping loops onto reconfigurable architectures. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 268–277, 1998.

[9] K. S. Chatha and R. Vemuri. Hardware-software codesign for dynamically reconfigurable architectures. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 175–184, 1999.

[10] N. Cheung, S. Parameswaran, and J. Henkel. INSIDE: INstruction Selection/Identification & Design Exploration for extensible processors. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 291–297, 2002.

[11] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 129–140, 2003.

[12] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 183–189, 2004.

[13] N. G. de Bruijn. *Asymptotic Methods in Analysis*. Dover Publications, 1981.

[14] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 203–213, 2000.

[15] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[16] C. Hardnett, K. V. Palem, and Y. Chobe. Compiler optimization of embedded applications for an adaptive SoC architecture. In *Proceedings of the ACM International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 312–322, 2006.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[18] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[19] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):605–627, 2002.

[20] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. In *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC)*, pages 616–622, 1999.

[21] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. 49(2):291–307, 1970.

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

[23] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 649–654, 2002.

[24] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the 37th ACM/IEEE Design Automation Conference (DAC)*, pages 507–512, 2000.

[25] Z. Li and S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Proceedings of the ACM/SIGDA 10th International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 187–195, 2002.

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

[27] K. M. G. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers*, 48(6):579–590, 1999.

[28] Stretch Inc. Stretch S5530 software configurable processor.

[29] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of the 41st ACM/IEEE Design Automation Conference (DAC)*, pages 723–728, 2004.

[30] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 69–78, 2004.

[31] P. Yu and T. Mitra. Disjoint pattern enumeration for custom instructions identification. In *Proceedings of the 17th IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 273–278, 2007.