The Design of Dynamically Reconfigurable Datapath Coprocessors

ZHINING HUANG and SHARAD MALIK
Department of Electrical Engineering, Princeton University
and
NAHRI MOREANO and GUIDO ARAUJO
Institute of Computing—UNICAMP Campinas

Increasing nonrecurring engineering and mask costs are making it harder to turn to hardwired application specific integrated circuit (ASIC) solutions for high-performance applications. The volume required to amortize these high costs has been increasing, making it increasingly expensive to afford ASIC solutions for medium-volume products. This has led to designers seeking programmable solutions of varying sorts using these so-called programmable platforms. These programmable platforms span a large range from bit-level programmable field programmable gate arrays to word-level programmable application-specific, and in some cases even general-purpose processors. The programmability comes with a power and performance overhead. Attempts to reduce this overhead typically involve making some core hardwired ASIC like logic blocks accessible to the programmable elements. This paper presents one such hybrid solution in this space—a relatively simple processor with a dynamically reconfigurable datapath acting as an accelerating coprocessor. This datapath consists of hardwired function units and reconfigurable interconnect. We present a methodology for the design of these solutions and illustrate it with two complete case studies: an MPEG2 coder, and a GSM coder, to show how significant speedups can be obtained using relatively little hardware. This work is part of the MESCAL project, which is geared towards developing design environments for the development of application-specific platforms.

Categories and Subject Descriptors: J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD)

General Terms: Algorithm, Design, Performance

Additional Key Words and Phrases: Loop pipelining, reconfigurable datapath, coarse-grain reconfigurable fabric, interconnection design, datapath synthesis

1. INTRODUCTION

Deep submicron issues of interconnect delay and signal integrity have significantly increased the design costs of application specific integrated circuits

This research was supported by the Gigascale Silicon Research Center (GSRC), sponsored by MARCO and DARPA.
Authors' addresses: Z. Huang, S. Malik, Department of Electrical Engineering, Princeton University, Princeton, NJ 08544; email:{znhuang,sharad}@ee.princeton.edu; N. Moreano, G. Araujo, Institute of Computing—UNICAMP Campinas, SP Brazil 13084-971; email:{nahri,guido}@ic.unicamp.br.

(ASICs)—both due to the higher engineering costs resulting from longer design cycles, as well as due to the increasing cost of design tools [Keutzer et al. 2000]. This increase in nonrecurring engineering (NRE) costs, coupled with increasing mask costs, has pushed up the break-even volume point at which the per-device cost for the manufactured product is viable. As a consequence, for many products below this break-even volume mark there is a desperate need to satisfy the application performance (timing and power) requirements using specialized programmable solutions. These programmable solutions are increasingly referred to as programmable platforms, and their design (and use) is an exercise in trying to match application concurrency with architectural concurrency. The goal here is to provide the efficiency of ASICs combined with the flexibility of general purpose processors (GPPs).

There is a vast range of these platforms, from bit-level programmable fine- and coarse-grained field programmable gate arrays (FPGAs) to word-level programmable application-specific processors. The application-specific processors themselves span a fairly large range—from specialized VLIW processors with complete instruction sets tailored to the application domain, to simple RISC processors with a few specialized instructions implemented using hardwired logic blocks [Wang et al. 2001]. These processors bridge the ASIC-GPP gap by adding the efficiency of ASICs to the flexibility of GPPs. A relatively new class of platforms consists of simple processors with programmable coprocessors, which are essentially accelerating datapaths with very little, if any, control. These coprocessors bridge the ASIC-GPP gap by moving in the other direction, by adding the flexibility of GPPs to the efficiency of ASICs.

Early research on programmable coprocessors uses generic reconfigurable logic to accelerate application kernels. These reconfigurable coprocessors can be divided into the fine-grained category, such as GARP [Hauser and Wawrzynek 1997], NAPA [Rupp et al. 1998], Chimaera [Hauck et al. 1997], and PRISC [Razdan and Smith 1994]; and the coarse-grained category, such as Pleiades [Wan et al. 2000], PipeRench [Goldstein et al. 1999], Chameleon [Salefski and Caglar 2001], and RAPID [Ebeling et al. 1996]. Coarse-grained configurable logic has the advantage of providing for faster reconfiguration, fewer configuration bits and faster clock speed in the reconfigurable logic. Usually coarse-grained configurable architectures are more suitable for data-intensive applications in the multimedia and communication domains, while fine-grained architectures are better for bit-level computation.

In this paper, we present the design of a coarse-grained dynamically reconfigurable datapath, working as a coprocessor to accelerate application kernels. The reconfigurable datapath has fixed hardwired logic blocks and programmable interconnections. The number and type of logic blocks in the datapath, as well as the interconnections between them, are selected specific to a given application. The application is specified by providing its key computation kernels. The reconfigurable datapath is constructed by first designing a dedicated datapath for each of these kernels that will result in the maximum throughput, and then combining these datapaths into a single reconfigurable datapath using programming interconnections. The reconfigurable datapath can be configured into any of the kernel-specific datapaths at run-time. We provide a complete

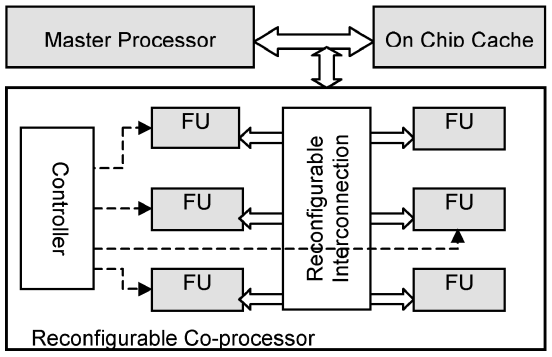

Fig. 1. Architectural model.

methodology for the design of such a datapath coprocessor given an application. The design methodology has been implemented and fully automated. The design flow is illustrated through two nontrivial case studies, an MPEG2 encoder and a GSM encoder. Two reconfigurable datapaths are formed, for the two applications respectively, from their C language descriptions. We show significant speedups through the two reconfigurable datapaths using relatively little hardware.

To further improve the performance of a datapath coprocessor, some carefully designed complex function units can be used in a datapath to replace the function of several simple function units. Usually this results in lower datapath latency. However, introducing complex irregular function units results in less sharing across the datapaths of different kernel loops, thus increasing the hardware cost of the reconfigurable datapath. In this paper, we propose the idea of merging function units on the critical path to improve the datapath performance while avoiding excessive hardware cost. Preliminary experimental results show significant performance improvement through this function unit merging process.

The rest of the paper is organized as follows. Section 2 introduces the architectural model and Section 3 covers the overview of the design flow. In Section 4, we explain how to design a custom loop datapath for a selected kernel loop and in Section 5, we give an algorithm to merge all the custom loop datapaths into a single reconfigurable datapath. We provide experimental results in Section 6, and Section 7 has some conclusions and future research topics.

2. ARCHITECTURAL MODEL

The architectural model we propose in this paper consists of a master processor and a reconfigurable coprocessor as illustrated in Figure 1. The master processor controls the execution of programs as well as the reconfiguration of the datapath coprocessor. When the master processor hit a kernel loop during execution that can benefit from running on the coprocessor, it reconfigures the reconfigurable datapath and switches the execution to the coprocessor.

The coprocessor is built on the same chip as the master processor. We assume the coprocessor can access different levels of on- and off-chip cache and

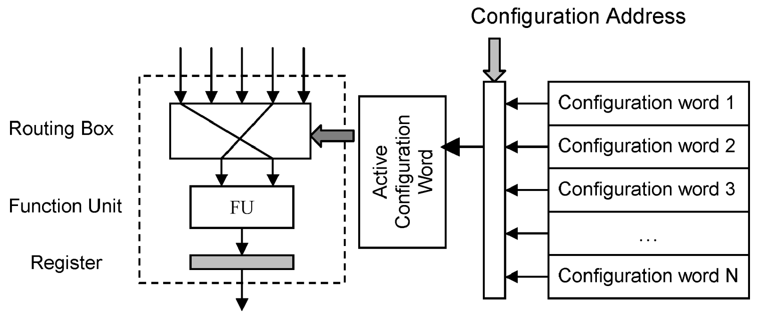

Fig. 2. Routing box for function units.

main memory as the master processor can. The coprocessor mainly consists of a reconfigurable datapath and some control logic. The controller works as a state machine and controls the start and end of the datapath execution as well as the dynamical reconfiguration of the datapath during execution. The reconfigurable datapath has fixed ASIC-like functional blocks and reconfigurable interconnections.

In the reconfigurable datapath, there is a routing box in front of each function unit as shown in Figure 2. The routing box is used to select inputs from different interconnections for the function unit (FU). The configuration bits for a routing box to select the inputs are called configuration word. The configuration words for all routing boxes in the reconfigurable datapath form a reconfiguration context. During execution, the interconnections of the datapath are dynamically reconfigured at each clock cycle by selecting configuration words for the routing boxes according to the configuration address, which is obtained from the datapath controller. The function units in the datapath are hardwired logic blocks. No control bits are necessary. By programming the interconnections, the reconfigurable datapath forms the kernel-specific datapath for a specific kernel loop.

3. METHODOLOGY OVERVIEW

In this paper, we focus on the design methodology of the application-specific reconfigurable datapath. The design methodology can be partitioned into two major stages as shown in Figure 3. The first stage is to design a custom datapath for each selected kernel loop. The second stage is to merge all the datapaths for selected kernel loops into one reconfigurable datapath for an application or application domain.

The first step in the design flow is the identification of the computation intensive loops (kernel loops) in the application. For each kernel loop an initial custom datapath is designed with the maximum possible operation and loop level parallelism under no hardware resource limit. Then, given the hardware resource constraints in the system, function units and registers in the datapath are shared and allocated to make a new datapath. After optimization, a final custom datapath is designed for each selected kernel loop.

Datapaths designed for different loops in an application are then merged into a single reconfigurable datapath, using a maximum clique graph-theoretic

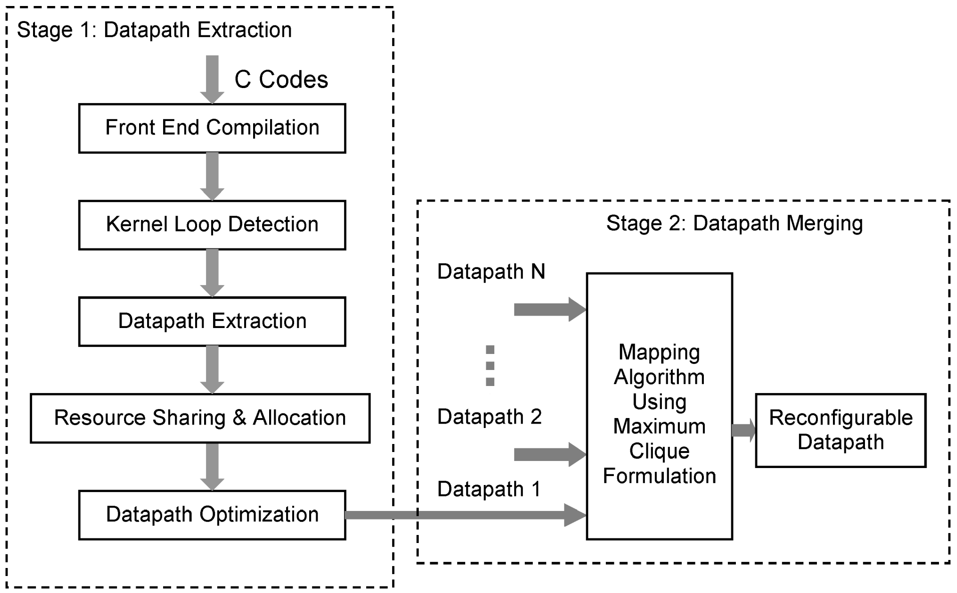

Fig. 3. Design methodology flow.

formulation. The reconfigurable datapath reconfigures itself into different kernel loop datapaths through the programming of interconnections between logic blocks. The interconnection program bits for a loop datapath are referred to as the reconfiguration contexts of the loop. If the size of this program is small (as we will demonstrate) and the number of contexts is small (as we will also show), then the contexts can all be stored locally in the coprocessor and need not be brought in from external memory. This enables rapid switching of contexts—a switch can be accomplished within a single cycle.

4. DATAPATHS FOR KERNEL LOOPS

This section describes the first stage in the design flow—how to design custom loop datapaths for each kernel loop given high-level language descriptions of an application.

4.1 Extracting Kernel Loops

We use the IMPACT compiler [Chang et al. 1991] as the front end to do preprocessing, including performance profiling and loop detection. The application is described in C and this is the input to the front-end compiler. As we only map the innermost loops to custom datapaths in the coprocessor at this stage, the result of extracting kernel loops is a list of the most-executed innermost loops in Lcode (a meta-assembly language), which is the intermediate representation of the IMPACT compiler.

In order to map kernel loops to customized pipelined datapaths, we need to perform data dependence analysis to derive the following: register live-in set of the loop body; register live-out set of the loop body; data dependence between instructions within loop iterations; data dependence between instructions in different loop iterations.

The live-in and live-out sets are used during the switch between the master processor and the coprocessor. When the execution of the application hits the loop, the general processor switches the execution to the coprocessor. During the switch, the values of registers in the live-in set are copied into the coprocessor. The live-out set contains registers whose value has changed during the loop execution and will be used later. After the loop execution, the coprocessor switches execution back to the master processor and writes back the live-out register values.

The data dependence information helps in constructing the datapath for the loop body.

4.2 Direct Mapping of Kernel Loop Datapath

Given the Lcode intermediate representation for a kernel loop, our custom datapath design for that loop proceeds by starting with a direct hardware mapping of the Lcode to hardware blocks. Direct mapping here means that one instruction in software corresponds to one function unit in the hardware. By connecting all the function units in the hardware according to data flow, we form the initial datapath for a loop body.

Each function unit in the hardware executes an instruction in the software. Since the data goes through the datapath, no write back to the register file is needed. This further implies no register bandwidth limitation within loops. The advantage of direct mapping is the simplicity of the methodology, which makes it easy to implement automatically. If there is more than one basic block in the loop body, the two sides of branch instructions result in different datapaths, which are then merged using multiplexes.

4.3 Pipelining the Execution

The techniques used in this section are similar to software pipelining [Lam 1988, Rau 1996]. As the first step in pipelining the execution, we need to assign the function units to different pipe stages in the datapath. Function units in the following discussion refer to all hardware blocks, including memory ports, registers, adders, multipliers, and so on.

The scheduling of the pipeline stages for the datapath maximizes the parallelism of the loop body within the iteration. The scheduling of execution of consecutive loop iterations maximizes the parallelism between loop iterations. If there is no data dependence between loop iterations, at each clock cycle, we can fetch and begin to execute the next loop iteration until we finish all the iterations. If there is data dependence between loop iterations, we look at the pipeline stages of the instructions that cause the dependence to decide whether a delay or a bypass is needed. The delay of fetching consecutive loop iterations is referred to as the iteration interval or initial interval (II).

Assume that data dependence between loop iterations occurs between function unit A and function unit B. The dependence arc is from B to A, which means the output of B is used as input of A in the next loop iteration. If FU A and FU

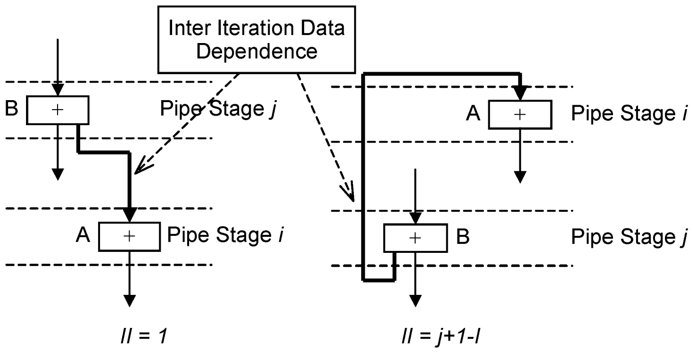

Fig. 4. Inter loop iteration data dependence determines the loop II.

B are scheduled at pipeline stages i and j respectively, the output of B is available at pipe stage $j + \text{Delay}(B)$ and is used at stage i as the input to A in the next iteration. Delay(B) represents the latency of function unit B measured in clock cycles. If $j + \text{Delay}(B) \leq i + 1$, the output of B is always generated before it is used in A because of the pipelined execution. The next iteration is always one step behind this iteration during the pipeline execution. So all we need is a bypass from function unit B to function unit A. If $j + \text{Delay}(B) > i + 1$, the output of B is available only after it is needed in A if new loop iterations are still fetched every clock cycle. Obviously this is not feasible. In this case, an extra delay is needed in starting the new iteration. The new initial interval for fetching new loop iterations is $II = j + \text{Delay}(B) - i$. A feedback interconnection is now needed from FU B to FU A. The two cases are illustrated in Figure 4.

There are two kinds of interconnections in the datapath. The first one is normal connections from data dependence within a loop iteration. The second one is feedback connections from data dependence between loop iterations. The first one determines the total pipeline stage number of the loop datapath. The second one determines the *II* of loop iteration fetch. To synchronize the execution, after scheduling, a certain number of pipeline registers are inserted between function units if necessary.

We assume that function units which take multiple cycles to complete can be internally pipelined. For example, an integer multiplier typically takes three cycles. So each multiplier occupies three pipeline stages in the datapath. Similarly, memory ports also take three pipeline stages.

4.4 Estimation of the Pipeline Execution Time

The pipeline execution of the datapath exploits the maximum parallelism that exists within the loop body and between loop iterations. If there is not much parallelism in a loop, it may not be worth switching the execution to the reconfigurable datapath. Due to the limited on-chip memory size, only a few kernel loops can be mapped and executed on the reconfigurable datapath. The kernel loops are selected based on the estimation of execution time and speedup obtained using their datapaths.

The total execution time of a loop on the datapath is given by

$$T = [S + II \times (N - 1)] + O \quad \text{cycles} \tag{1}$$

S denotes the total number of pipeline stages. II is the iteration interval, which represents the delay between consecutive loop iterations. N is the loop iteration count. O is the overhead of execution switch (read-in and write-back between master processor and reconfigurable coprocessor) in cycles.

4.5 Function Unit Merging on Critical Path

From equation (1), obviously II is a dominant factor for the total loop execution time on the reconfigurable datapath, since usually N is a large number. For the pipelined datapath, the minimum II value is 1. From the analysis in Section 4.3, $II > 1$ denotes that there is some data dependence between loop iterations. Assume the data dependence happens between operation A and operation B, where A is scheduled at pipeline stage i, B is scheduled at pipeline stage j. The II value is $(j - i)+$ Delay(B). Thus, the value $(j - i)$ should be reduced so that II of the datapath is minimized.

In a datapath, each feedback arc represents a data dependence between loop iterations, and has a minimum requirement of II, say II_{arc}. Then II of this datapath can be determined as

$$II = \max\{II_{\text{arc}}\} \quad \text{for any arc in the datapath.} \tag{2}$$

The normal arcs in a datapath represent the data flow within loop iteration. The critical path of a feedback arc r is defined as the set of all direct normal paths from operation A to operation B, where feedback arc r is from operation B to operation A. Normal path means a path only including normal arcs (no feedback arcs). Obviously, the critical path of feedback arcs in a datapath is the key to minimize II, since the total delay of the critical path determines the pipeline stage distance between operation A and operation B. Sometimes a rescheduling of the operations A and operation B can reduce II. One extreme case is shown in Figure 5(b), where there is no critical path between operation A and operation B. Operation A is rescheduled to a later pipeline stage that is the same as operation B. In this way II is reduced to the minimum value, $II = 1$. However, a reschedule may enlarge the total pipeline stage number of the datapath. The direct mapping scheme guarantees the minimum total pipeline stage number. Although the total pipeline stage number is larger after rescheduling, it is easy to prove that the total execution time of this kernel loop on the datapath is reduced when the loop iteration count $N > 2$.

However, in most cases there is a critical path between operation A and operation B. One way to reduce the delay of the critical path is to merge function units on the critical path and make a special complex function unit, which can perform the exact same function. Previous study has shown that such complex function units usually have much less delay compared to the total delay of the several regular function units [Wang et al. 2001].

This leads to the following question: How do we determine the entire function unit set on a critical path? For a feedback arc B to A, there are three possibilities for the critical path as shown in Figure 5. Figure 5(a) shows the most common

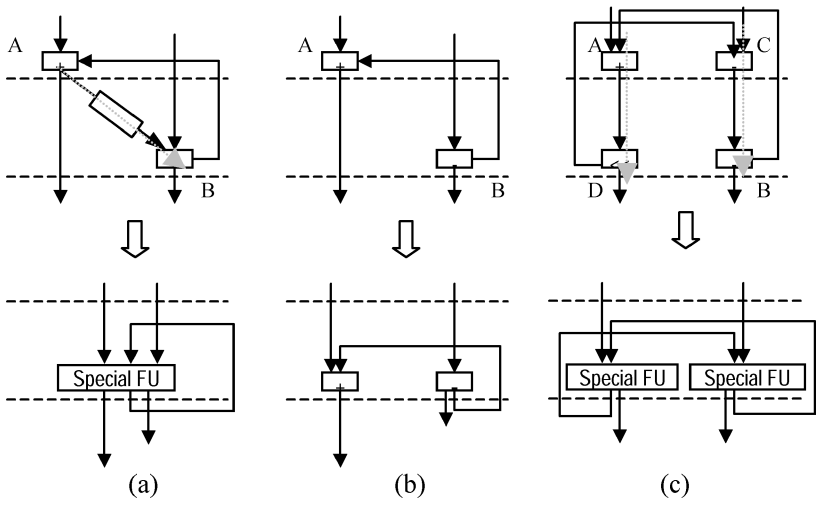

Fig. 5. Finding critical path for feedback arcs.

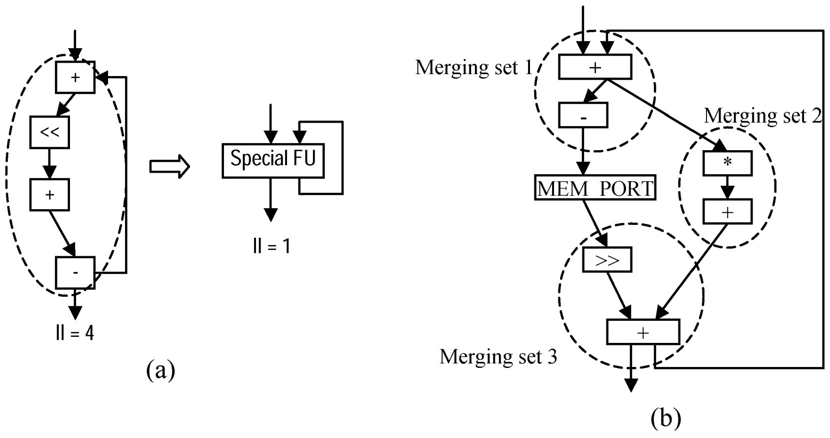

Fig. 6. FU merging on critical path.

case. Figure 5(b) considers the case when no critical path exists. In Figure 5(c), there is no direct normal path from operation A to operation B. However, there is another feedback arc D to C. There are direct normal paths from A to D and from C to B. Both the paths should be considered as the critical path since there is no way to reschedule A, B, C, D to achieve the minimum *II*. They depend on each other and result in a scheduling deadlock.

If there is no memory port, all the function units on a critical path form a merging candidate set. You can either merge all the function units in the candidate set, as shown in Figure 6(a), or merge some subset of function units according to a match in any complex function unit library. If there are memory ports on the critical path, the critical path set is broken into several subsets as required by the memory ports, as shown in Figure 6(b). Each subset becomes a merging candidate set.

Our preliminary solution for the function unit merging problem is to merge all the function units in a merging candidate set. We assume the delay of the special function unit is the maximum delay among normal function units in the merging set. While this holds in some cases, this is not universal and needs to be relaxed. However, it does give us the upper bound on the benefits obtained from function unit merging process for a datapath.

4.6 Hardware Resource Sharing and Allocation

The datapath generated by direct mapping fully exploits the operational level parallelism under no hardware resource constraint. Usually, the number of function units in a datapath is the same as the number of instructions in the kernel loop. Since there is no integrated register file in the datapath, there is no register file bandwidth limit. However, since the datapath coprocessor shares the same memory system as the master processor, the memory bandwidth is limited. In another words, the number of memory ports in a datapath is limited. Currently, we do not limit other hardware resources (adders, multipliers, MUXs, registers) under the assumption of enough silicon area for the system.

If the number of memory ports in a datapath exceeds the bandwidth limit, one memory port needs to be shared by different memory operations at different clock cycles. Loop iterations can no longer be fetched at each clock cycle. This results in an increase in the iteration interval, to say II_{mem}. The new II for the loop datapath is

$$II = \max\{II_{\mathrm{dep}}, II_{\mathrm{mem}}\} \tag{3}$$

$$II_{\mathrm{mem}} = \left\lceil \frac{\textit{Memory Port Number}}{\textit{Memory Port Number Limit}} \right\rceil. \tag{4}$$

In (3), II_{dep} is the II from data dependence analysis in Sections 4.3 and 4.5. If II of the loop datapath is larger than 1, loop iterations are not fetched every clock cycle. Just as memory ports are shared by different memory operations, other function units can be shared as well. The problem we need to solve now is to schedule the datapath to minimize the requirement of hardware resources while keeping the maximum operational level parallelism. There is significant prior research in this area. The scheduling problem for minimizing the hardware resources in datapath synthesis has been formalized into an integer linear programming (ILP) problem [Lee et al. 1989]. We follow this work and combine the scheduling process of simultaneously maximizing operation level parallelism and minimizing hardware resources in the following ILP formulation.

As pointed out in Section 4.5, direct mapping optimizes the total pipeline stage number S. However, loosening the constraint of using this S and rescheduling some operations in the datapath can minimize the II as shown in Figure 5(b) at the cost of increasing S. Also, after introducing the memory bandwidth limitation, II may be changed. The memory ports need to be rescheduled, which may cause the total pipeline stage number S to increase. To minimize II and S, we integrate the optimization into the scheduling problem and use the ILP solver in three steps.

Operations here represent function units in initial datapaths. Since initial datapaths are generated using direct mapping, they can also be viewed as the data flow graph (DFG) of a kernel loop. The variables used in the ILP formulations are listed as the following:

M_k is the number of function units of type k.
C_k is the hardware cost of function unit type k.

$$X_{i,j} = \begin{cases} 1 \text{ if } & O_i \text{ is scheduled into pipe stage } j; \\ 0 \text{ otherwise.} \end{cases}$$

AN operation (function unit) in the datapath is labeled O_i, $1 \leq i \leq n$.
$O_i \rightarrow O_j$ denotes a direct connection from operation O_i to O_j.

The ILP formulations we consider have the following objective functions,

$$\text{Minimize } II; \tag{5.1}$$

$$\text{Minimize} \sum_{i=1}^{n} \sum_{j=0}^{S} j \times X_{i,j}; \tag{5.2}$$

$$\text{Minimize} \sum_{k=1}^{m} C_k \times M_k \tag{5.3}$$

The ILP formulations have the following constraints:

$$M_{k=\text{mem\_port}} \leq \text{Memory\_Port\_Limit}; \tag{6.1}$$

$$\sum_{i=1,\text{Type}(i)=k}^{n} \sum_{p=0}^{\left\lfloor \frac{s-j}{II} \right\rfloor} X_{i,j+p\times II} - M_k \leq 0, \quad \text{for } 0 \leq j < II, 1 \leq k \leq m; \tag{6.2}$$

$$\sum_{j=0}^{S} X_{i,j} = 1, \quad \text{for } 1 \leq i \leq n; \tag{6.3}$$

$$\sum_{j=0}^{S} j \times X_{i,j} - \sum_{j=0}^{S} j \times X_{k,j} \leq -D_i, \quad \text{for all } O_i \rightarrow O_k; \tag{6.4}$$

$$\sum_{j=0}^{S} j \times X_{i,j} + D_i - \sum_{j=0}^{S} j \times X_{k,j} \leq II, \quad \text{for all feedback arc } O_i \rightarrow O_k. \tag{6.5}$$

The objective function in (5.1) minimizes II based on data dependence. The objective function in (5.2) minimizes the total pipeline stage number S. The objective function in (5.3) minimizes the total hardware cost of function units. Here C_k is the cost of function unit of type k. Currently, all C_k are of unit cost, which means the objective function minimizes the total number of function units. Constraint function (6.1) states that number of memory ports in the datapath cannot exceed the system bandwidth limit. Constraint function (6.2) states that at any clock cycle, the active operations in the datapath cannot exceed the number of function units of that type. If II is 1, which means every operation is active at any clock cycle, the minimum number of functions is the number of operations in the datapath. Constraint function (6.3) states that any operation should be schedule to one and only one pipeline stage. Constraint function (6.4) states the normal connections in the datapath. If there is a normal direct connection from O_i to O_k,O_k should be scheduled at least D_i stages below O_i, where D_i is the delay of function unit i. Constraint function (6.5) ensures the II is not violated during operation rescheduling.

The ILP solver is used in three steps as illustrated in the following:

Step 1: Solve objective function (5.1), subject to constraints (6.3)–(6.5);
Step 2: Solve objective function (5.2), subject to constraints (6.1)–(6.5);
Step 3: Solve objective function (5.3), subject to constraints (6.1)–(6.5).

In Step 1, the ILP solver finds the best II for data dependence. The datapath II is then determined by formula (3). This value of II is then used in Step 2 to determine the minimum total pipeline stage number S. Usually, $2S$ (here S is the original minimum value) is enough to solve the ILP problem in practice. In Step 3, the ILP solver determines the optimal scheduling, which needs the smallest number of function units, under the constraint of optimal II and S determined in Steps 1 and 2.

Using the scheduling results from Step 3, operations are allocated to function units in the final datapath. Connections between operations correspond to interconnections between function units. Distributed registers are inserted for pipelined execution. A final datapath is generated to perform the kernel loop execution. If II is 1, the final datapath is the same as the initial datapath from direct mapping, unless there is some scheduling change during the optimization. The interconnections are not changed during the kernel loop execution in this case. If II is larger than 1, the interconnections are dynamically reconfigured at every cycles between the II sets of configuration words. This is because the same function unit can be shared between operations within the same iteration. In this case, for each kernel loop executing on the reconfigurable datapath, multiple configuration words may be necessary for a routing box at different clock cycles (see Figure 2). The number of configuration words equals the value II of this kernel loop. In this case, for each kernel loop, multiple (up to II) reconfiguration contexts may be needed.

For each selected kernel loop, a customized datapath is designed using the methodology described above. The next step of the design process is to merge all those datapaths together to make a single reconfigurable datapath.

5. RECONFIGURABLE DATAPATH DESIGN

The goal of the reconfigurable datapath design is to design a datapath with the minimum number of function units and interconnections, which can be reconfigured at run time to execute different preselected kernel loops at the fastest speed. The design methodology here is to first design the optimal datapath for each kernel loop, as described in Section 4, and then merge them all together to make the reconfigurable datapath. The datapath-merging problem is solved using graph-based techniques. In this section, we first describe how to model the problem and then how to solve it using an algorithm based on a maximum clique formulation.

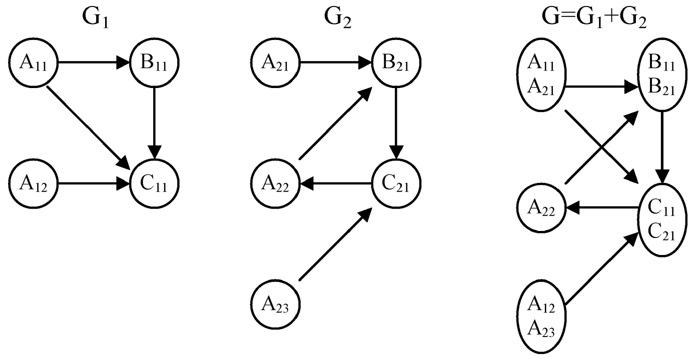

Fig. 7. Graph merging.

5.1 Graph Modeling

Each loop datapath i is modeled as a directed graph $G_i = (V_i, E_i)$, where the vertices in V_i represent the hardware blocks in the datapath, and the arcs in E_i are associated with the interconnections between the hardware blocks. The types of hardware blocks (e.g., adders, multipliers, registers, and so on) are modeled by a labeling function L_i of V_i, such that, for each vertex $u \in V_i$, $L_i(u) = T_{ij}$ is a label that represents the type of the hardware block associated with u. More specifically, we say that vertex u in graph G_i is associated with the jth hardware block of type T. For example, Figure 7 shows two directed graphs G_1 and G_2, corresponding to the two datapaths of loop 1 and 2. Each vertex in G_1 and G_2 is identified by its label. For instance, vertex A_{23} is associated with the third unit of type A in graph G_2.

The design of the reconfigurable datapath is equivalent to constructing a graph G as follows:

1. Create a graph G such that, $G_i \subseteq G$ for all $i = 1, \ldots, N$.
2. The cost of G is least, where the cost of G is $|E|$, the total number of edges in G.
3. The number of vertices of type k in graph G is equal to the maximum number of vertices of type k among all datapath G_i,

$$|V, L(v) = K| = \max\{|V_i, L(v_i) = K|, i = 1, \ldots, N\}.$$

Obviously, this vertex number is the least possible number given Condition 1 above.

Clearly adding any more vertices (hardware blocks of any type, except memory ports) cannot increase parallelism for any kernel loop since each loop datapath is already the optimal result of the design process under a certain constraint (e.g., memory bandwidth).

The vertex set of graph G is easily determined. We pick the maximum number of each type of function unit among all the loop datapaths. The problem now is how to label the vertices associated with all the loop datapath G_i, in other words, how to map the vertices of G_i onto vertices of G in order to minimize the number of interconnections. Figure 7 shows an example mapping.

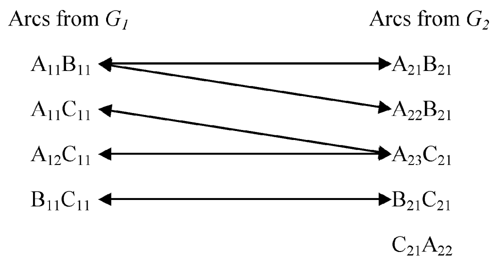

Fig. 8. All possible mappings of arcs from G_1 and G_2.

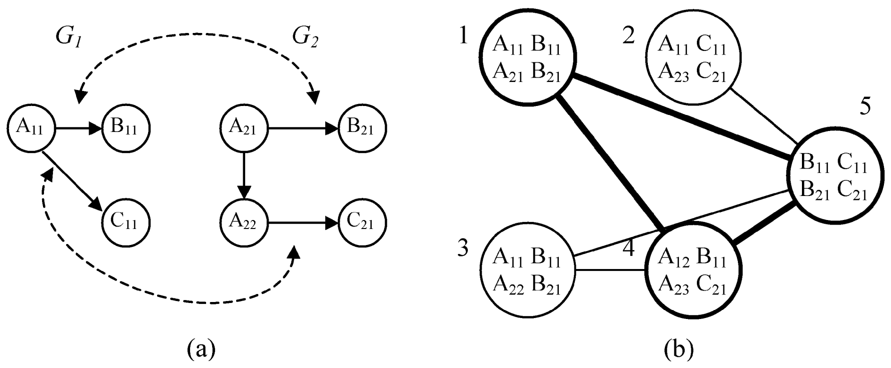
G_1
G_2
A_{11}
B_{11}
C_{11}
A_{21}
B_{21}
A_{22}
C_{21}
(a)
1
A_{11} B_{11} A_{21} B_{21}
2
A_{11} C_{11} A_{23} C_{21}
3
A_{11} B_{11} A_{22} B_{21}
4
A_{12} B_{11} A_{23} C_{21}
5
B_{11} C_{11} B_{21} C_{21}
(b)

Fig. 9. The compatibility graph. (a) Incompatible mappings. (b) Clique on the compatibility graph H.

5.2 The Compatibility Graph

We solve the problem of finding the merged graph G using an arc mapping approach. Initially all possible arc mappings between two graphs G_i and G_j are generated. Two arcs (interconnections), (t, u) and (v, w), from G_i and G_j respectively, can be mapped (overlapped) if and only if $L_i(t) = L_j(v)$ and $L_i(u) = L_j(w)$. In other words, the source vertex of the arcs must have the same label, as well as their destination vertices. Figure 8 lists those arcs from graphs G_1 and G_2 in Figure 7 that can be overlapped. In Figure 8 each mapping is represented by a double-arrow line uniting the arcs that can be overlapped. We represent a possible mapping using a "/", for example, in $(A_{11}, B_{11})/(A_{21}, B_{21})$ arcs (A_{11}, B_{11}) and (A_{21}, B_{21}) can be overlapped.

A compatibility graph H is constructed, where each vertex of H corresponds to a possible mapping of two arcs, one from G_i and another from G_j. There exists an edge between two vertices of H if the arc mappings represented by the vertices are compatible. In order to build the compatibility graph we need to define the notion of mapping compatibility. Two arc mappings are not compatible if and only if they map the same vertex of G_i to two different vertices of G_j, or vice versa. This problem is illustrated in Figure 9(a). In this figure, two loop datapath graphs G_1 and G_2 are shown. There are two possible arc mappings between G_1 and G_2, which are $(A_{11}, B_{11})/(A_{21}, B_{21})$ and $(A_{11}, C_{11})/(A_{22}, C_{21})$. These two mappings are incompatible since they map the same vertex A_{11} from G_1 to two different vertices, A_{21} and A_{22}, in G_2.

By using the compatibility criterion discussed above the compatibility graph H can be easily constructed. Figure 9(b) shows the compatibility graph H resulting from the mappings of arcs from G_1 and G_2 in Figure 8. For example, in mappings $(A_{11}, B_{11})/(A_{21}, B_{21})$ (vertex 1 in H) and $(B_{11}, C_{11})/(B_{21}, C_{21})$ (vertex 5 in H), no vertex from G_1 maps to two distinct vertices in G_2 and vice versa. As a result, these two mappings are compatible, and an edge (1, 5) is required in H. On the other hand, no edge exists in H between vertex 2 and 3. The reason is that the mappings represented by 2 and 3 are incompatible, since A_{11} in G_1 maps to both A_{22} and A_{23} in G_2.

5.3 Maximum Clique Formulation

Each node in the compatibility graph H means a pair of possible vertex mappings, which share the same arc in the merging graph G. To minimize the arcs in the merging graph G, it is necessary to find the maximum number of arc mappings that are compatible with each other. This is actually the problem of finding the maximum clique of the compatibility graph H. For example, in the compatibility graph H in Figure 9(b), a possible maximum clique has vertices 1, 4, and 5. The maximum clique problem is known to be NP-complete [Garey and Johnson 1979]. A heuristic algorithm is used to solve the problem in polynomial time [Battiti and Protasi 2000].

After finding the maximum clique of the compatibility graph, the mapping represented by the vertices from the maximum clique of H is used to construct the merging graph G. Each vertex from the clique gives an arc mapping between G_i and G_j (and their corresponding vertices). Those vertices are mapped together to the same vertices in the graph G and share the same arc in G. After that, the vertices from G_i that were not mapped can be mapped with any vertex of G_j, provided it has the same label and has not been mapped yet. If there are no more such vertices in G_j, the vertex in G_i just maps itself to any unused vertex of the same type in G. The same is done for vertices in G_j.

The solution presented above merges two datapath graphs. In order to merge several graphs, this method is used as a heuristic and applied iteratively as described in Figure 10.

5.4 Reconfigurable Datapath

After datapath merging, the result datapath graph G is used to construct the reconfigurable datapath. Vertices are the selected function units, including registers. Arcs are the interconnections between them. There is a routing box in front of each function unit, as shown in Figure 2, to select inputs for the function units from the various interconnections. During kernel loop execution on the datapath coprocessor, the reconfigurable datapath is reconfigured at every clock cycle. The reconfiguration patterns repeat every II cycles, which means that there are II reconfiguration contexts for each kernel loop. For an application, the total number of contexts is

$$\sum_{i=1}^{N} II_i, \quad \text{for } N \text{ selected kernel loops.}$$

```
procedure Datapath Merging ($G_1, \ldots, G_N, G$)
input:   $N$ directed graphs $G_i = (V_i, E_i)$ and
         labeling functions $L_i : V_i \to T, 1 \le i \le N$
output:  directed graph $G = (V, E)$ and labeling function $L : V \to T$,
         such that $|V|$ and $|E|$ are minimum, and $\forall u \in V$,
         $u$ maps possibly one vertex with label $L(u)$ from each $V_i$

/* Initially, $G$ is the first input graph $G_1$ */
$G \leftarrow G_1$;
$L \leftarrow L_1$;
/* Iteratively merge $G$ with input graph $G_i$ */
for $i \leftarrow 2$ to $N$ do
         $H \leftarrow$ Construct_Compatibility_Graph($G, G_i, L, L_i$);
         $C \leftarrow$ Find_Maximum_Clique($H$);
         $(G, L) \leftarrow$ Reconstruct_Resulting_Graph($C, G, G_i, L, L_i$);
```

Fig. 10. Algorithm datapath merging.

However, the actual number of stored contexts may be far less than this number, since many kernel loops share the same datapath and have the same configuration context. For such cases, only one copy is stored. Since only the interconnections of the datapath coprocessor are reconfigurable, each configuration context is small enough that we can store the entire configuration context for the application into the distributed cache on the coprocessor. During each kernel loop execution, the reconfiguration word of a routing box is preload into the active configuration word in the previous clock cycle as shown in Figure 2. At the next cycle, the routing box switches its configuration word to the active word. The correct interconnections are selected for the function unit. The delay of the routing box is about two-gate delays.

The critical path in the reconfigurable datapath determines the clock speed in the coprocessor. In the reconfigurable datapath, the critical path of a pipeline stage is the critical path of the fixed function unit plus the delay of the routing box plus wire delays. Sometimes a complicated function unit is pipelined inside and executed in multiple clock cycles. Since the functional units are hardwired, their delay is the same as it would be if the functional unit was in the general processor. The same is true for the wire delays. As the routing box just needs to select between its inputs, its delay is just two-gate delays, which is not significant. In a general purpose processor, the critical path is no longer in the functional unit stage, but rather in the more sophisticated branch control and decoding stage. This may be significantly larger than a functional unit delay. In fact, two arithmetic logic units (ALUs) on the Pentium IV processors are clocked at twice the core processor frequency. Thus, it is reasonably safe to assume that the coprocessor clock period is no slower than the master processor clock speed, with the potential for significant speed up.

6. EXPERIMENTAL RESULTS

We have implemented the design methodology of the dynamically reconfigurable datapath described in this paper. The custom loop datapaths for the kernel loops and the final reconfigurable datapath are generated fully automatically. The applications we studied are specified in C. The software is compiled and profiled using the IMPACT compiler. The IR (Lcodes) we used to extract the kernel loop datapaths is generated using the processor platform "*EPIC_1G_1BL*" (EPIC 1-issue 1-branch processor) [Schlansker and Rau 2000], which is representative of today's single-issue speculative processor.

The datapath coprocessor execution time for a kernel loop is estimated using formula (1), where the total pipeline stage number S and loop iteration fetch initial interval II are obtained from the results of the loop datapath design process. The loop iteration number N is obtained from the profiling data from the front-end compilation. The reconfiguration overhead is mainly from the reading and writing of live-in and live-out register values from/to master processor. Usually the two register sets contain 10–20 register for each kernel loop. In our experiments we assume four register values can be accessed each time. Thus, the transfer of 10 register values introduces a three clock cycle overhead.

The execution time is measured in clock cycles. The speed up is calculated under the assumption that the datapath coprocessor is running at the same clock speed as the master general processor.

We did experiments on several applications from Mediabench [Lee et al. 1997], including the MPEG2 video coder and GSM coder. Their results are listed in the following sections.

6.1 MPEG2 Video Coder

The source code and test data files of the MPEG2 encoder are from Mediabench. The data file for test coding is a bit-stream of YUV components (four frames). Those innermost loops with software time more than 1 million cycles are selected as kernel loops. Software time is the execution cycle count on the single-issue speculative processor. There are 23 loops satisfying this execution time requirement of kernel loops. Table I gives the performance analysis of the reconfigurable datapath. In the table, the first "hardware time" with a $'$ stands for the performance of the initial datapaths generated from the direct mapping. The initial datapaths have no hardware resource constraint. The second "hardware time" measures the performance of the final loop datapath. The final datapaths are generated after hardware resource sharing and allocation. The final datapaths contain only regular function units. The experiment of merging function units on the critical path is shown at Section 6.3. The only hardware constraint we used in the experiments is the memory bandwidth. For each loop datapath, the memory port number is limited to 2. The overall hardware time is the sum of the kernel loop execution time on the datapath coprocessor and the execution time of the remaining code on the master processor.

From Table I, we see that the reconfigurable datapath can result in a speed up of the execution of kernel loops from around 2 to 20 times. The overall speed up of the dynamically reconfigurable datapath system against a single-issue processor for the application is 4.49. With no memory bandwidth limit, the

Table I. Performance Analysis for MPEG2 Coder

	Software Time[a]	II'	Hardware Time′	Speedup′	II	Hardware Time	Speedup
Loop 1	820	16	105	7.81	16	104	7.88
Loop 3	85	1	8.29	10.25	3	17.9	4.75
Loop 15	35.7	1	7.57	4.72	1	7.57	4.72
Loop 4	33.5	1	4.12	8.13	2	6.6	5.08
Loop 5	33.1	1	4.08	8.11	2	6.52	5.08
Loop 16	31.4	1	7.03	4.47	1	7.03	4.47
Loop 8	20.2	1	0.625	32.32	2	1.0	20.20
Loop 7	13.6	1	0.675	20.15	5	2.05	6.63
Loop 6	11.5	16	1.42	8.10	16	1.41	8.16
Loop 2	10.6	16	1.32	8.03	16	1.31	8.09
Loop 10	8.3	1	0.270	30.74	2	0.428	19.39
Loop 9	7.31	2	0.856	8.54	2	0.861	8.49
Loop 17	6.08	1	1.28	4.75	2	1.89	3.22
Loop 12	5.23	1	1.26	4.15	1	1.26	4.15
Loop 13	5.07	1	0.52	9.75	1	0.52	9.75
Loop 14	4.46	1	1.01	4.42	2	1.55	2.88
Loop 21	4.05	1	0.224	18.08	5	0.625	6.48
Loop 11	3.86	1	0.201	19.20	2	0.362	10.66
Loop 22	2.57	1	0.495	5.19	1	0.495	5.19
Loop 19	2.35	1	0.192	12.24	1	0.192	12.24
Loop 18	2.19	1	0.271	8.08	3	0.622	3.52
Loop 23	1.45	1	0.28	5.18	2	0.438	3.31
Loop 20	1.1	1	0.174	6.32	1	0.174	6.32
Rest	117	–	117	1	–	117	1
Overall	1266	–	264	4.80	–	282	4.49

[a]Software time and hardware time in million cycles.

intermediate datapaths run only slightly faster. The reason is that only a small number of kernel loops have $II_{\text{mem}} > II_{\text{dep}}$, which means (1) lots of loops have no more than two memory operations; (2) loops which have more than two memory operations suffer from data dependence too.

Table II shows the datapath details for the 23 kernel loops. Again, column names with a $'$ stand for the data for the initial datapaths obtained from direct mapping. As stated above, the memory port number of the final datapath is no more than 2. In the table, FU stands for real function units such as adders, multipliers, and so on, not including memory ports and registers. $|E|$ stands for the interconnection number.

After the datapath merging, the reconfigurable datapath has 176 nodes and 732 arcs (interconnections). Among the 172 nodes, 2 are memory ports, 41 are function units, 96 are pipeline registers, and 37 are constant registers.

An estimation of total reconfiguration bit size for programming the interconnections is about 1.1K bits. This is the size of one configuration context for the interconnections. For the MPEG application, the total number of configuration contexts for the 23 kernel loops is $\sum II = 89$. However, many kernel loops share the same datapath, for example, loops 1, 2, and 6 (in Table II). The total number of different contexts is actually 54. We believe it is very reasonable to store the 60K bits required for this in the distributed cache.

Table II. Loop Datapaths for MPEG

	MEM_PORT #′	FU′	RG′	$\|E\|'$	MEM_PORT #	FU	RG	$\|E\|$[a]
Loop 1	32	83	62	308	2	7	4	80
Loop 2	32	83	62	308	2	7	4	80
Loop 3	5	11	8	41	2	7	3	36
Loop 4	3	9	5	30	2	7	3	28
Loop 5	3	9	5	30	2	7	3	28
Loop 6	32	83	62	308	2	7	4	80
Loop 7	9	28	54	129	2	10	12	96
Loop 8	3	24	186	244	2	18	94	242
Loop 9	3	24	65	121	2	15	34	119
Loop 10	3	23	174	230	2	16	90	227
Loop 11	3	15	34	71	2	10	19	71
Loop 12	1	5	2	14	1	5	2	13
Loop 13	2	7	1	19	2	7	1	17
Loop 14	3	3	8	21	2	2	4	20
Loop 15	2	5	1	15	2	5	1	15
Loop 16	2	4	0	13	2	4	0	13
Loop 17	4	4	15	32	2	2	8	28
Loop 18	6	5	12	36	2	2	5	29
Loop 19	2	10	2	26	2	10	2	24
Loop 20	2	4	0	12	2	4	0	11
Loop 21	9	26	55	125	2	9	12	85
Loop 22	1	5	2	14	1	5	2	13
Loop 23	3	4	8	23	2	3	4	22

[a] $|E|$ = interconnection number.

6.2 GSM Coder

GSM (Global System for Mobile communication) is one of the most popular wireless communication standards. In this section, we present experiment results for the GSM encoder, which compresses speech signals. More specifically, GSM 06.10 compresses frames of 160 16-bit linear samples into 33-byte frames. Both the source code and test data in this experiment are from Mediabench. Table III shows the speedup of the datapath coprocessor against a single-issue speculative processor. Again, those table heads with a ′ stand for the data for the initial datapath obtained using direct mapping. The final datapaths are constructed with regular function units under the memory bandwidth constraint of two memory ports. Those innermost loops with software time more than 100K cycles are selected as kernel loops. For the 21 kernel loops, the speed up of the final loop datapaths ranges from around 2 to 20 times. The overall speed up for the GSM application is 4.02.

Table IV gives the details of the 21 loop datapaths. For the final datapaths, the memory port number is limited to no more than 2. After merging the 21 loop datapaths, the reconfigurable datapath has 164 nodes and 935 arcs (interconnections). Among the 164 nodes, 2 are memory ports, 43 are function units, 41 are pipeline registers, and 90 are constant registers.

An estimation of the total reconfiguration bit size for programming the interconnections is about 1.4K bits. The total number of configuration contexts for the 21 kernel loops is $\sum II = 67$. There are 60 distinct sets of contexts that need

Table III. Performance Analysis for the GSM Coder

	Software Time[a]	II'	Hardware Time$'$	Speedup$'$	II	Hardware Time	Speedup
Loop 9	610	1	5.09	119.8	20	61.2	9.97
Loop 21	333	7	91.8	3.63	3	59.2	5.63
Loop 1	57.6	1	2.21	26.06	5	8.12	7.09
Loop 12	42.1	1	1.48	28.45	5	7.08	5.95
Loop 4	28.5	1	1.51	18.87	2	2.61	10.92
Loop 3	20.7	1	1.84	11.25	2	3.32	6.23
Loop 7	20.7	1	1.70	12.18	1	1.70	12.18
Loop 11	20.6	1	1.53	13.46	1	1.53	13.46
Loop 14	14.0	1	1.48	9.46	1	1.48	9.46
Loop 13	12.6	1	1.48	8.51	1	1.48	8.51
Loop 10	11.9	1	1.73	6.88	1	1.73	6.88
Loop 15	10.5	5	5.61	1.87	5	5.61	1.87
Loop 8	10.4	1	1.73	6.01	2	1.73	6.01
Loop 16	9.42	1	1.23	7.66	4	1.87	5.04
Loop 17	6.79	1	0.701	9.69	1	0.701	9.69
Loop 5	6.19	1	0.328	18.87	2	0.569	10.88
Loop 20	4.28	1	0.701	6.11	2	0.996	4.30
Loop 2	2.95	1	0.701	4.21	1	0.701	4.21
Loop 18	2.08	1	0.194	10.72	3	0.323	6.44
Loop 19	2.08	1	0.194	10.72	3	0.323	6.44
Loop 6	1.91	1	0.101	18.91	2	0.172	11.10
Rest	191	–	191	1	–	191	1
Overall	1419	–	314	4.52	–	353	4.02

[a]Software time and hardware time in 100K cycles.

Table IV. Loop Datapaths for GSM

	MEM_PORT #$'$	FU$'$	RG$'$	$\|E\|'$	MEM_PORT #	FU	RG	$\|E\|$[a]
Loop 1	10	27	78	153	2	11	18	120
Loop 2	2	3	3	14	2	3	3	14
Loop 3	3	11	25	55	2	7	13	53
Loop 4	4	13	37	73	2	10	19	70
Loop 5	4	13	37	73	2	10	19	70
Loop 6	4	13	37	73	2	10	19	70
Loop 7	1	10	6	30	1	10	6	30
Loop 8	2	3	4	15	2	3	4	15
Loop 9	40	164	461	870	2	12	29	441
Loop 10	1	5	0	12	1	5	0	11
Loop 11	1	10	6	30	1	10	6	30
Loop 12	9	20	51	109	2	5	11	67
Loop 13	2	5	6	21	2	5	6	21
Loop 14	2	5	6	21	2	5	6	21
Loop 15	0	12	13	39	0	7	4	34
Loop 16	8	29	55	133	2	11	16	107
Loop 17	1	10	6	30	1	10	6	30
Loop 18	5	23	47	105	2	13	18	98
Loop 19	5	23	40	98	2	13	15	91
Loop 20	3	41	79	176	2	23	41	160
Loop 21	3	25	77	134	2	11	27	114

[a]$|E|$ = interconnection number.

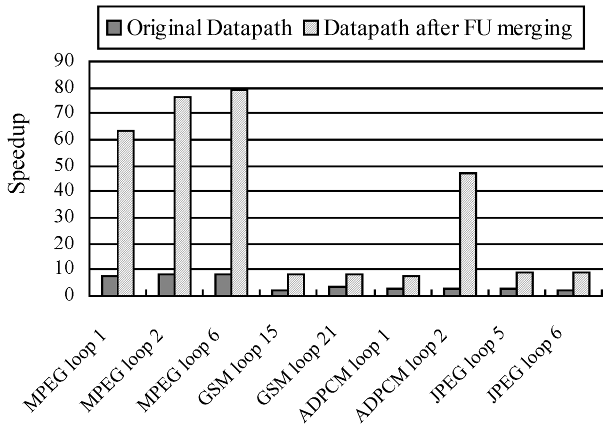

Fig. 11. Performance of datapaths before/after FU merging.

to be stored. The total size of contexts that need to be stored in the distributed cache is about 84K bits. Again, this is very reasonable for storage in the on-chip distributed memory.

6.3 Function Unit Merging on the Critical Path

We did the function unit merging experiment on several benchmarks from Mediabench. Not all the kernel loop datapaths are affected by function unit merging, for example, those loop datapaths with $II = 1$. Figure 11 shows some loops from several benchmarks that affected by function unit merging on the critical path. In Figure 11, "original datapath" represents the initial datapath after direct mapping. FU merging experiments are based on the initial datapaths.

In Figure 11, FU merging on critical path brings significantly more speed up for the loop datapaths. As described in Section 4.5, the reason is FU merging greatly reduces the value of II, which is the most important factor in determining the loop datapath performance.

The experimental results shown in Figure 11 give the largest possible performance improvement from FU merging process. The maximum improvement can be gained when the assumption on the clock period of the special function units holds, that is, the delay of the special function unit is the maximum delay among each function unit in the merging set. Usually, the number of FUs in the merging set is a good sign of how reasonable this is. In Table V, the FUs in the merging sets of the nine loops are listed.

In Table V, JPEG loop 5 and loop 6 only need to reschedule function units to minimize II down to 1. Very likely, set 1 and set 2 of loop 1 in ADPCM coder cannot find any special function unit replacements. However, replacing part of the function units in the merging set can still gain speed up, although not as much as in Figure 11. The three MPEG loops have the same merging set as shown in Figure 12(a). Although there is no such special function unit with a delay of 1 cycle to replace the merging set, after doing some datapath transform, the II can still be reduced to 1, as shown in Figure 12(b). Furthermore, an optimized version of the datapath is shown in (c), which has only a 5 cycle

Table V. Function Unit Merging Sets

	Set 1	SET 2	Set 3
MPEG loop 1	16 adders		
MPEG loop 2	16 adders		
MPEG loop 6	16 adders		
GSM loop 15	4 shifters, 1 adder, 1 MUX	1 shifter, 1 comparator, 1 sub, 1 MUX	
GSM loop 21	2 adders, 2 comparators 2 adders, 2 MUX		
ADPCM Coder loop 1	4 subs, 5 comparators, 7 MUXs, 3 shifters, 3 ors	1 shifter, 4 adders, 6 MUXs, 3 comparators, 1 sub	1 adder, 2 comparators, 2 MUXs, 1 shifter
ADPCM decoder loop 1	1 adder, 2 comparator, 2 MUXs	1 sub, 1 adder, 2 comparators, 3 MUXs	
JPEG loop 5	3 MUXs		
JPEG loop 6	1 adder, 1 MUX	1 adder, 1 MUX	

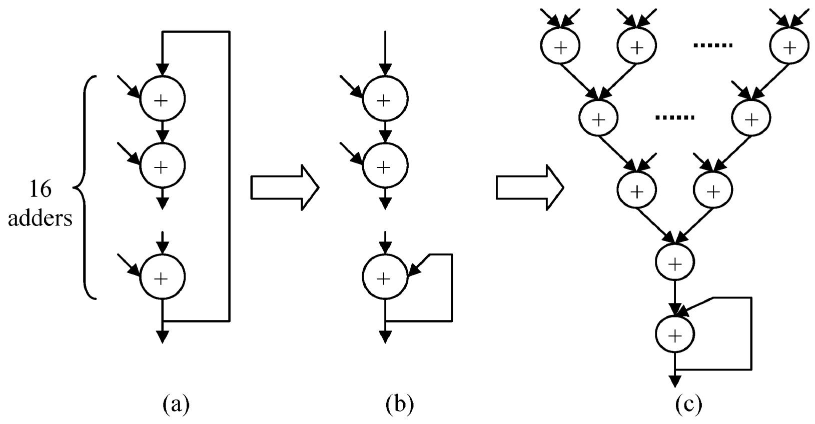

Fig. 12. Datapath transforms for the merging set.

total delay instead of 16 cycles as in (a) and (b). As shown above, merging sets are important for improving the datapath performance. There are various ways to optimize loop datapaths, by replacing function units in the merging set with special function units, rescheduling function units in merging set, or doing datapath transforms for the merging sets. The goal is to reduce the impact of data dependence. Usually, this needs to be combined with improving memory systems. For example, in the three MPEG loops shown above, the best performance in Figure 11 is achieved with 32 memory ports in the datapath. Although we can improve the II_{dep} to 1, if the memory bandwidth is limited to two ports, which means $II_{\text{mem}} = 16$, there is no way to improve the performance from the original datapath.

7. CONCLUSION AND FUTURE WORK

This paper presents a design methodology for a dynamically reconfigurable datapath coprocessor for a specific application (domain). We first describe how to construct a datapath for a selected kernel loop with the goal of exploiting maximum operation level parallelism, and then how to merge all kernel loop datapaths to make a single reconfigurable datapath. Through the two case studies on an MPEG2 video coder and a GSM coder, we demonstrate how significant speedups compared to a single processor execution can be obtained.

We also show some preliminary experimental results for function units merging on the critical path for a datapath. Introducing complex function units lowers the potential of sharing hardware resources among kernel loops, but such complex function units that replace regular function units on critical paths can significantly improve the datapath performance.

In this paper, we have been quite conservative about the parallelism and speed of the coprocessor. We have assumed that the coprocessor clock speed is the same as the master general processor, a fact that we have argued is likely to be conservative. A detailed analysis is needed to determine how much faster the clock on the coprocessor can be compared to the master processor. A faster clock speed can bring further speedup. In the experiments, we have limited the memory bandwidth into two memory ports. Today's memory system design for embedded systems can bring larger memory bandwidth, for example, by introducing multiple memory banks. Also, we have not implemented any sophisticated compilation techniques to reduce the memory operations in a loop body. Such improvements can further add to the speedup that can be obtained.

To design a datapath coprocessor from selected kernel loops ensures the optimal performance of such kernel loops on the datapath coprocessor. To bring additional design flexibility to such a datapath coprocessor system, it is very interesting to develop techniques to compile a new loop onto an existing reconfigurable datapath as described here. With such compiler techniques, a minor change in the application kernels will not require a change of the system hardware, but rather a change of the configuration contexts. For this additional flexibility to be exploited, additional flexibility in constructing datapaths on the fly may be required in the form of some general functional units as well as interconnections. The design of these additional "flexible" components in an existing application-specific reconfigurable datapath bears further study.

REFERENCES

Battiti, R. and Protasi, M. 2000. Reactive local search for the maximum clique problem. *Algorithmica 29*, 4, 610–637.

Chang, P. P., Mahlke, S. A., Chen, W. Y., Warter, N. J., and Hwu, W. W. 1991. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture*. 266–275.

Ebeling, C., Cronquist, D., and Franklin, P. 1996. RaPiD—Reconfigurable Pipelined Datapath. In *The 6th International Workshop on Field-Programmable Logic and Applications*.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

Goldstein, S. C., Schmit, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R., and Laufer, R. 1999. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*. 28–39.

Hauser, J. R. and Wawrzynek, J. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '97). 24–33.

Hauck, S., Fry, T. W., Hosler, M. M., and Kao, J. P. 1997. The chimaera reconfigurable functional unit. In *IEEE Symposium on Field-Programmable Custom Computing Machines*.

Huang, Z. and Malik, S. 2001. Managing dynamic reconfiguration overhead in system-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In *Proceedings of the Design Automation and Test in Europe, Conference*, Munich, Germany, 735–740.

Huang, Z. and Malik, S. 2002. Exploiting operation level parallelism through dynamically reconfigurable datapaths. In *Proceedings of the 39th Design Automation Conference*, New Orleans, LA, 337–342.

Keutzer, K., Malik, S., Rabaey, J. M., Newton, A. R., and Sangiovanni-Vincentelli A. 2000. System level design: Orthogonolization of concerns and platform-based design. *IEEE Trans. Comput.-Aided Des. 19*, 12.

Lam, M. 1998. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, Atlanta, GA.

Lee, C., Potkonjak, M., and Mangione-Smith, W. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, NC, 330–335.

Lee, J.-H., Hsu, Y.-C., and Lin, Y.-L. 1989. A new integer linear programming formulation for the scheduling problem in data path synthesis. In *Proceedings of IEEE ICCAD 89*, Santa Clara, CA, 20–23.

Moreano, N., Araujo, G., Huang, Z., and Malik, S. 2002. Datapath merging and interconnection sharing for reconfigurable architectures. In *Proceedings of the 15th International Symposium on System Synthesis*, Kyoto, Japan, 38–43.

Rau, B. 1996. Iterative modulo scheduling. *Int. J. Parallel Process. 24*, 1.

Razdan, R. and Smith, M. 1994. High-performance microarchitectures with hardware-programmable functional units. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*. 172–180.

Rupp, C. R., Landguth, M., Garverick, T., Gomersall, E., Holt, H., Arnold, J. M., and Gokhale, M. 1998. The NAPA adaptive processing architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines*.

Salefski, B. and Caglar, L. 2001. Re-configurable computing in wireless. In *Proceedings of the 38th Design Automation Conference*, Las Vegas, Nevada, 178–183.

Schlansker, M. S. and Rau, B. R.. 2000. EPIC: Explicitly parallel instruction computing. *IEEE Comput. 33*, 2 (Feb.), 37–45.

Wan, M., Zhang, H., George, V., Benes, M., Abnous, A., Prabhu, V., and Rabaey, J. 2000. Design methodology of a low-energy reconfigurable single-chip DSP system. *J. VLSI Signal Process.*

Wang, A., Killian, E., Rowen, C., and Maydan, D. 2001. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of the 38th Design Automation Conference*, Las Vegas, Nevada, 184–188.

Received January 2003; revised July 2003; accepted August 2003