Shared Reconfigurable Fabric for Multi-core Customization

Liang Chen, Tulika Mitra
Department of Computer Science
National University of Singapore
{chenliang,tulika}@comp.nus.edu.sg

ABSTRACT

Processor customization in the form of application specific instructions can provide significant power and performance boost to an embedded application while maintaining high flexibility. The emergence of multi-core architectures opens up the possibility of creating an application-specific heterogeneous computing platform by customizing a set of homogeneous cores. We propose a multi-core architecture where the cores share a reconfigurable fabric that accommodates the custom instructions. We develop an efficient algorithm that exploits this shared fabric through customization and runtime reconfiguration to minimize the execution time of multi-threaded applications. Experimental results reveal that shared reconfigurable fabric helps applications achieve substantial speedup compared to per-core private fabrics.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Algorithm, Performance, Design

Keywords

Shared reconfigurable logic, multi-core.

1. INTRODUCTION

Computing systems have made an irreversible transition towards parallel architectures with the emergence of multi-cores. At the same time, we are witnessing the development of a very dynamic and diverse landscape of embedded applications especially in the consumer electronics space. A multi-core chip fabricated with a set of identical cores is fundamentally at odds with the diversity of the applications. At the other end of the spectrum, multiprocessor system-on-chip (MPSoC) platforms are being increasingly deployed in high-performance embedded systems. MPSoCs employ heterogeneous processing elements (e.g., general purpose processors, DSPs, application-specific hardware accelerators) to construct a system that perfectly matches the application requirements. However, system integration and lack of flexibility are some of the major challenges faced by the designers in creating platforms with such disparate architectures.

Extensible processor cores, where the existing base ISA can be enhanced with application-specific custom instructions, are emerging as promising alternative in this context. A custom instruction encapsulates frequently executed computational pattern. Custom instructions are implemented as hardwired datapaths in the existing processor core and help improve the power and performance of the application. A heterogenous multi-core may consist of a number of extensible processor cores, where each core has been customized [8, 9]. As all the cores share the same base ISA, application development on such platforms is relatively straightforward.

Custom instructions can be implemented either in ASIC (e.g., Xtensa [3]) or in reconfigurable logic (e.g., Stretch [4]). Reconfigurable logic provides a flexible solution where the custom instructions can be changed for different applications. Moreover, the custom instructions implemented in the fabric can change even within the lifetime of an application under software directives. Of course, this virtualization of the fabric comes at the cost of reconfiguration delay.

A straightforward extension of extensible processors to multi-cores will be to couple each core with its own reconfigurable fabric [1]. However, for multi-threaded applications with pronounced imbalance in execution time and custom instructions requirement among the threads, this fixed partitioning of the resource is not an ideal solution. We propose a multi-core architecture where the extensible cores can share a runtime reconfigurable fabric as shown in Figure 1. By sharing the resource among multiple cores, reconfigurable logic can be more efficiently integrated (at far less power and area cost) into multi-core architectures.

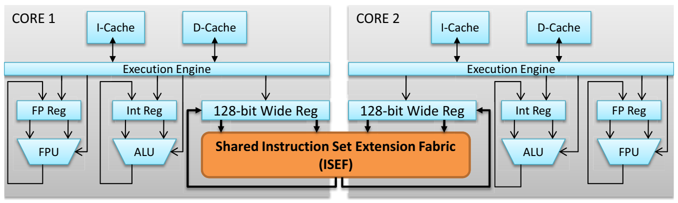

Figure 1: Proposed multi-core architecture with shared reconfigurable fabric.

To exploit this architecture, one needs to select the appropriate set of custom instructions and partition them into different configurations so as to maximize the performance of a multi-threaded application. This is a complex optimization problem. In this paper we first provide an optimal solution for temporal and spatial partitioning of the custom instructions. The optimal solution has limited scalability due to its high computational complexity. Hence we propose an iterative refinement algorithm that quickly attains good quality solution. We thoroughly evaluate our technique with real embedded applications. The evaluation confirms that sharing reconfigurable fabric among the cores leads to far superior solutions compared to per-core dedicated fabric.

Related Work. Multiple extensible cores sharing reconfigurable fabric is a relatively unexplored research direction. ReMAP (Reconfigurable Multicore Acceleration and Parallelization) [10] is a heterogeneous multi-core architecture where the threads share a common reconfigurable fabric in a time-multiplexed, round-robin fashion. The architecture supports limited spatial sharing as the fabric is strictly divided up into equal partitions. Shared reconfigurable coprocessor has been proposed in [2] to improve the throughput of multiple processes concurrently executing on a multi-core system. The focus is on time-multiplexed sharing of the same physical kernel by multiple processes while maintaining process isolation. In both [10] and [2], the reconfigurable fabric is loosely coupled with the cores, whereas in our architecture, it is integrated in the processor datapath. Finally [1] investigates the synergy between multi-core processors and rISE — an architecture where reconfigurable device is used to implement the custom instructions. However, they use dedicated reconfigurable logic per core. Most importantly, none of these works provide design space exploration algorithm to exploit the architecture. For single-core extensible processor with reconfigurable logic, [5] has proposed an algorithm for temporal and spatial partitioning of the custom instructions. However, this algorithm cannot be directly extended to multi-threaded applications that require optimizing the critical path.

2. ARCHITECTURE

Our proposed architecture with shared reconfigurable fabric is a multi-core version of the commercial Stretch processor [4] with minimal modification as shown in Figure 1. Stretch processor incorporates Xtensa processor [3] and the Stretch Instruction Set Extension Fabric (ISEF). The ISEF is software-configurable datapath based on programmable logic. It consists of arithmetic/logic elements and multiplier elements interlinked in a programmable routing fabric. This configurable fabric acts as a functional unit to the processor and resides alongside other traditional functional units such as ALU and FPU. A set of programmer defined custom instructions are implemented in this fabric.

We propose a multi-core architecture where instead of allocating private ISEF for each core, we let a cluster of cores (2 or 4) share a larger ISEF. The idea of sharing resources between neighboring cores is inspired by the conjoined-core chip multiprocessing [7] where the cores share instruction cache, data cache, and FPU. The interface between the processor core and the ISEF is through 32-entry 128-bit wide register files per core. The architecture supports dedicated load/store operation to move data between memory and the wide register files. We allow the shared ISEF to write to the wide register file of each core. A simple round-robin arbiter ensures that only one core can issue a custom instruction to the shared logic every ISEF cycle (note that the ISEF cycle may be multiple of the processor cycle).

A distinguishing aspect of ISEF is that it is run-time configurable. If the computation resource requirement of the custom instructions exceeds the capacity of ISEF, then the instructions can be partitioned into different configurations. Stretch has two mechanisms for managing ISEF configurations — software-directed configuration pre-loading or hardware-assisted reconfiguration. In hardware-assisted reconfiguration, when a custom instruction is issued, the hardware checks and loads the corresponding configuration into the ISEF if it is not already present. As we allow multiple threads to share a configuration, it is possible that a thread might move ahead and issue instructions from the next configuration while other threads are still issuing instructions from the current configuration. This would result in repeated swapping in and out of the configurations. To avoid this problem, we only allow software-directed loading of new configuration. A new configuration is loaded when all the threads have completed using current configuration.

3. PROBLEM DEFINITION

Our architecture is a multi-core system with N cores where all the cores share a reconfigurable fabric (RF). We will use the more generic term reconfigurable fabric instead of ISEF for the rest of the paper. Let $AREA$ be the area of the shared RF and ρ be the reconfiguration latency.

We assume a multi-threaded application with at most N threads running on this multi-core system, i.e., at most one thread is mapped to each core. A thread T_i is modeled as a sequence of n_i tasks $T_{i,1} \ldots T_{i,j} \ldots T_{i,n_i}$. Note that our technique is *not* restricted to linear chain of tasks per thread. If a thread is modeled as a task graph, the tasks can be scheduled through a topological sort of the task graph that respects the dependencies among the tasks. The resulting linear schedule is used as input to our technique. Moreover, it is easy to model applications with pipelined parallelism (e.g., streaming application). Each pipeline stage of the application can be modeled as a thread that maps to a core. All the tasks corresponding to a pipeline stage can be scheduled to create a sequence of tasks for that thread.

Each task is associated with multiple custom instruction set (CIS) versions. A CIS version consists of a set of custom instructions extracted from the corresponding task. The CIS versions are generated according to the tradeoff between area and execution time. Let $\{c^0_{i,j}, \ldots, c^{m_{i,j}}_{i,j}\}$ denote the set of possible CIS versions for task $T_{i,j}$. In addition, let $t^k_{i,j}$ and $a^k_{i,j}$ denote the execution time and area requirement of the CIS version $c^k_{i,j}$. We assume $c^0_{i,j}$ corresponds to the completely software implementation of the task, i.e., $a^0_{i,j} = 0$. That is, for each task $T_{i,j}$, we have a choice of one software implementation and $m_{i,j}$ implementations accelerated with custom instructions. In addition, $a^0_{i,j} < \ldots a^k_{i,j} < \ldots < a^{m_{i,j}}_{i,j}$ and $t^0_{i,j} > \ldots t^k_{i,j} > \ldots > t^{m_{i,j}}_{i,j}$. The CIS version of a task must fit into the available area, i.e., $a^k_{i,j} \leq AREA$.

Example. Figure 2(a) shows an example of two threads with CIS versions. We assume $AREA = 10$. The first thread has 4 tasks while the second thread has 5 tasks. Each task has multiple CIS version. For example, task $T_{1,1}$ has 3 CIS versions with 0 (software), 1 and 4 area units. The corresponding execution times are 300, 50, and 30 time units.

Our objective is to select a CIS version for each task and appropriate reconfiguration points so as to minimize the execution time of the multi-threaded application, i.e., minimize the execution time of the critical thread.

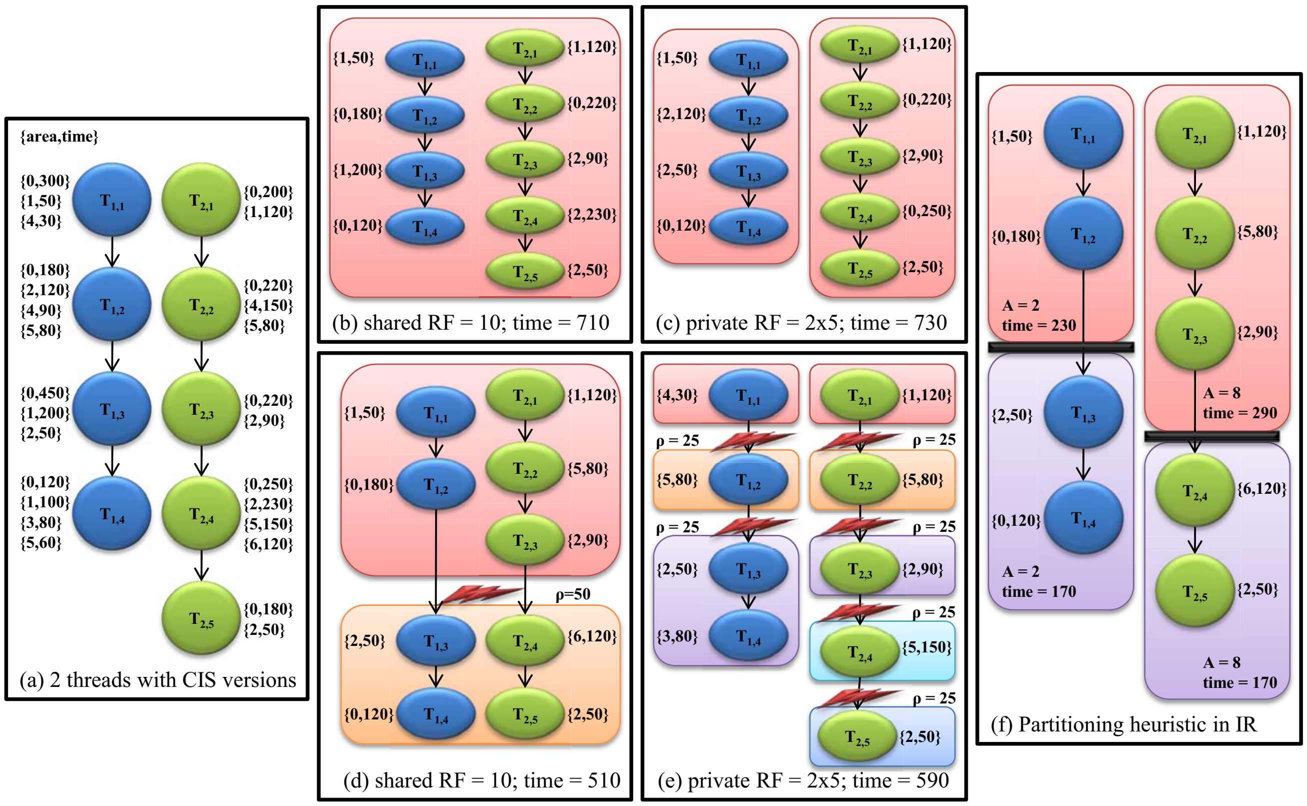
{area,time}
(a) 2 threads with CIS versions
(b) shared RF = 10; time = 710
(c) private RF = 2x5; time = 730
(d) shared RF = 10; time = 510
ρ=50
(e) private RF = 2x5; time = 590
ρ = 25
A = 2
time = 230
A = 8
time = 290
A = 2
time = 170
A = 8
time = 170
(f) Partitioning heuristic in IR

Figure 2: Motivating Example.

Static configuration. Let us first concentrate on a restricted version of the problem where we do not allow any dynamic reconfiguration of the fabric. Let $x_{i,j}^k$ be a binary variable that is set to 1 if the CIS version $c_{i,j}^k$ is chosen corresponding to task $T_{i,j}$ and 0 otherwise. Then our goal is to **minimize** the following objective function:

$$\max_{i=1,\ldots,N} \sum_{j=1}^{n_i} \sum_{k=0}^{m_{i,j}} x_{i,j}^k \times t_{i,j}^k$$

subject to the following constraints

$$\sum_{i=1}^{N} \sum_{j=1}^{n_i} \sum_{k=0}^{m_{i,j}} x_{i,j}^k \times a_{i,j}^k \leq AREA; \quad \forall i,j \quad \sum_{k=0}^{m_{i,j}} x_{i,j}^k = 1$$

This is 0-1 Integer Linear Programming (ILP) problem.

Example. Figure 2(b) and 2(c) show the optimal solutions with shared and private RFs respectively. In case of private RF per core, we assume each core has access to $10/2 = 5$ units of RF. It is easy to prove that shared RF will always lead to better execution time than private RFs. In our example, we get 710 units of execution time with shared RF compared to 730 units of execution time with private RFs.

Dynamic reconfiguration. Allowing dynamic reconfiguration of the fabric adds significant complexity to the problem. Let P be the total number of configurations. In the worst case, each task can have its exclusive configuration, i.e., $P \leq \sum_{i=1}^{N} n_i$. Let $p(T_{i,j})$ be the configuration that $T_{i,j}$ belongs to. Then we have the constraint $p(T_{i,j}) \leq p(T_{i,j+1})$ as partitions contain consecutive tasks. Clearly, each configuration must satisfy area constraint. Therefore

$$\sum_{\forall i,j \ s.t. \ p(T_{i,j})=q} \ \sum_{k=0}^{m_{i,j}} x_{i,j}^k \times a_{i,j}^k \leq AREA \quad q \in \{1 \ldots P\} \quad (1)$$

The execution time of the application is the summation of the execution time of each configuration plus the reconfiguration latency. The execution time in each configuration corresponds to the critical thread in that configuration. So our goal is to **minimize** the following objective function:

$$\rho \times (P-1) + \sum_{q=1}^{P} \max_{i=1,\ldots,N} \left(\sum_{\forall j \ p(T_{i,j})=q} \ \sum_{k=0}^{m_{i,j}} x_{i,j}^k \times t_{i,j}^k \right) \quad (2)$$

Concretely, our goal is to select the CIS version of each task (i.e., assign the $x_{i,j}^k$ binary variables) and assign the configuration for each task $p(T_{i,j})$ such that the total execution time specified by Equation 2 is minimized.

Example. Figure 2(d) and 2(e) show the optimal solutions for shared and private RFs with dynamic reconfiguration. Reconfiguration latency (ρ) is 50 and 25 corresponding to shared and private RFs, respectively. For shared RF, the application has been partitioned into 2 configurations with execution times 290 unit and 170 unit, respectively. Hence the total execution time is ($290+170+50 = 510$) unit. Shared RF allows flexibility in terms of allocating area to each thread. However, the reconfiguration for all the threads have to be synchronized as discussed in Section 2. Thus load-imbalance among the threads can have a negative impact. If the threads have private RFs as in 2(e), each thread can reconfigure its own fabric independently and asynchronously. In our example, T_1 reconfigures 2 times while T_2 reconfigures 4 times. Still the optimal solution with private RFs requires 590 time units compared to 510 unit for shared RF. This is because T_2 has inherently more requirement of custom instructions that can be satisfied with shared RF. Therefore, the design space exploration algorithm needs to carefully take into account the tradeoff between imbalance in load and area requirement among the threads.

The presence of both the partition variables and the CIS version selection variables in the objective function introduces non-linearity making ILP solution infeasible. A much simpler version of the partitioning problem where all the threads have identical number of tasks and the same partitioning is applied to all the threads (there is no reconfiguration delay and CIS versions) is known as the multi-stage linear array assignment problem (MLAA) [6]. The MLAA problem has been shown to be NP-complete. We now present an optimal solution to our problem followed by an efficient iterative refinement algorithm that achieves close to the optimal solution.

4. OPTIMAL SOLUTION

The optimal solution is constructed in a bottom-up fashion by first computing the solutions per thread, then combining them for multi-threading without reconfiguration before finally proceeding to incorporate multiple configurations.

Algorithm 1: Compute $time_{i,s,e}(A)$ for all i, s, e, A

```
for $i \leftarrow 1$ to $N$ do
    for $s \leftarrow 0$ to $n_i$ do
        for $e \leftarrow s+1$ to $n_i$ do
            for $A \leftarrow 0$ to $AREA$ do
                for $k \leftarrow 0$ to $m_{i,j}$ do
                    if $(a^k_{i,e} \leq A)$ then
                        $time_{i,s,e}(A) = min(time_{i,s,e}(A),$
                        $time_{i,s,e-1}(A - a^k_{i,e}) + t^k_{i,e})$
```

Single thread. The term $\sum_{\forall j \ p(T_{i,j})=q} \sum_{k=0}^{m_{i,j}} x^k_{i,j} \times t^k_{i,j}$ in Equation 2 defines the execution time of thread T_i in configuration q. Only a consecutive subsequence of tasks from T_i can be mapped to a configuration. Let $T_{i,s+1}$ and $T_{i,e}$ ($s \leq$e) be the start and end task of the subsequence of tasks from T_i mapped to a particular configuration. Then the execution time of the subsequence can be defined as

$$time_{i,s,e} = \sum_{j=s+1}^{e} \sum_{k=0}^{m_{i,j}} x^k_{i,j} \times t^k_{i,j}$$

Note that according to our definition $time_{i,0,n_i}$ corresponds to the execution time of the entire thread from task $T_{i,1}$ to task T_{i,n_i}. *Moreover, we assume that* $time_{i,s,s} = 0$ *corresponds to the execution time of an empty sequence of tasks.*

We first pre-compute the minimum value of $time_{i,s,e}$ for all possible values of i, s, e under different area constraints. We design a dynamic programming algorithm to compute these values. The recursive equation is

$$time_{i,s,j}(A) = \min_{\substack{k=0,\ldots,m_{i,j} \\ a^k_{i,j} \leq A}} (time_{i,s,j-1}(A - a^k_{i,j}) + t^k_{i,j})$$

where $time_{i,s,j}(A)$ (with $s < j$) is the minimum execution time of the subsequence $T_{i,s+1} \ldots T_{i,j}$ under area constraint A. Basically, we start with the task $T_{i,s+1}$ and add one task at a time till we reach the task $T_{i,e}$. For the task $T_{i,j}$, we go through all its CIS versions that can fit in the area A. For each such CIS version $c^k_{i,j}$, we allocate its area $a^k_{i,j}$ and the remaining area $A - a^k_{i,j}$ is given to the tasks $T_{i,s+1} \ldots T_{i,j-1}$. The execution time under this allocation is the execution time of the task $T_{i,j}$ with CIS version $c^k_{i,j}$ and the minimum execution time of the previous tasks under the remaining area constraint $time_{i,s,j-1}(A - a^k_{i,j})$. Then we choose the CIS version that produces minimum execution time under this scenario. In other words, we set $x^k_{i,j} = 1$ for that CIS version and 0 for all the other CIS versions. Algorithm 1 illustrates this computation. The complexity of the algorithm is $O(N \times n^2 \times m \times AREA)$ where n is the average number of tasks per thread and m is the average number of CIS versions per task.

Algorithm 2: Compute $time_{\langle s1,e1\rangle \ldots \langle sN,eN\rangle}$

```
for $i \leftarrow 1$ to $N$ do  $A_i = 0$;
for $A \leftarrow 0$ to $AREA$ do
    $critical = 0$; $maxTime = 0$;
    for $i \leftarrow 1$ to $N$ do
        if $time_{i,si,ei}(A_i) >= maxTime$ then
            $maxTime = time_{i,si,ei}(A_i)$; $critical = i$;
    $A_{critical} = A_{critical} + 1$;
return $maxTime$;
```

Multi-threading with Static Configuration. Let us suppose subsequences $[T_{1,s1+1} \ldots T_{1,e1}] \ldots [T_{N,sN+1} \ldots T_{N,eN}]$ have been mapped to a particular configuration. The execution time of this configuration will be determined by the subsequence with maximum execution time. We also need to satisfy the area constraint of the configuration (see Equation 1). We define $time_{\langle s_1,e_1\rangle \ldots \langle sN,eN\rangle}$ as the execution time of the subsequences $[T_{1,s_1+1} \ldots T_{1,e_1}] \ldots [T_{N,sN+1} \ldots T_{N,eN}]$ mapped to a configuration. We propose Algorithm 2 to efficiently compute the minimum value of $time_{\langle s1,e1\rangle \ldots \langle sN,eN\rangle}$.

Our goal is to partition $AREA$ among all the threads so as to minimize the execution time of the critical thread. Initially, we set the area assigned to each thread (A_i) to 0. In each step, we allocate unit area to the critical thread so as to reduce its execution time. The correctness of the algorithm can be easily proved through induction on area A, as the execution time can be potentially decreased only by assigning the area increment to the critical thread.

Note that $time_{\langle 0,n_1\rangle \ldots \langle 0,n_N\rangle}$ corresponds to the minimum execution time of the entire application with single configuration. That is, Algorithm 2 can generate the optimal solution for N threads with shared RF without reconfiguration. The complexity of this algorithm is $O(N \times AREA)$.

Multi-threading with Dynamic Reconfiguration. We now proceed to introduce reconfiguration. Let us define $time_{\langle s_1,e_1\rangle \ldots \langle s_N,e_N\rangle}|P$ as the minimum execution time of the task subsequences $[T_{1,s1+1} \ldots T_{1,e1}] \ldots [T_{N,sN+1} \ldots T_{N,eN}]$ with P configurations including reconfiguration overhead of $\rho \times (P-1)$. For one configuration, $time_{\langle s_1,e_1\rangle \ldots \langle s_N,e_N\rangle}|1 = time_{\langle s_1,e_1\rangle \ldots \langle s_N,e_N\rangle}$. We define a recursive equation to compute the execution time for P configurations given the execution times for $P-1$ configurations as follows.

$$\begin{aligned} time_{\langle s_1,e_1\rangle \ldots \langle s_N,e_N\rangle}|P &= \min_{\forall i \ s_i \leq v_i \leq e_i} (time_{\langle v_1,e_1\rangle \ldots \langle v_N,e_N\rangle} \\ &\quad + \rho + time_{\langle s_1,v_1\rangle \ldots \langle s_N,v_N\rangle}|(P-1)) \end{aligned}$$

The equation states that we need to explore all possible combination of starting points in each thread for the P^{th} configuration. This is achieved by setting v_i (starting points of P^{th}

Algorithm 3: Optimal Algorithm

```
$P = 1$;
repeat
    $P = P + 1$;
    $improve = false$;
    for all combinations of $e_i$ $(0 \le e_i \le n_i)$ do
        $min = time_{\langle 0,e_1\rangle \ldots \langle 0,e_N\rangle}|(P-1)$;
        for all combinations of $v_i$ $(0 \le v_i \le e_i)$ do
            $temp = time_{\langle 0,v_1\rangle \ldots \langle 0,v_N\rangle}|(P-1) + \rho + time_{\langle v_1,e_1\rangle \ldots \langle v_N,e_N\rangle}$;
            if $temp < min$ then
                $min = temp$;
                $improve = true$;
        $time_{\langle 0,e_1\rangle \ldots \langle 0,e_N\rangle}|P = min$;
until $!improve$;
$opt = time_{\langle 0,n_1\rangle \ldots \langle 0,n_N\rangle}|P$;
return opt;
```

configuration) between s_i and e_i for each thread T_i. Then $time_{\langle v_1,e_1\rangle \ldots \langle v_N,e_N\rangle}$ denotes the execution time of the P^{th} configuration. The remaining tasks are assigned to the $P-1$ configurations and $time_{\langle s_1,v_1\rangle \ldots \langle s_N,v_N\rangle}|(P-1)$ denotes the corresponding execution time. We add the reconfiguration overhead. The combination of starting points that provides the minimum execution time is the optimal solution.

Algorithm 3 describes the dynamic programming algorithm to find the optimal solution. We start with $P = 1$ configuration and increment the number of configurations by one in each step. We compute the execution time for all possible partitions and then select the one with the minimum execution time. If the execution time improves with the additional configuration, we continue. Otherwise, the algorithm terminates. Clearly, the algorithm has exponential complexity of $O(n^N)$ where n is the number of tasks per thread. However, this algorithm produces the optimal solution and provides a solid reference point.

5. ITERATIVE REFINEMENT

Algorithm 4: Iterative Refinement (IR) Algorithm

```
add static configuration to $SetP$;
$min = time_{\langle 0,n_1\rangle, \ldots \langle 0,n_N\rangle}$;
while $SetP \neq \phi$ do
    choose config $p$ from $SetP$ with max execution time;
    for $i \leftarrow 1$ to $N$ do
        $s_i = Start[p][i]; e_i = End[p][i]; A = Area[p][i]$;
        find $v_i$ with min $|time_{i,s_i,v_i}(A) - time_{i,v_i,e_i}(A)|$;
    $temp = time_{\langle s_1,v_1\rangle \ldots \langle s_N,v_N\rangle} + time_{\langle v_1,e_1\rangle \ldots \langle v_N,e_N\rangle} + \rho$;
    if $temp < Time[p]$ then
        $min = min - (Time[p] - temp)$;
        replace $p$ with partitions of $p$ in $SetP$;
        update $Start$, $End$, $Area$, $Time$;
    else
        remove $p$ from $SetP$;
return $min$;
```

Now we present an iterative refinement technique (see Algorithm 4) that avoids the exponential complexity of the optimal algorithm while achieving close to optimal solution. The basic idea is to start with the static configuration and partition one of the configurations in each step. Suppose we have P configurations (represented by $SetP$) after $P-1$ partitioning steps. Corresponding to each configuration, we maintain the start and end tasks of each thread ($Start$, End), the area required by each thread ($Area$), and the execution time ($Time$). We then choose the configuration p with the maximum execution time and attempt to partition it. The heuristic partitions each thread independently as follows. If in configuration p, thread T_i was allocated area $Area[p][i]$, then we allocate the same area to each of its partition. We then select the point v_i so as to maximize the balance between the two partitions of T_i. Once the partitioning points for all the threads have been selected, we compute the actual execution time per partition by invoking Algorithm 2 and add the reconfiguration overhead. If the execution time of p reduces with partitioning, then we add the new configurations to $SetP$. Otherwise, we remove p from further consideration. The algorithm terminates when we cannot optimize any configuration through partitioning. The complexity per iteration is $O(N \times n + N \times AREA)$. As the number of reconfigurations is typically quite small, the algorithm terminates quickly.

Example. We illustrate the algorithm with the example in Figure 2(a). We start with the static configuration, i.e., the solution in Figure 2(b) with execution time 710. Here T_1 occupies 2 units of area whereas T_2 occupies 7 units of area. We try to partition each thread independently as shown in Figure 2(f). Each partition of T_1 is assigned 2 units of area. With this constraint, the best partitioning point is after task $T_{1,2}$. As T_2 is the critical thread, 1 unit of unassigned area is added to its allocated 7 units of area. With area 8, the best partitioning point of T_2 is after task $T_{2,3}$. Now we determine the execution time of the configuration $\{\langle T_{1,1}T_{1,2}\rangle, \langle T_{2,1}T_{2,2}T_{2,3}\rangle\}$, which is 290. The execution time of the other configuration $\{\langle T_{1,3}T_{1,4}\rangle, \langle T_{2,4}T_{2,5}\rangle\}$ is 170. Hence the execution time of 2-configuration solution is (290+170+50=510), which is better than 1-configuration solution. Next we try to partition each of the configurations. But further partitioning does not improve execution time. So the algorithm returns the 2-configuration solution, which is the optimal solution as shown in Figure 2(d).

6. EXPERIMENT EVALUATION

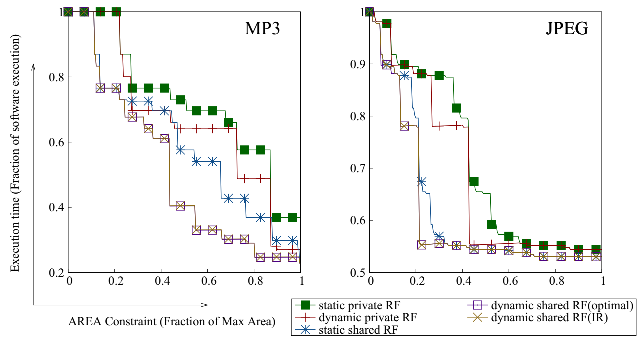

Figure 3: Execution time versus area for MP3 and JPEG with different architectures on 2 cores.

We perform our experimental evaluation on the Stretch platform [4]. We first evaluate our technique with JPEG encoder and MP3 encoder applications. We identify five hot kernels for each of these application through profiling. We partition each application into 2 pipeline stages. Each

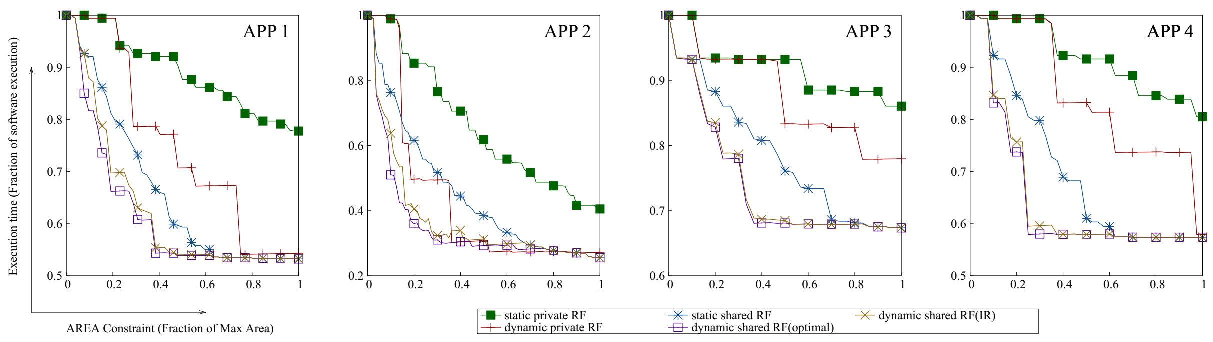

Figure 4: Execution time versus area for synthesized applications with different architectures on 4 cores.

pipeline stage is mapped to one processor core. Our goal is to minimize the execution time of the critical pipeline stage.

To evaluate the scalability of our technique, we also extract a large number of kernels (e.g., ycc2rgb, dequantize, idct, huffmenen, rgb2ycc, crc32, adpcmdec, autocor, dctquan, cjpeg, djpeg, rgb2cmyk, des, fir etc.) from embedded benchmark suites. We combine multiple kernels from the same application domain to create four applications each with 4 threads. The average number of tasks per thread ranges from 2–7. For these synthesized applications, our goal is to minimize the execution time of the critical thread.

For each kernel, we manually generate custom instructions for Stretch platform by using Stetch C language. We achieve different CIS versions by changing the unroll factor of the compute intensive loops or the number of custom instructions. The higher unroll factor results in larger hardware area requirement and better performance gain. The profiler in Stretch provides us the performance and hardware area of the CIS versions of each task.

For each application, we compare five different solutions: (a) static configuration with private RFs (Algorithm 1), (b) static configuration with shared RFs (Algorithm 2), (c) dynamic configuration with private RFs (Algorithm 3 with $N = 1$), (d) optimal solution (optimal) for dynamic configuration with shared RF (Algorithm 3), and (e) iterative refinement (IR) solution for dynamic configuration with shared RF (Algorithm 4). If N cores share $AREA$, then we assume each core has $AREA/N$ amount of private RF.

The configuration time of the whole RF of Stretch is 100μs. For each application, we vary the RF size between 0 to 1.0 (in steps of 0.01) of the area required to implement the best CIS versions of the constituent kernels for solution (b) and set the reconfiguration delay proportionately. When maximum area is available, an application achieves the speedup limit without reconfigurations. The execution time of different solutions are normalized w.r.t. the execution time of purely software implementation on multi-core.

Figure 3 shows the results for MP3 and JPEG applications. As expected, dynamic techniques perform better than static techniques for both shared RF and private RF. However, what is interesting is that static shared RF consistently outperforms private RF with dynamic reconfiguration. That is, even without reconfiguration, sharing the reconfigurable logic already provides substantial improvement. As an example, for MP3 application at 60% of the maximum area requirement, shared dynamic RF and shared static RF reduce execution time to 33% and 54% of software execution time. The corresponding numbers are only 64% and 70% for private dynamic RF and private static RF. Sharing RF with reconfiguration is the clear winner among all four choices. The solutions provided by the iterative refinement algorithm are either identical or closely match the optimal solutions.

Figure 4 shows the results for synthesized applications. The results are similar. Sharing brings about significant improvement to performance. Moreover, iterative refinement algorithm is quite effective for all the points.

Finally, we compare the running time of IR versus optimal algorithm on Intel Xeon 2.53GHz processor with 16GB memory. We synthesize applications with varying number of tasks per thread. The optimal algorithm is not scalable. It takes 220ms (3 tasks/thread) to 7 minutes (10 tasks/thread). With 20 tasks per thread, optimal algorithm fails to return any solution even after running for 10 hours. IR, on the other hand, returns solutions within 1 ms (3 tasks/thread) to 240 ms (20 tasks/thread).

7. CONCLUSION

We propose a multi-core architecture with shared reconfigurable logic to implement application-specific instructions. We develop an efficient algorithm that can minimize the execution time for multi-threaded applications running by selecting appropriate custom instructions and reconfiguration points. By comparing the system with private reconfigurable logic per core, we conclude that sharing reconfigurable logic brings substantial acceleration for applications.

Acknowledgements. This work was partially supported by MOE Singapore research grant MOE2009-T2-1-033.

8. REFERENCES

[1] Z. Chen, R. N. Pittman, and A. Forin. Combining multicore and reconfigurable instruction set extensions. In *FPGA*, 2010.
[2] P. Garcia and K. Compton. Kernel sharing on reconfigurable multiprocessor systems. In *FPT*, 2008.
[3] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2), 2000.
[4] R. E. Gonzalez. A software-configurable processor architecture. *IEEE Micro*, 26(5), 2006.
[5] H. P. Huynh and T. Mitra. Runtime reconfiguration of custom instructions for real-time embedded systems. In *DATE*, 2009.
[6] R. K. Kincaid, D. M. Nicol, and D. R. Shier. A multistage linear array assignment problem. *Operations Research Journal*, 38, 1990.
[7] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *MICRO*, 2004.
[8] S.L. Shee and S. Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *DAC*, 2007.
[9] F. Sun, S. Ravi, A. Raghunathan, and N.K. Jha. Application-specific heterogeneous multiprocessor synthesis using extensible processors. *IEEE TCAD*, 25(9), 2006.
[10] M.A. Watkins and D.H. Albonesi. ReMAP: A reconfigurable heterogeneous multicore architecture. In *MICRO*, 2010.