An ILP Formulation for Task Mapping and Scheduling on Multi-core Architectures

Ying Yi, Wei Han, Xin Zhao, Ahmet T. Erdogan and Tughrul Arslan
University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, UK

***Abstract*-Multi-core architectures are increasingly being adopted in the design of emerging complex embedded systems. Key issues of designing such systems are on-chip interconnects, memory architecture, and task mapping and scheduling. This paper presents an integer linear programming formulation for the task mapping and scheduling problem. The technique incorporates profiling-driven loop level task partitioning, task transformations, functional pipelining, and memory architecture aware data mapping to reduce system execution time. Experiments are conducted to evaluate the technique by implementing a series of DSP applications on several multi-core architectures based on dynamically reconfigurable processor cores. The results demonstrate that the proposed technique is able to generate high-quality mappings of realistic applications on the target multi-core architecture, achieving up to 1.3x parallel efficiency by employing only two dynamically reconfigurable processor cores.**

I. INTRODUCTION

An important trend in embedded systems is the use of multi-core architectures to meet application's functional and performance requirements. Multi-core designs offer high performance and flexibility, at the same time promise low-cost and power-efficient implementations. However, the semiconductor industry is still facing several other technological challenges with multi-core systems. Important issues in multi-core designs are the communication infrastructure, memory architecture, and task mapping and scheduling. In multi-core architectures, the performance of the entire system is affected by the execution order of tasks and communications. It is well known that task mapping and task scheduling are highly inter-dependent. Therefore the two issues need to be handled together in order to obtain efficient mapping and scheduling.

Dynamic reconfigurable (DR) processor combines the flexibility of FPGAs with the programmability found in general purpose processors (CPUs/DSPs) in a unified and easy programming environment. It is a strong candidate for multi-core systems. In our proposed embedded multi-core platform which has several DR processors [1], the shared memory heavily affects the execution time and power consumption. The time of data transmission between different processors must be considered during scheduling such that the design result can conform to the real situation. In addition, in order to meet the system throughput constraints, the design is pipelined to construct more efficient architectures. Pipelining divides the design into concurrently executing stages, thus increasing the throughput.

In multi-core architectures all parallel tasks in an application have the potential to be executed simultaneously. However the number of such tasks may exceed the number of available processors. Therefore task mapping is required to assign the parallel tasks to the available processors. In the past, task merging and task replication have been proposed with the goal of re-allocating tasks when performance bottlenecks are met. Since task merging requires more local memory and task replication needs more processors to implement the same task [2], a multi-core architecture which does not feature sufficient memory and processors will severely limit the available mapping options using the existing methodology.

Application development on multi-core architectures requires the designer, or automated tool, to divide tasks between available processors and to determine data mappings for the required memory elements. A SystemC-based simulation framework for mapping an application to a platform and evaluating its performance has been presented in [3]. The authors in [4, 5] have introduced scheduling and mapping parallel applications onto an MPSoC platform. Mapping solutions for bus-based and NoC-based MPSoCs have been described in [6] and [7]. Some automated system-level mapping techniques for application development on network processors have also been proposed [8].

This paper addresses the problem of automated application mapping and scheduling on DR processor based multi-core architectures. An Integer Linear Program (ILP) based approach is proposed for loop level task partitioning, task mapping and pipelined scheduling while taking the communication time into account for embedded applications. The efficacy of the technique is demonstrated by a series of DSP applications.

The paper is organized as follows: Section 2 introduces the target DR processor as well as the target multi-core architecture. Section 3 describes the task mapping methodology. Section 4 gives a more detailed description of the problem addressed in this paper. Section 5 describes the proposed ILP based approach to solve the problem. The experimental results are given in section 6 followed by conclusions in section 7.

II. TARGET MULTI-CORE ARCHITECURE

Some applications demand a closer interconnection between the participating processors to achieve the required performance. Such a communication can be realised using distributed shared register files. The target multi-core platform is designed for DSP applications, which typically have intensive computations and a stream of input data. The architecture described in a previous work [2] consists of a selectable number of DR processors, which communicate with a shared memory through a full crossbar network. This architecture has been extended and modified by incorporating the shared register file into the system memory architecture in order to support the loop level parallelism proposed in this

978-3-9810801-5-5/DATE09 © 2009 EDAA

paper.

The target multi-core architecture is based on a recently introduced DR processor architecture [9]. The DR processor offers comparable computation performance to leading DSP processors with a significant reduction in power consumption [1]. It contains an array of instruction-set functional units connected by a programmable interconnection. The DR processor is realised using an array of Instruction-Cells (ICs) that is reconfigured every processor cycle to map data-paths consisting of both dependent and independent instructions. The salient characteristics of the DR processor are that it is able to be fully customisable at design stage and can be set according to application requirements. All instruction cells in the DR processor can be connected to the entire shared memory or register file through interface cells. This scheme allows all possible connections between the instruction cells and the shared memory elements, but with a reduced multi-core interconnection complexity. The existing DR processor tool-flow gives full support by providing all the required files for the ILP model, which include a machine description file, a static profiling file, and a task graph (control data dependent graph).

III. Proposed Mapping Flow

Many DSP algorithms target streaming based applications and need to operate in real-time: the DSP application must read input data, process it, and write processed results out before the next input data is ready. A key concern in a DSP system is maintaining real-time execution. It is necessary to pipeline the application into concurrently executing stages to meet the throughput constraints of these systems. The implementation of DSP applications on a multi-core architecture mainly involves partitioning, mapping and scheduling the application tasks onto the processors as well as specifying data mapping and data transmission between these processors. The partitioning, mapping and scheduling of tasks are complex optimisation problems, which need to be solved simultaneously to maximise the throughput. In addition, the data communication time between different processors must also be taken into account during task mapping and scheduling to maximise the system throughput.

The proposed mapping flow allows the designer to explore the application implementations on the target architecture platform as shown in Fig. 1. This paper mainly focuses on the automatic mapping. An ILP-based approach is proposed for loop level task partitioning, task mapping, and pipelined scheduling, while taking the data communication time into account for embedded applications targeted on the multi-core platform.

During the process of task partitioning and task mapping, the estimation of execution-time for the different tasks as well as for transferring the data between processors is required. It is also necessary to schedule the execution order of these pipelined tasks to improve the system performance. The execution time of a task can be obtained from profiling file generated by the single DR simulator. The mapping flow starts from the description of an application in standard sequential C code which is then optimised and profiled for a single DR processor implementation.

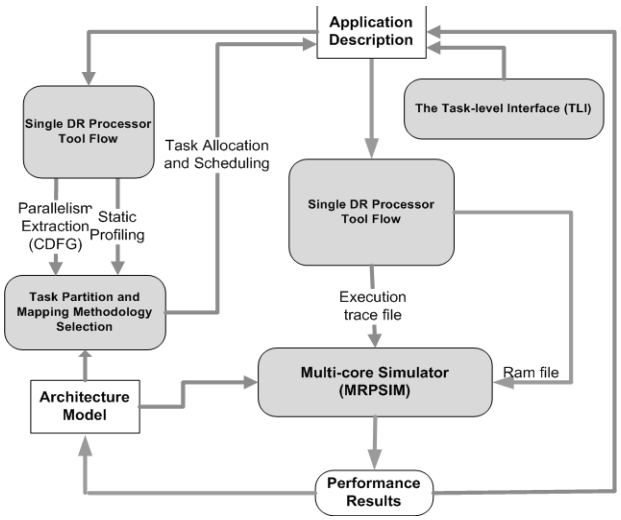

Fig.1: Mapping methodology

Application developers can use the generated task mapping and scheduling information and a task-level interface (TLI) to build multi-core application code. A TLI interface is an application programming interface. It can be used for developing parallel application program on multi-core architectures. The TLI interface provides services for inter-task communication and task allocation. It must allow parallelism and communication to be made explicit to enable mapping to multi-core architectures. For example, if a task uses an abstract interface for synchronization with other tasks, it hides the detailed implementation of the synchronization. The multi-core application code is compiled and simulated with the single DR processor. The single DR processor simulator generates an execution trace file, which is used as an input to the multi-core simulator (MRPSIM) [9]. MRPSIM is a trace-driven simulator which can correctly and efficiently simulate the run-time multi-core environment, allowing the throughput of the modelled system to be measured.

The proposed mapping approach also takes into account the task graph (control data flow graph), the multi-core architecture model, and static profiling file. The static profiling information contains the timing characteristics for each task and the access frequency for the various data items. The multi-core architecture model, also called the machine description file, consists of the set of processors and the set of memory. These are used for mapping tasks to available processors as well as mapping various data items to memory architecture.

Our solution consists of dividing the problem into two stages and solving each consecutively. The first stage assigns and schedules tasks to processors, assuming an idealistic memory mapping, where all data items are mapped to the fastest possible level of memory, ignoring memory capacities. The ILP formulation of this stage includes task merging, task replication, and loop level splitting and fusion. The task merging combines several tasks into a single task that performs tasks in an efficient order. This technique reduces the number of required processors, but needs more local instruction and data memories. Task replication assigns the same task to

several processors such that all instances of the task are executed in parallel. Therefore, task replication needs more processors to implement an application and also more global memory to save the shared data. A new mapping approach is needed when the workload among the processors is unbalanced and task replication cannot be used due to the limited number of available processors. The new mapping approach divides the tasks at basic block level instead of at the function level in order to explore the loop level parallelism.

In tasks and communications scheduling process, in order to consider data dependency between tasks and resolve resource contention, we model the scheduling problem with data dependency constraints between tasks, constraints that represent resource contention. The task graph generated by the DR compiler provides basic block and function level control and data dependency. A task is defined as a small procedure, function or just a basic block. The *start* task has no incoming arcs, and *end (leaf)* task has no outgoing arcs.

The ILP model with and without functional pipelining is proposed in [10] but can only handle this in a restricted way. The restriction is that the first computation of a task in the *(i+1)-th* iteration is only possible if all leaf tasks are finished in the *i-th* iteration. This limitation will generate inefficient solution for some task graphs, which is illustrated in Fig.2. Fig. 2(a) gives a task graph of a small example. The time interval between two successive iterations of the algorithm is called latency (LT). The overall computation time (OCT) of n frames without functional pipelining is equal to $n \cdot OET$, where OET is the overall execution time for one frame. The method given in [10] provides a longer latency (LT=OET) which is shown in Fig. 2(b). Our model removes the above limitation, which results in more efficient mapping and scheduling shown in Fig. 2(c). This approach will be illustrated in detail in the following two sections.

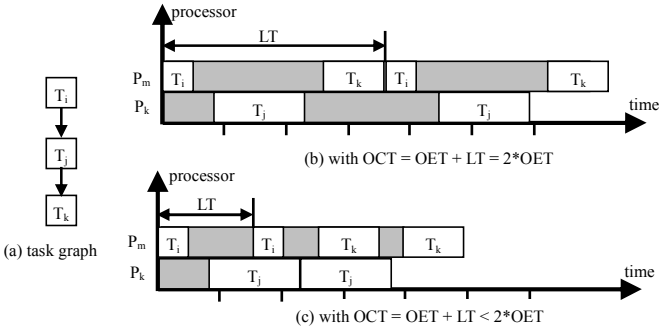

Fig.2. Advantage of functional pipelining

The second stage performs a mapping of data items to memory architecture and explores the memory architecture for minimizing memory access latencies. Each DR processor can access three types of memory: (a) the shared multi-bank register file, (b) local memory, and (c) shared memory. The local memories are private to a processor and cannot be accessed by other processors. Thus, shared data items, accessed by tasks assigned to different processors, cannot be mapped to these memories. Instead, they are mapped either to the shared multi-bank register file or to shared memory, which can be accessed by the different processors. Here, we adopted a similar ILP formulation given in [8] for this stage.

IV. Mapping Heuristics and Problem Definition

In this section, we define the problem of task mapping and scheduling for DR processor based multi-core architectures. Given a task graph, a multi-core target architecture with its parameters, and a mapping of tasks and data on the target architecture including processors and memory, the problem is to find a mapping and scheduling of task executions and communication transactions which yields minimum execution time of the task graph on the target architecture. To solve the problem, target architecture, task and applications definitions are presented. Then, an ILP formulation or a heuristic algorithm is introduced to map and schedule tasks and communications.

A. Architecture Definition

A target multi-core architecture is specified by the set of processors *P* and the set of memory elements *M*. Each processor *p* is a 2-tuple *p=(pid, pcm)* where *pid* is the processor identifier and *pcm* is the instruction memory of the processor. Each memory element *m* is given by a 4-tuple *m=(mid, mc, mt, nm)*, where *mid* is the memory element identifier, *mc* is the capacity of the memory, *mt* is the time required to access the memory, and *nm* is the type of the memory (shared memory, local memory, or shared register file). The target architecture model based on the above specification is extracted from the multi-core architecture model file.

B. Application Definition

An application is represented by the hierarchical task graph, which is an acyclic directed graph *G=<V, E>*, where the vertex set *V* is a *set of tasks* and the edge set *E* is communication edges. Each small procedure or a basic block is defined as a task which is a 5-tuple *t=(tid, pt, tc, td, nd)* where *tid* is the task identifier, *pt is* the task execution time excluding memory conflicting accesses delay, *tc* is the total instruction memory required by the task, *td* is the amount of memory required to store the local data of the task, and *nd* is the number of times the local data item is accessed. Each communication edge is described with the form *(mid, sid, sd, nsd),* where *mid* is the master task identifier, *sid* is the slave task identifier, *sd* is the amount of memory required to store the shared data between two tasks, and *nsd* is the number of times the shared data item is accessed. The application task model is profiled to obtain timing characteristics for each task and the access frequency for the various data items.

The objective is to map and obtain a static mapping and pipelined scheduling of the task graph on the target multi-core architecture such that the throughput is maximized while satisfying performance constraints. The result of the mapping procedure is the decision which tasks run on which processors at what time.

V. ILP Formulations

In this section, we present the ILP formulation that gives an optimal solution for the problem described in section 4. Our solution is based on the mapping strategy given in [8] and extended the ILP model with pipelined scheduling. The ILP model supports task merging and task replication. It means that

a task may be performed on several processors in order to exploit more parallelism. We assume that each processor has its own local memory and only one task can be executed at a time by one processor. Each DR processor can execute all tasks. Tasks on different processors can be executed in parallel. The ILP formulation incorporates task merging and replication by first assigning processes into batches, which are then assigned or replicated to processors [8]. Now, a short summary of the abbreviation is given. N represents the number of tasks; M is the number of available DR processors.

T set of tasks $T = \{T_1, T_2, \ldots, T_n\}$
L set of the end (leaf) tasks $L \subset T$
S set of the start tasks $S \subset T$
P set of processors $P = \{P_1, P_2, \ldots, P_m\}$
B set of batches $B = \{B_1, B_2, \ldots, B_l\}$ $\{l = min(m, n)\}$

Constants and variables used in the ILP formulation:

$s_{ij,k}$ *start time of task* T_i *on processor* P_j *in the k - th iteration*

$C_{ii',k}$ *communication time for sending data from* T_i *to* $T_{i'}$ *in the k - th iteration, if there is* $(i, i') \in E$;

$$t_{il} = \begin{cases} 1 & \text{task } T_i \text{ is assigned to batch } l \\ 0 & \text{otherwise} \end{cases}$$

$$b_{lj} = \begin{cases} 1 & \text{batch } B_l \text{ is assigned to processor } j \\ 0 & \text{otherwise} \end{cases}$$

$$r_{lj} = \begin{cases} 1 & \text{batch } B_l \text{ is replicated on } j \text{ processors} \\ 0 & \text{otherwise} \end{cases}$$

$$d_{ii'j} = \begin{cases} 1 & \text{tasks } T_i \text{ and } T_{i'} \text{ are allocated on the same processor } P_j, \text{ and } T_i \text{ starts execution before task } T_{i'} \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ji} = \begin{cases} 1 & \text{the processor } P_j \text{ that extectue task } T_i \text{ providing data for any task } T_{i'} \text{ with } (i,i') \in E; \text{ the two tasks are allocated on the differnt processors} \\ 0 & \text{otherwise} \end{cases}$$

An objective function depending on the system throughput (*TP*) and the latency (*LT*) need to be minimised. *TP* and *LT* are continuous variables in our ILP model. The objective function is given below. The weights k_1 and k_2 of the costs *TP* and *LT* can be tuned by the designer.

The objective function used in the ILP formulation:

$$minimise\ (k_1 \bullet TP + k_2 \bullet LT)$$

Constraints used in the ILP formulation:

- Every task must be assigned to a single batch;

$$\forall i \in T : \sum_{l \in B} t_{il} = 1 \quad (1)$$

- A batch is replicated on *n* processors, and then exactly *n* processors must execute that batch.

$$\forall l \in B : \sum_{\forall n} n \cdot r_{ln} = \sum_{j \in P} b_{lj} \quad (2)$$

- Each processor must be assigned to a single batch.

$$\forall j \in P : \sum_{l \in B} b_{lj} = 1 \quad (3)$$

- A batch must be assigned to one or more processors only if there is at least one processor assigned to the batch. Otherwise, the batch can be ignored.

$$\forall l \in B : \sum_{j \in P} b_{lj} \cdot MAX\_VAL \geq \sum_{i \in T} t_{il} \quad (4)$$

where MAX_VAL is a very large value.

- The instruction size of all the tasks assigned to a batch cannot exceed the size of the available instruction memory of the processor.

$$\forall l \in B, \forall j \in P : \sum_{i \in T} t_{il} \cdot tc(i) \leq pcm(j) \quad (5)$$

- The throughput is equal to the maximum effective time over all batches.

$$\forall l \in B : TP \geq \sum_{\forall n} \frac{r_{ln}}{n} \sum_{i \in T} t_{il} \cdot pt(i) \quad (6)$$

- The finishing time of each leaf task is less than or equal to OET

$$\forall i \in L, \forall j \in P : s_{ij,0} + pt(i) \leq OET + (1 - t_{il} \cdot b_{lj}) \cdot MAX\_VAL \quad (7)$$

- A data dependency constraint exists between the two tasks T_i and $T_{i'}$ if there is an edge between two tasks T_i and $T_{i'}$ in the task graph *G(V, E)*. The execution of task T_i has to be finished before the execution of $T_{i'}$ if they are on the same processor [constraint (8.1)]. When the tasks are allocated to different processors, $T_{i'}$ can start $c_{ii'}$ time units after T_i has finished [constraint (8.2)]. Since the ILP model supports task replication, a task can be allocated to several processors. We need to consider all possible task allocations of replicated tasks on all processors and also need to know which processor that executes task T_i provides the data for task $T_{i'}$. This is done by variable x_{ji} given in the beginning of section.

$$\forall (i,i') \in E, \forall j \in P : \quad s_{ij,k} + pt(i) \leq s_{i'j,k} + (2 - t_{ij} - t_{i'j}) \cdot MAX\_VAL \quad (8.1)$$

$$\forall (i,i') \in E, \forall j, j' \in P, j \neq j' : \quad s_{ij,k} + pt(i) + C_{ii',k} \leq s_{i'j',k} + (2 - t_{i'j'} + t_{ij'} - x_{ji}) \cdot MAX\_VAL \quad (8.2)$$

- Two independent tasks must not be executed on the same processor at the same time. i.e., Task T_i is executed either before task $T_{i'}$ ($d_{ii'j} = 1$) (9.1) or after task $T_{i'}$ ($d_{ii'j} = 0$) (9.1) on processor P_j.

$$\forall (i,i') \notin E, \forall j \in P : \quad s_{ij,k} + pt(i) \leq s_{i'j,k} + (3 - t_{ij} - t_{i'j} - d_{ii'j}) \cdot MAX\_VAL \quad (9.1)$$

$$s_{i'j,k} + pt(i') \leq s_{ij,k} + (2 - t_{ij} - t_{i'j} + d_{ii'j}) \cdot MAX\_VAL \quad (9.2)$$

- The start time of all tasks have to be positive.

$$\forall i \in T, \forall j \in P : s_{ij,k} \geq 0 \quad (10)$$

- If processor P_j executes task T_i ($x_{ji} = 1$), then task T_i is assigned one batch and this batch is allocated to processor P_j.

$$\forall i \in T, \forall j \in P : x_{ji} \leq \sum_{l \in B} t_{il} \cdot b_{lj} \quad (11)$$

- To minimise the OCT it is necessary to begin the *start* task in iteration *(i+1)* as soon as possible. Each task without replication should be allocated to the same processor in each iteration. In addition, it is required that each *start* task of *(i+1)-th* iteration can only start on a

processor after this task's *i-th* iteration is finished on this processor. If a task has been replicated, there is no constraint between the different iterations. The latency time is affected by the first *start* task of the *i-th* iteration and the first task of the *(i+1)-th* iteration.

$$\forall i \in S, \forall j \in P : s_{ij,k} + LT \le s_{ij,k+1} + (1 - t_{il} \cdot b_{lj}) \cdot MAX\_VAL \quad (12)$$

VI. CASE STUDY: LOOP LEVEL PARALLELISM

The following section demonstrates the effectiveness of the proposed mapping methodology using a series of DSP applications. The application set includes (1) a 64-tap Finite Impulse Response (FIR) filter, (2) an Advanced Encryption Standard (AES) application, (3) a Fast Fourier Transform (FFT) application, (4) a smoothing and edging image processing application for a 256*256 grayscale image, and (5) a Freeman demosaicing application for a 1138*850 RGB image.

Some compiler optimization techniques have been adopted in our multi-core mapping models, which includes loop splitting and loop fusion. Loop splitting attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. Loop fusion (loop combining) attempts to reduce loop overhead. When two adjacent loops iterate the same number of times, their bodies can be combined as long as they make no reference to each other's data.

Let us consider the application of the 64-point 6-stages radix-2 FFT to demonstrate loop splitting. Mapping solution is not only dependent on the strategy but also on the architecture design. The multi-core FFT implementation is mainly affected by the number of processors and the shared register file size in the multi-core architecture. To demonstrate the proposed mapping methodology, the FFT application is mapped to several different multi-core architectures including: (a) limited processor cores with limited shared register banks, (b) limited processor cores with sufficient shared register banks, and (c) sufficient processor cores with sufficient shared register banks.

The FFT-I application is mapped onto a multi-core architecture with two processor cores, sufficient local and shared memories and an 32*32 shared register file shown in Table 3. The most time consuming part of the application is the

TABLE I. A CODE EXAMPLE OF LOOP SPLITTING

```
for (stage=0; stage<STAGES; stage++) {
shuffle(in, out, SIZE);
for (i=0; i<SIZE; i=i+2) {
   getW(&w, (i/2), stage);
   fft(w, out[i], out[i+1], &(in[i]), &(in[i+1]));
   }
}
```

```
for (stage=0; stage<STAGES/2; stage++) {
   shuffle(t1, t2, SIZE);
   for (i=0; i<SIZE; i=i+2) {
      getW(&w, (i/2), stage);
      fft(w, t2[i], t2[i+1], &(t1[i]), &(t1[i+1]));
      }
   }  /* assigned to Processor 0*/

for (stage=STAGES/2;stage<STAGES; stage++) {
   shuffle(t1, t3, SIZE);
   for (i=0; i<SIZE; i=i+2) {
      getW(&w, (i/2), stage);
      fft(w, t3[i], t1[i+1], &(t1[i]), &(t1[i+1]));
      }
   }  /* assigned to Processor 1 */
```

TABLE II. A CODE EXAMPLE OF LOOP FSION

Initial Code	Combining Code

/* Smooth – Processor 0 */
for (y=0; y<HEIGHT; y++)
for (x=0; x<WIDTH; x++)
sharedImage[(y*WIDTH)+x] = filter(x, y, smooth, image);
/* Laplacian – Processor 1 */
for (y=0; y<HEIGHT; y++)
for (x=0; x<WIDTH; x++)
result[(y*WIDTH)+x] = filter(x, y, laplacian, sharedImage);
	for (y=0; y<HEIGHT+3; y++)
	{
	/* Smooth – Processor 0 */
	if (y<HEIGHT)
	for (x=0; x<WIDTH; x++)
	sharedImage[(y*WIDTH)+x] = filter(x, y, smooth, image);
	/* Laplacian – Processor 1*/
	if (y>=3)
	for (x=0; x<WIDTH; x++)
	result[((y-3)*WIDTH)+x] = filter(x, (y-3), laplacian, sharedImage);
	}

2-level loop body shown in Table 1. The loop is split into two parts: the first part executes the beginning 3 stages and the second part executes the last 3 stages (Table 1). Since there is a limited shared register file, which is not sufficient to save all the shared data. The FFT will make use of shared memory to send data from one processor to another. In the general case, data cannot simply be written to and read from shared memory in a multi-core architecture. Programmer can use the mutex or semaphore instruction defined in TLI interface to synchronise the data transfer between processors.

If there are sufficient shared registers, the shared data items can be mapped to the shared registers instead of the shared memory. This detailed implementation called FFT-II is given in Table 3. These shared registers are 8 bank 32*32 bit data registers, whose access time (2ns) is much smaller than one of shared memory (5ns). Therefore, frequently read and written shared data should be mapped to the shared register file in order to reduce memory access time and memory access conflicts. A more efficient mapping can be implemented for a multi-core architecture which has three available processor cores (FFT-III) and sufficient shared registers. The number of stages is evenly divided into three processors, and each processor executes 2 stages of FFT.

A similar technique is adopted by the 64-tap FIR filter. The FIR-I filter is split up and implemented on two processors architecture: the first processor executes the first 32 taps and the second processor executed the last 32 taps. The FIR-II application has been implemented on 4 processors with each processor executing 16 taps. A fully parallel implementation of the 64-tap FIR filter requires 64 processors, which is determined by the number of taps.

An image processing application (IMP) includes two stages: image smoothing and edge enhancement. Image smoothing attempts to capture important patterns in the image data while leave out noise. Edge enhancement is a digital image processing filter that improves the apparent sharpness of an image. The initial implementation is given in Table 2. Edge detection waits for entire image smoothing to finish before it begins. The implementation performance is limited by synchronization; however there is no need to wait for entire image. Two stages can synchronize at every line of pixels. Loop combining is adopted in the IMP application. Combining code is given in Table 2, which results in nearly two times better performance of the original code, and nearly 90% processor efficiency compared to the original 50% efficiency.

A series of DSP applications targeted on several multi-core

TABLE III. DATE MAPPING

Apps.	Memory Capacity			Memory Access Count		
	Shared (KB)	Local (KB)	S.Reg. (b)	Shared (KB)	Local (KB)	S. Reg. (b)
FIR(I)	256	64	32*32	228	4,566	361
FIR(II)	256	64	4*32*32	2,062	5,041	1,086
AES	256	32	32*32	8	422	26
FFT(I)	256	32	32*32	388	4,273	19
FFT(II)	256	64	8*32*32	132	4,273	283
FFT(III)	256	64	8*32*32	133	5,289	300
IMP	256	128	4*32*32	212,384	382,964	20
Freeman	256	128	4*32*32	1,5451,764	532	3,414

TABLE IV. IMPLEMENTATION RESULT

Apps.	No.of Proc.	Execution Time (ms)	Speedup	parallel efficiency	Average Idle Ratio
FIR(I)	2	3.209	2.59	1.30	1.2%
FIR(II)	4	1.658	5.01	1.25	2.56%
AES	2	0.232	1.77	0.89	12.3%
FFT(I)	2	2.957	1.85	0.93	15.5%
FFT(II)	2	2.860	1.92	0.96	14.1%
FFT(III)	3	1.989	2.76	0.92	12.5%
IMP	2	254.848	1.76	0.88	18%
Freeman	2	4.004	1.82	0.91	11.2%

architectures are described in Tables 3 and 4. To make fair performance comparisons, all applications are executed with 100 frames. The experimental results are based on the following assumptions: the DR processors operate at 500MHz, the shared memory access delay is 5ns, the local private memory access delay is 4ns, and the shared multi-bank register file access delay is 2ns. The memory sizes together with the number of memory access operations for each application are given in Table 3, where each processor has an equal local memory size. A memory access occurs when information is read from or written to a memory unit. Table 3 also provides the average number of memory access operations per frame. Table 4 shows the total execution time, the speedup, parallel efficiency, and average idle ratio of the different applications. Speedup refers to the amount by which a parallel algorithm speeds-up compared to a corresponding sequential algorithm. The parallel efficiency (PE) metric indicates how efficiently the processors are utilized in solving the problem, and is obtained by dividing the speedup achieved by the number of processor cores used. The idle ratio (IR) metric refers to the ratio between the periods when a DR processor core is idle and the overall simulation time. The IR given in Table 4 provides the average idle rate of each processor. As the results show, the applications with sufficient processors and memory elements achieve both the highest speedup and the highest parallel efficiency, compared to other task mapping solutions. In all applications with loop splitting and loop fusion, all processor cores are much more efficient with very low idle ratios.

The FIR filter with sufficient local memory and shared registers gains a super linear speedup [11] where the speedup is greater than the number of processor cores. The super linear speedup obtained in this paper attributes to the data locality which reduces the accesses to the slower shared data memory and dramatically improves the performance. Up to 2.59x super linear speedup and a parallel efficiency of 1.30 have been achieved with only two DR processor cores with a local memory and employing loop splitting. The simulation results show that the proposed mapping methodology with loop level parallelism provides superior performance. .

VII. CONCLUSIONS

The focus of this paper is on modeling the task mapping and scheduling problem as an ILP which allows the use of standard tools for solving it. The proposed mapping technique utilizes profiling-driven task partitioning and loop level transformations. These are intelligently fused with loop splitting, loop fusion and a memory aware data mapping in order to reduce system execution time. Several applications based on different multi-core architectures have been generated using our mapping and scheduling tool. Simulation results demonstrate the effectiveness of the proposed mapping and scheduling strategies, showing up to 1.3 parallel efficiency for a multi-core architecture with two DR processor cores.

REFERENCES

[1] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The Reconfigurable Instruction Cell Array," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 16, pp. 75-85, 2008.

[2] Wei Han, Ying Yi, M. Muir, N. Ioannis, T. Arslan, and A. T. Erdogan, "Efficient Implementation of WiMAX Physical Layer on Multi-core Architecture with Dynamically Reconfigurable Processors", Scalable Computing: Practice and Experience Scientific international journal for parallel and distributed computing, Vol. 9, ISSN 1097-2803, 2008.

[3] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout, "A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms," in Proceedings of the conference on Design, Automation and Test in Europe (DATE), pp. 876-881, 2005,.

[4] P. G. Paulin, "Automatic mapping of parallel applications onto multiprocessor platforms: a multimedia application," in Digital System Design, Euromicro Symposium, pp. 2-4, 2004.

[5] N. Pazos, A. Maxiaguine, P. lenne, and Y Leblebici, "Parallel modeling paradigm in multimedia applications: Mapping and scheduling onto a multi-processor system-on-chip platform", in Proceedings of the International Global Signal Processing Conference, Santa Clara, California, 2004.

[6] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, M. Milano, "Communication-aware allocation and scheduling framework for stream-oriented multi-processor system-on-chip", in Proceedings of the Conference on Design, Automation and Test in Europe (DATE), pp. 3-8, 2006.

[7] C. Marcon, A. Borin, A. Susin, L. Carro, F. Wagner, "Time and Energy Efficient Mapping of Embedded Applications onto NoCs", Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 33- 38, Vol. 1, 2005.

[8] C. Ostler and K.S. Chatha, "An ILP Formulation for System-Level Application Mapping on Network Processor Architectures", Design, Automation & Test in Europe Conference & Exhibition, (DATE), pp.1-6, 2007.

[9] Wei Han, Ying Yi, M. Muir, N. Ioannis, T. Arslan, and A. T. Erdogan, "MRPSIM: a TLM based Simulation Tool for MPSoCs targeting Dynamically Reconfigurable Processors," 21st Annual IEEE International SOC Conference, pp. 41-44, September, 2008.

[10] A. Bender, "Design of an Optimal Loosely Coupled Heterogeneous Multiprocessor System", Proceedings of European Design and Test Conference (ED&TC 96), pp 275-281, 1996.

[11] David Culler, J.P. Singh and A. Gupta, "Parallel Computer Architecture: A Hardware/Software Approach", Morgan Kaufmann, 2nd edition, 1999.