An Energy-Efficient Patchable Accelerator For Post-Silicon Engineering Changes

Hiroaki Yoshida
VLSI Design and Education Center
The University of Tokyo
CREST, Japan Science and Technology Agency
hiroaki@cad.t.u-tokyo.ac.jp

Masahiro Fujita
VLSI Design and Education Center
The University of Tokyo
CREST, Japan Science and Technology Agency
fujita@ee.t.u-tokyo.ac.jp

ABSTRACT

With the shorter time-to-market and the rising cost in SoC development, the demand for post-silicon programmability has been increasing. Recently, programmable accelerators have attracted more attention as an enabling solution for post-silicon engineering change. However, programmable accelerators suffers from 5~10X less energy efficiency than fixed-function accelerators mainly due to their extensive use of memories. This paper proposes a highly energy-efficient accelerator which enables post-silicon engineering change by a control patching mechanism. Then, we propose a patch compilation method from a given pair of an original design and a modified design. Experimental results demonstrate that the proposed accelerators offer high energy efficiency competitive to fixed-function accelerators and can achieve about 5X higher efficiency than the existing programmable accelerators.

Categories and Subject Descriptors

B.5.2 [**REGISTER-TRANSFER-LEVEL IMPLEMENTATION**]: Design Aids—*Automatic synthesis*; C.1.3 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; C.5.4 [**COMPUTER SYSTEM IMPLEMENTATION**]: VLSI Systems; D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Code generation; Incremental compilers; Retargetable compilers*

General Terms

Algorithms, Design, Performance

Keywords

Engineering change, high-level synthesis, energy efficiency

1. INTRODUCTION

High-level synthesis has become a key technology in SoC development to achieve a short turn-around time and a low design cost. Fixed-function accelerators synthesized by such a technology offer 100~1000X more energy efficiency than general-purpose processors, and hence used in many embedded application domains to meet both high performance and high energy efficiency requirements. For example [2], an ASIC implementation of OFDM receiver, which is one of the central technologies in the next generation mobile phone, can achieve the efficiency of 200GOPS/W (5pJ/op) in a 90nm technology. On the other hand, efficient embedded processors achieve 4GOPS/W (250pJ/op, 50X more energy than ASIC) and mobile general-purpose processors achieve 0.04GOPS/W (25nJ/op, 5,000X more energy than ASIC). Thus, fixed-function accelerators are becoming increasingly important. According to ITRS 2009 Update [11], an SoC will have more than 1,000 accelerators in the next decade (Figure 1).

Due to extremely high non-recurring-engineering costs in ASIC development, the engineering change (EC) methodology has been utilized to make a design change at the very end of the design process. Engineering change typically takes place due to bug fixes and design specification changes. Figure 2 shows an example of en-

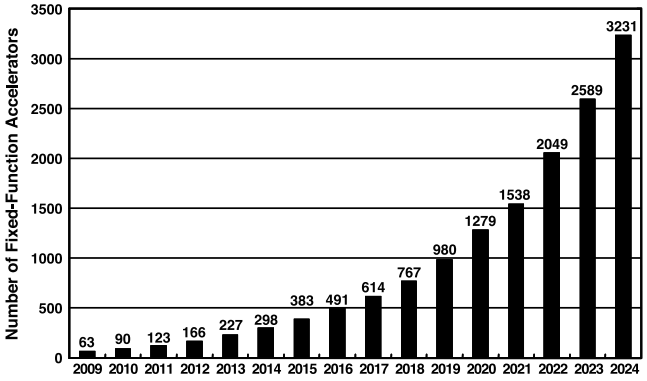


Figure 1: Trends on the number of fixed-function accelerators on consumer portable SoCs [11].

```
            ...
for(k=0;k<N4;k++) {
            ...
  X_out[     n]= RE(x);
  X_out[N2-1-n]=-IM(x);
  X_out[N2+  n]= IM(x);
  X_out[N -1-n]=-RE(x);
}
            ...
```

(a) Version 1.33

```
            ...
for(k=0;k<N4;k++) {
            ...
  X_out[     n]=-RE(x);
  X_out[N2-1-n]= IM(x);
  X_out[N2+  n]=-IM(x);
  X_out[N -1-n]= RE(x);
}
            ...
```

(b) Version 1.34

Figure 2: An example of engineering change: bug-fix to `mdct.c` in FAAD2.

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

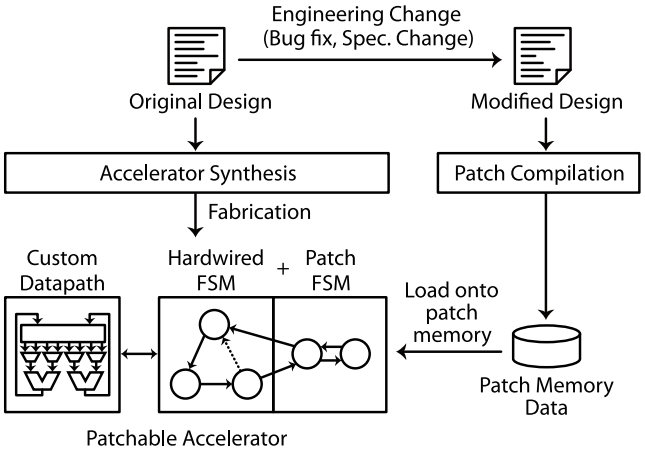
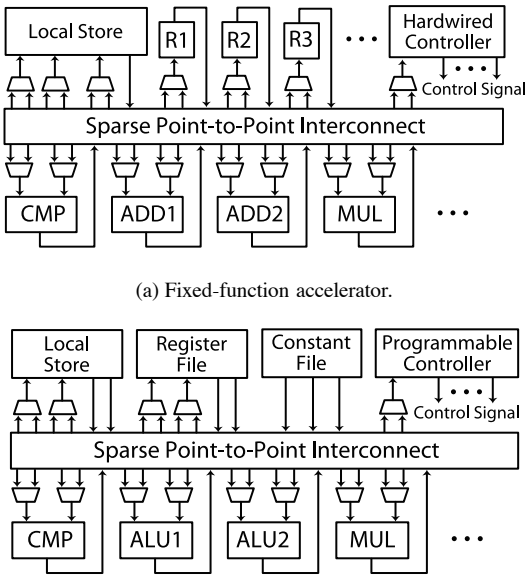


Figure 3: Design flow of the proposed patchable accelerator.

gineering change [4] from FAAD2 [3], an open-source Advanced Audio Coding (AAC) decoder software. From Version 1.33 to Version 1.34, a bug is fixed by changing the signs of expressions. Even though engineering changes affect a limited portion of a design, any post-silicon EC requires a significant amount of design and fabrication efforts. Consequently, the demand for post-silicon programmability has been increasing. Recently, programmable accelerators have attracted more attention as an enabling solution for post-silicon engineering change. However, programmable accelerators suffer from low energy efficiency mainly due to their extensive use of memories in their controllers and the centralized register files.

In this paper, we propose a novel patchable accelerator which can achieve both high performance and high energy efficiency. Since EC affects only a limited portion of a design, we employ a patching mechanism instead of using a horizontal microcoded controller. Figure 3 shows a design flow of the proposed patchable accelerator. Given a high-level description of an original design, a patchable accelerator consisting of custom datapath, hardwired FSM and patch FSM is synthesized. When engineering change takes place due to bug fixes or specification changes after fabrication of the chip, a patch data is compiled from the modified design description. By loading the data onto the patch memory in the patchable accelerator, the accelerator behaves as described in the modified design description. Thus, a respin (*i.e.* a re-fabrication of the chip) can be avoided and hence a time-to-market and a development cost can be dramatically reduced. Since the proposed accelerator is an enhancement of a fixed-function accelerator, the proposed approach can be applied to any fixed-function accelerator generated by typical high-level synthesis tools. The tradeoff between efficiency and programmability can be made by controlling the amount of patch memory. Then, we propose a patch compilation technique from a given pair of an original design and a modified design. Experimental results demonstrate that the proposed accelerator can achieve a higher energy efficiency than the existing programmable accelerators. The main contributions of this paper are as follows.

- A novel energy-efficient patchable accelerator enabling post-silicon ECs (Section 3).
- A practical patch compilation method (Section 4).
- A comparison of energy efficiency between the proposed accelerator and the existing accelerators (Section 5).



(a) Fixed-function accelerator.

(b) Programmable accelerator.

Figure 4: Architecture templates of the conventional accelerators.

2. RELATED WORK

In this section, we review the prior work in the following two categories:

Programmable Accelerators

There have been several attempts to introduce programmability to fixed-function accelerators. No-Instruction-Set Computer (NISC) [9] consists of a programmable controller and a custom datapath. The controller is a horizontal microcoded controller consisting of a control memory and a state register. The functionality of the accelerator can be modified by changing the content of the control memory. Since highly-customized datapaths have limited connections between FUs and registers, centralized register files are introduced to increase flexibility. Programmable Loop Accelerator (PLA) [5] offers further flexibility introducing several techniques such as MOV operation, global bus, port swapping, and so forth. Although these programmable accelerators are shown to be more efficient than embedded processors, they are still 5~10X less efficient than fixed-function accelerators [4]. This is mainly due to their extensive use of memory in the controller and the centralized register files as shown in our experimental results.

Energy-Efficient Processors

As modern processors continue to integrate more processors on a chip, *utilization wall* problem [13] has emerged as a critical issue which limits the fraction of the processors one can use at the same time. To overcome this utilization wall problem, energy-efficient processors such as ELM [2] and AnySP [14] have been proposed. Also, HiveLogic Platform from Silicon Hive [12] provides a highly-customized VLIW architecture with distributed register files. While these processors can cover a broad spectrum of applications, they are still less efficient than a fixed-function accelerator for a specific application.

More recently, conservation cores [13] has been proposed to reduce the energy further by incorporating specialized accelerators tightly coupled with each processor. Also, Hameed *et al.* pro-

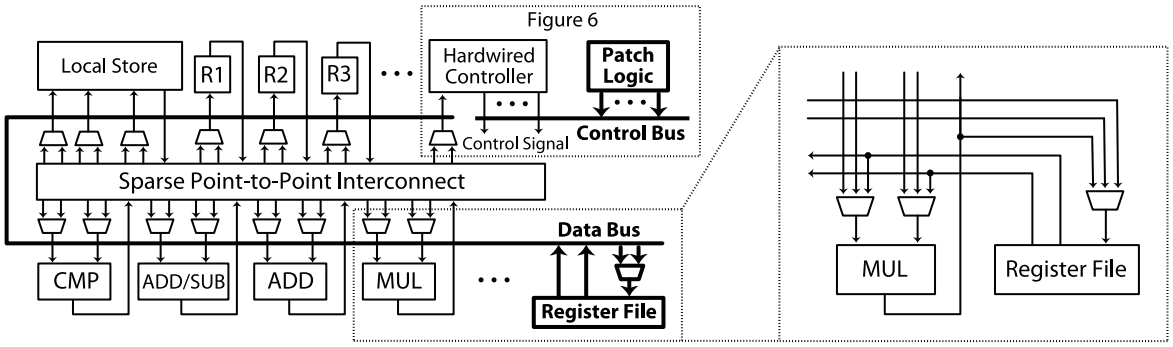


Figure 5: Architecture template of the proposed patchable accelerator.

vided an thorough study on the sources of inefficiency in processors and concluded that processors can achieve ASIC-like efficiency by bringing in application-specific customized accelerators [6]. These studies clearly demonstrate that application-specific customization is the key to achieving a high energy efficiency.

3. ACCELERATOR ARCHITECTURE

3.1 Fixed-Function Accelerator

In this section, we explain a fixed-function accelerator since our accelerator is an enhancement of a fixed-function accelerator. Figure 4 (a) shows an architecture template of a fixed-function accelerator. It consists of functional units (FUs), interconnects between FUs, a hardwired controller. Each FU performs a pre-defined set of operations with respect to its type. Typical FU types are an adder (ADD), a subtractor (SUB), a multiplier (MUL), a comparator (CMP), and a shifter (SHFT). A local store (LS) is a RAM to hold the values of arrays and global variables. An LS is also used to communicate with external hardware components. An LS has two types of ports; one for the address and the other for the data. A register (R1, R2, . . .) is used to hold the value in a variable. We model every write or read port of an LS and a register as a distinct FU. Thus, any access to a memory unit can be scheduled and bound in the same way as other functional units. These FUs are connected by sparse point-to-point interconnect consisting of multiplexers (MUXes) and wires. Each FU input is connected to either the output of a functional unit or the output of a multiplexers. Similarly, each FU output is connected to some inputs of FUs and multiplexers through a wire. The inputs of a multiplexer are connected to outputs of functional units. Using multiplexers, each FU input can select the input signal. A hardwired controller is a hardwired logic implementation of a finite state machine (FSM) which generates the control signals for FUs and multiplexers. The controller has an input which determines the state transition, which realizes an *if-then-else* controlling mechanism.

3.2 Programmable Accelerator

An architecture template of a programmable accelerator is shown in Figure 4 (b). Compared to a fixed-function accelerator, a hardwired controller is replaced with a programmable controller which is a horizontal microcoded controller consisting of an instruction memory and a program counter which holds the current state. Each address in the memory corresponds to a state and the corresponding data includes the control signals for the state as well as the set of next states. Also, a register file and a constant file (*i.e.* constant generator) are introduced to increase the flexibility. In this way, a programmable accelerator extensively uses memories to offer high programmability and flexibility.

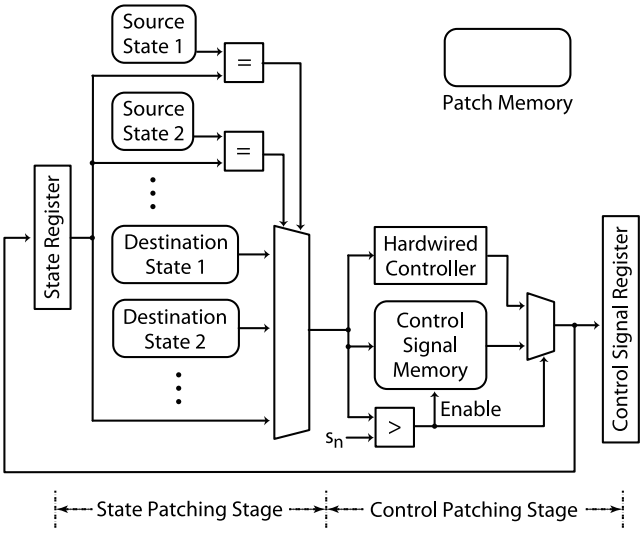
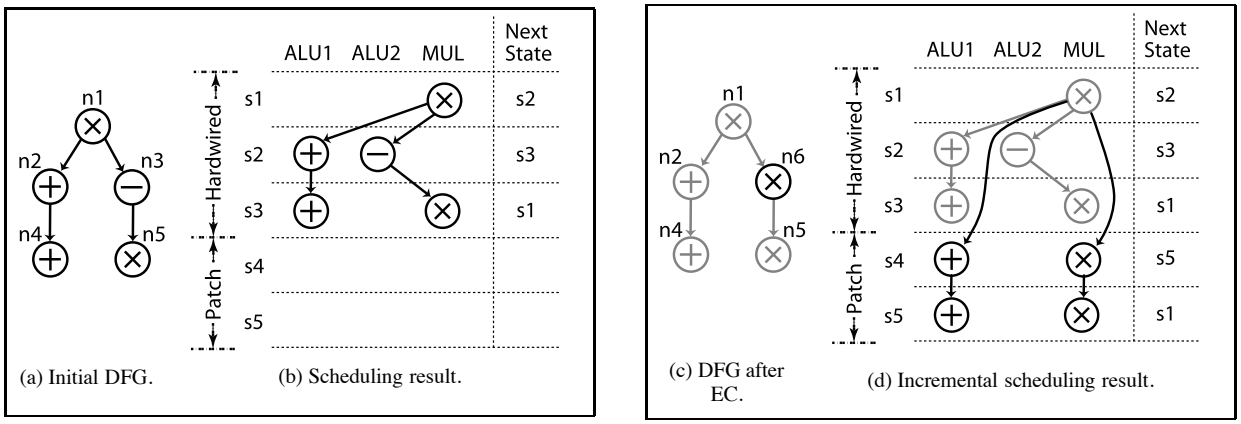


Figure 6: Patch logic.

3.3 Proposed Patchable Accelerator

In spite of a high programmability of a programmable accelerators, it is unnecessary to offer full programmability particularly for engineering changes. The basic idea of our accelerator architecture is to offer the minimum programmability for engineering changes by incorporating a small amount of memory into a fixed-function accelerator. The proposed accelerator can achieve high energy efficiency by using a hardwired controller for unmodified states. Besides, the degree of programmability can be controlled by the amount of extra memory.

As mentioned earlier, our proposed accelerator is an enhancement of a fixed-function accelerator. An architecture template of a patchable accelerator is shown in Figure 5. To enable post-silicon engineering changes, we add two components: *a patch logic* and *a register file*. A patch logic modifies the control signals of some FSM states. The details are explained in the next section. A multiport register file (RF) is used when the datapath registers are not available. A patch logic and a register file are connected to a fixed-function accelerator through global buses, which is a similar idea to PLA [4]. There are two global buses: one is a control bus for



Index	Source	Destination
1	s2	s4
2	––	––

(e) State conversion table.

Figure 7: An illustrative example of patching mechanism.

patching control signals and the other one is a data bus for the communication between FUs and a register file. As shown in Figure 5, a data bus increase only one input to each multiplexer, the performance overhead is low. Furthermore, FUs may be enhanced to increase the flexibility. For example, one of the adders in Figure 5 is replaced with an adder/subtractor. If the original datapath does not have essential types of FUs, they may be added and connected through the data bus.

We would like to note that loop accelerators such as PICO-NPA [10] and PLAs [5] have slightly different architectures to execute software-pipelined loops. Though this paper focuses on a simplified architecture in Figure 5 for ease of explanation, an extension to such loop accelerators should be straightforward.

3.4 Patch Logic

A patch logic can modify control signals for several states. For unpatched states, the control signals are generated from the hardwired controller. As shown in Figure 6, the patch logic consists of a state patching stage and a control patching stage. Suppose that the hardwired controller implements the states $\{s_1, ...s_n\}$ and the control signal memory contains the control signals for the states $\{s_{n+1}, ..., s_{n+m}\}$. The state patching stage converts a subset of the hardwired controller states to the patch memory states $\{s_{n+1}, ..., s_{n+m}\}$. If the converted state corresponds to the patch memory state, the control signals are generated from the patch memory.

Using an example in Figure 7, we explain the patching mechanism. Suppose that the datapath has two ALUs and one multiplier. An initial dataflow graph (DFG) in Figure 7 (a), the scheduling result is Figure 7 (b). The hardwired controller implements $s1$, $s2$ and $s3$, and the state transition is s1→s2→s3 →s1→ ⋯ . Next, the dataflow graph after EC is shown in Figure 7 (c) where a subtractor is changed into a multiplication. Since this change corresponds to $s2$, the state needs to be re-scheduled. Thus, a new state $s4$ in the patch logic is introduced as shown in Figure 7 (d). The scheduling result is stored in the control signal memory, and $s2$ is converted to $s4$ as shown in Figure 7 (e). Since the input of the operation $n5$ has been changed, the state $s3$ is also re-scheduled. After patching, the state transition is s1→s4→s5→s1→ ⋯ .

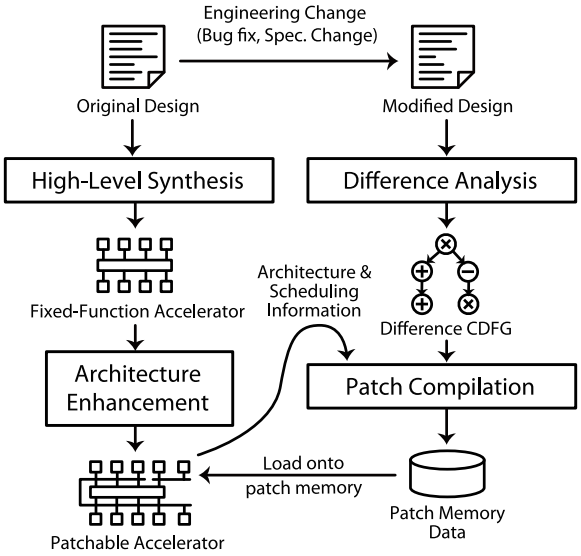


Figure 8: Detailed design flow.

To prevent a performance degradation due to the patching mechanism, the controller is pipelined by introducing control signal registers at the output as shown in Figure 6. Such a control-pipelined execution can be achieved by scheduling every branch instruction one control step ahead.

4. PATCH COMPILATION METHOD

4.1 Overall Flow

The design flow of patchable accelerators is shown in Figure 8. Given an original design description of an application, a fixed-function accelerator is generated by high-level synthesis. Then, a patchable accelerator is generated by enhancing the fixed-function accelerator as explained in Section 3.3. When engineering change takes place due to bug fixes or specification changes after fabrication of the chip, the original design description is modified accord-

ing to the engineering change. Then, a difference CDFG, which will be explained in the next section, is computed by analyzing a textual difference between the original design description and the modified design description. The patch compiler takes the difference CDFG and the accelerator architecture, a patch memory data is compiled. By loading the data onto the patch memory in the patchable accelerator, the accelerator behaves as described in the modified design description. The remainder of this section explains the patch compilation method. Basically, the patch compilation is performed by incrementally scheduling and binding each modified operation. We first formulate the incremental scheduling and binding problem and then explain how to solve the problem in detail.

4.2 Problem Formulation

Given a high-level description of an application, a control data flow graph (CDFG) is constructed by analyzing the description. It is assumed that the underlying expressions are of a static single assignment (SSA) form. A CDFG consists of a control flow graph (CFG): $G_C = (V_C, E_C)$ and a data flow graph (DFG): $G_D = (V_D, E_D)$. A CFG consists of control nodes V_C and control edges E_C where each control node corresponds to a basic block and each control edge represents a control flow between two control nodes. A basic block includes one or more operation nodes and does not include any conditional execution. A DFG consists of operation nodes V_D and data edges E_D where each operation node corresponds to an operation and each data edge represents a data dependency between operations.

Design descriptions before and after EC are represented as a *Difference-CDFG (Δ-CDFG)* which is a single CDFG structure combining two CDFGs before and after EC. In a Δ-CDFG, the set of operation nodes V_D is a union of four disjoint sets $V_D = V_F \cup V_N \cup V_R \cup V_M$: a set of unmodified operations V_F, a set of added operations V_N, a set of removed operations V_R, and a set of modified operations V_M. A modified operation is an operation such that any of its inputs is a newly-added operation. Hence, a modified operation requires neither re-scheduling nor re-binding but the corresponding control signals need to be modified. Now, a set of operations before EC is $V_F \cup V_R \cup V_M$ and a set of operations after EC is $V_F \cup V_N \cup V_M$. For example, the operations in Figure 7 are partitioned as follows: $V_F = \{n1, n2, n4\}$, $V_N = \{n6\}$, $V_R = \{n3\}$ and $V_M = \{n5\}$. The CFG in a Δ-CDFG is equivalent to the CFG after EC. The operations which do not exist after EC, V_R, do not have any corresponding basic block. Similarly, the data edges E_D in a Δ-CDFG are equivalent to the data edges of the DFG after EC. That is, the removed operations V_R have neither incoming edges nor outgoing edges. Those edges are unnecessary because the removed operations will be removed during the patch compilation.

$U = \{s_1, s_2, ...\}$ is a set of the states in the controller. A datapath $D = (G, P)$ consist of a set of FUs $G = \{f_1, f_2, ...\}$, and a set of registers $P = \{r_1, r_2, ...\}$ including the registers in the register file. A schedule $S : V_D \rightarrow U$ maps each operation node onto a state. An FU bind $F : V_D \rightarrow G$ maps each operation node onto an FU, and a register bind $R : V_D \rightarrow P$ maps each operation node onto a register. For each operation before EC $v \in V_F \cup V_R \cup V_M$, its state is $S_o(v)$ and the bound FU and register are $F_o(v)$ and $R_o(v)$, respectively. The incremental scheduling and binding problem finds a state $S(v)$, a bound FU $F(v)$, and a bound register $R(v)$ for each added operation $v \in V_N$. The states of added operations and modified operations are called as *patched states* and will be stored in a patch memory. The objective of the incremental scheduling and binding algorithm is to find the scheduling such that the number of the patched states is minimized.

```
procedure SCHEDULE-AND-BIND(G_C, G_D, D, S, B, R)
    // G_C = (V_C, E_C) is the input control flow graph
    // G_D = (V_D, E_D) is the input data flow graph
    // D = (G, P) is the input datapath
    // S[n] is the schedule which maps each operation node n to a state
    // B[n] is the FU bind which maps each operation node n to an FU
    // R[n] is the register bind which maps each operation node n to
    // a register
 1: for all removed operation n ∈ V_R do
 2:     UNSCHEDULE-AND-UNBIND(S, B, R, n)
 3: for all modified operation n ∈ V_M do
 4:     MODIFY(S, B, R, n)
 5: for all basic block B ∈ V_C do
 6:     SMS-SORT(B)
 7:     for all operation node n ∈ B, taken in sorted order do
 8:         V ← AVAILABLE-SLOTS(n)
 9:         d ← SCAN-DIRECTION(n)
10:         for all state s ∈ V, taken in order of d do
11:             S[n] ← s
12:             BIND(D, n, S[n])
13:             if B[n] ≠ nil then break
14:         if B[n] = nil then
15:             S[n] ← NEW-STATE(n, d)
16:             BIND(D, n, S[n])
17: GENERATE-PATCH-DATA(S, B)
```

Figure 9: Incremental scheduling and binding algorithm.

4.3 Incremental Scheduling and Binding

Our algorithm performs the scheduling, FU and register binding *concurrently*. For each operation node $n \in V_D$, the scheduler finds the state in which n is executed, the FU binder finds the FU which executes the operation of n, and the register binder finds the register which stores the result of n. Although the present algorithm does not deal with operation chaining for ease of explanation, operation chaining can be performed in a straightforward manner. Moreover, our implementation of the patch compiler can perform operation chaining. Since the patch memory and the registers in the register file are limited resources, it is preferable to find the schedule and bind which minimize the usage of the resources. To achieve this goal, the proposed scheduling algorithm shown in Figure 9 is based on the Swing Modulo Scheduling [8]. The swing modulo finds the schedule such that the critical path is prioritized in the first place and the variable lifetime is minimized in the second. Therefore, the numbers of patched states and used registers in the register file can be minimized. Note that the present algorithm *does not* perform a modulo scheduling, *i.e.* software pipelining. However, it can be easily extended to perform a modulo scheduling in a straightforward manner. First, the removed operations are all unscheduled and unbound (Line 1-2) so that the scheduling slots become available. Also, the modified operations are scheduled and bound to newly-created states in the patch memory (Line 3-4). For each basic block B, SMS-SORT() determines the scheduling order of operation nodes B using the swing modulo scheduling algorithm [8] (Line 5-6). For each operation node n in the sorted order, AVAILABLE-SLOTS() finds a set of states S in which n can be scheduled (Line 7-8). SCAN-DIRECTION() determines the direction how the states in S are scanned. For each state in the direction, the binding is performed (Line 10-13). If no binding is found, a new state is created (NEW-STATE()) in the patch memory and binding is performed again (Lines 14-16). Finally, the patch memory data is generated (GENERATE-PATCH-DATA()). For each state, a control word is generated according to the FU and register binding.

Figure 10 shows the FU and register binding algorithm. Given a scheduled operation node n, AVAILABLE-FUS() finds a set of FUs which can be bound to n. Then, SORT-FUS() sorts the FUs in ascend-

Table 1: Performance comparison of different IDCT accelerator implementations (FreePDK 45nm technology).

Implementation Type	Post-Layout Area		Operating Frequency		Energy Efficiency			
					Not Patched		Fully Patched	
	Actual $[\mu m^2]$	Normalized	Actual $[MHz]$	Normalized	Actual $[GOPS/W]$	Normalized	Actual $[GOPS/W]$	Normalized
Fixed Function	42648	1.00	435	1.00	128	1.00	––	––
Proposed (16-state patchable)	47017	1.10	431	0.99	125	0.98	122	0.95
Proposed (32-state patchable)	60876	1.43	427	0.98	124	0.97	110	0.86
Proposed (128-state patchable)	144025	3.38	407	0.94	118	0.93	45	0.35
Fully Programmable (128 states)	281701	6.61	380	0.87	24	0.19	24	0.19

```
procedure BIND(D, S, B, R, n, s)
    // n is the node to be bound
    // s is the scheduled state of n
 1: H ← AVAILABLE-FUS(G, B, n, s)
 2: SORT-FUS(H)
 3: for all functional unit f in H, taken in sorted order do
 4:   B[n] ← f
 5:   success ← true
 6:   for all input or output m of n do
 7:     if ALL-DEPENDENTS-SCHEDULED(S, B, m) = true then
 8:       success ← BIND-REGISTER(P, R, m, f)
 9:     if success = false then break // No register available.
10:   if success = true then return // Binding found.
11: B[n] ← nil // No binding found.
```

Figure 10: FU and register binding algorithm BIND().

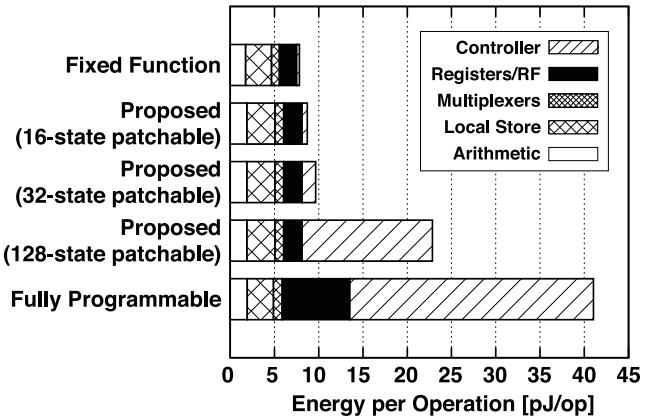


Figure 11: Energy breakdown of different IDCT accelerator implementations (FreePDK 45nm technology).

ing order of their binding costs. The binding cost of an operation node n to an FU f is the number of required registers in the register file when n is bound to f. For each FU f in sorted order, n is bound to f (Line 3-4). For each input or output m of n, we check if all the dependents are already scheduled. If so, a register is bound to store the value. If there is no individual register available, a register in the register file is bound. If some dependents are not scheduled yet, the register binding is inserted into a pending queue and it will be performed again once all the dependents are scheduled. If the register binding is successful for all inputs and outputs, the procedure returns to SCHEDULE-AND-BIND(). Otherwise, the binding of other FUs are performed.

5. EXPERIMENTAL RESULTS

5.1 Tool Implementation

We have implemented the proposed patch compiler in *Cyneum* synthesis and optimization framework which we have developed recently. Internally, LLVM compiler infrastructure [7] is used for analyzing an input C program and building a CDFG in SSA form. Given a pair of C programs before and after EC, a Δ-CDFG is constructed by analyzing the difference between two CDFGs. Also, a datapath organization, the scheduling and binding information corresponding to the original C program are given as inputs to the compiler. After patch compilation, the enhanced datapath and the patch memory data is generated in synthesizable Verilog HDL.

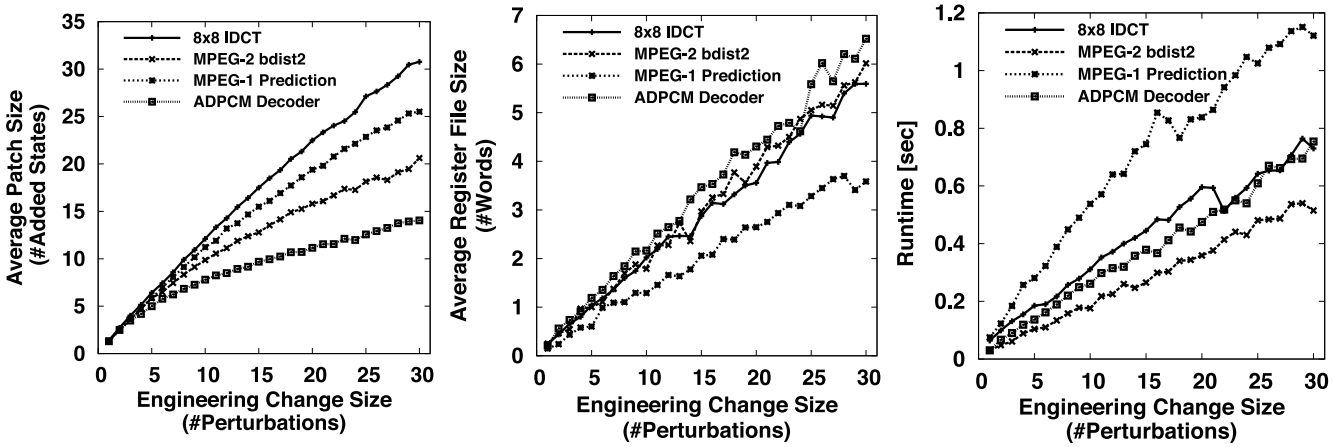
5.2 Energy Efficiency Comparison

In this section, we compare the energy efficiency of the proposed accelerators against fixed-function accelerators and programmable accelerators. As a benchmark design, a C description of 8x8 inverse discrete cosine transform (IDCT) is used. Then, we designed five types of accelerators: fixed-function, 16-state patchable, 32-state patchable, 128-state patchable, and fully programmable accelerators. Every accelerator implements 99 states and takes 727 cycles to complete the execution. Note that we present the 128-state patchable accelerator only as a reference. Since engineering change is assumed to be small ($10 \sim 20\%$) compared to a whole design, 16-state or 32-state patchable accelerator should be sufficient. A fixed-function accelerator shown in Figure 4 (a) is synthesized by a typical high-level synthesis algorithm. Then, a fully programmable accelerator shown in Figure 4 (b) is generated from the fixed-function accelerator by replacing the hardwired controller with a horizontal microcoded controller and the distributed registers with a centralized register file having multiple read and write ports. Using a standard cell library from Nangate implemented in the virtual 45nm technology FreePDK45 [1], the designs are synthesized using Synopsys Design Compiler Ultra with a high-effort option including gated clock optimization. All memory elements such as control memory, register/constant file and local store are implemented using flip-flops, *i.e.*, no SRAM is used in the designs. The mapped netlists are placed and routed using Cadence SoC Encounter. Then, we simulated one whole execution of 8x8 IDCT using Synopsys VCS. Using the simulated data in VCD format, the energy consumption is calculated using Synopsys PrimeTime PX.

Table 1 presents the comparisons of the five accelerators with respect to their post-layout area, operating frequency and energy efficiency. Figure 11 shows the energy breakdown of the five accelerators. As for the area, the programmable accelerator is about 7X larger than the fixed-function accelerator due to the control memory. The area of the 32-state patchable accelerator is about 43% larger than the fixed-function accelerator. As for the operating frequency, the programmable accelerator is much slower mainly due to the access time to the control memory and the register file. In contrast, the 32-state patchable accelerator is competitive to the fixed-function accelerator. As for the energy efficiency, the programmable accelerator is 5X less efficient than the fixed-function

An original DFG. Type 1: Changing the type of a node. Type 2: Changing the source of an edge. Type 3: Inserting a new node.

Figure 12: Design perturbation types.



(a) Patch size. (b) Register file size. (c) Runtime.

Figure 13: Patch compilation results of randomly-generated engineering changes of benchmark designs.

Table 2: Descriptions of benchmark applications.

Circuit	#Ops	Description
`idct`	286	8x8 inverse discrete cosine transform
`mpeg_pred`	369	MPEG-1 prediction
`bdist2`	182	MPEG-2 `bdist2` function
`adpcm_decoder`	178	ADPCM decoder

accelerator due to the programmable controller and the centralized register file. Without using the patching mechanism, the 32-state patchable accelerator shows only a few percentages of efficiency degradation. With fully using the patching mechanism, the degradation of energy efficiency is 14%. As can be seen from the energy breakdown, the register energy of the proposed accelerator does not increase significantly because the distributed registers are mostly used and the register file is not used frequently. These results clearly demonstrates the effectiveness of the proposed accelerators.

5.3 Patch Size & Runtime Evaluation

Next, we evaluated the proposed compilation method using the four benchmarks described in Table 2. Like [5], engineering change examples are obtained by iteratively applying a random graph perturbation to the original CDFG. Figure 12 shows the three types of graph perturbation. The first type of perturbation selects an operation node randomly and changes the node type randomly. The second type selects a data dependence edge randomly and change the source of the edge randomly. The third type inserts a new operation node of a random type at the randomly-selected place. For each iteration, one of the three types is randomly chosen and applied to the CDFG. If a node has no outgoing edge after applying graph perturbations, the node is removed from the CDFG. The degree of engineering change is estimated by the number of graph perturbations. Figure 13 (a) presents the average patch size with respect to the engineering change size. The increase rate is dependent to many aspects such as the original scheduling, binding, datapath structure, and control complexity. If the original scheduling is very tight, any engineering change may introduce a new state. Figure 13 (b) presents the average register file size. The graph shows that a large register file is not necessary for the engineering changes in this experiment. Finally, Figure 13 (c) presents the average patch compilation runtime. This demonstrates that the patch compilation method is applicable to practical designs.

6. CONCLUSIONS

This paper first proposed a novel energy-efficient patchable accelerator which enables post-silicon engineering change. The proposed accelerator can achieve high energy efficiency by implementing the controllers mostly by hardwired logic and providing a control patching mechanism. Then, we proposed a patch compilation method from a given set of an original design and a modified design. The experimental results demonstrated that the proposed accelerators offer high energy efficiency competitive to fixed-function accelerators and can achieve about 5X higher efficiency than the existing programmable accelerators.

7. ACKNOWLEDGMENTS

This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Cadence Design Systems, Inc. This work is also supported by the Japanese Ministry of Education, Culture, Sports, Science and Technology (MEXT), Grant-in-Aid for Young Scientists (B) 22760245.

8. REFERENCES

[1] *FreePDK45 Contents Page,* `http://www.eda.ncsu.edu/wiki/FreePDK45:Contents`. North Carolina State University, 2010.

[2] W. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *IEEE Computer*, 41(7):27 –32, 2008.

[3] FAAD2. `http://www.audiocoding.com/faad2.html`.

[4] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proc. IEEE Int. Symp. on High-Performance Computer Architecture (HPCA)*, pages 313–322, Feb. 2009.

[5] K. Fan, H. Park, M. Kudlur, and S. Mahlke. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *Proc. IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, pages 124–133, Apr. 2008.

[6] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proc. ACM/IEEE Int. Symp. on Computer Architecture (ISCA)*, pages 37–47, June 2010.

[7] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, page 75, May 2004.

[8] J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *Proc. IEEE Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 80–87, Oct. 1996.

[9] M. Reshadi and D. Gajski. A cycle-accurate compilation algorithm for custom pipelined datapaths. In *Proc. IEEE/ACM Int. Symp. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 21–16, Sept. 2005.

[10] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *HP Labs Technical Report HPL-2001-249*, Oct. 2000.

[11] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors 2009 Update*. 2009.

[12] Silicon Hive. `http://www.siliconhive.com/`.

[13] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. In *Proc. ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, Mar. 2010.

[14] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Anysp: Anytime anywhere anyway signal processing. In *Proc. ACM/IEEE Int. Symp. on Computer Architecture (ISCA)*, pages 128–139, June 2009.