A Tightly Coupled General Purpose Reconfigurable Accelerator LAPP and Its Power States for HotSpot-Based Energy Reduction

Jun YAO^{†a)}, Yasuhiko NAKASHIMA[†], *Members*, Naveen DEVISETTI^{†*}, *Nonmember*, Kazuhiro YOSHIMURA^{†**}, and Takashi NAKADA^{†***}, *Members*

SUMMARY General purpose many-core architecture (MCA) such as GPGPU has recently been used widely to continue the performance scaling when the continuous increase in the working frequency has approached the manufacturing limitation. However, both the general purpose MCA and its building block general purpose processor (GPP) lack a tuning capability to boost energy efficiency for individual applications, especially computation intensive applications. As an alternative to the above MCA platforms, we propose in this paper our LAPP (Linear Array Pipeline) architecture, which takes a special-purpose reconfigurable structure for an optimal MIPS/W. However, we also keep the backward binary compatibility, which is not featured in most special hardware. More specifically, we used a general purpose VLIW processor, interpreting a commercial VLIW ISA, as the baseline frontend part to provide the backward binary compatibility. We also extended the functional unit (FU) stage into an FU array to form the reconfigurable backend for efficient execution of program hotspots to exploit parallelism. The hardware modules in this general purpose reconfigurable architecture have been locally zoned into several groups to apply preferable low-power techniques according to the module hardware features. Our results show that under a comparable performance, the tightly coupled general/special purpose hardware, which is based on a 180nm cell library, can achieve 10.8 times the MIPS/W of MCA architecture of the same technology features. When a 65 technology node is assumed, a similar 9.4x MIPS/W can be achieved by using the LAPP without changing program binaries.

key words: reconfigurable architectures, multi-core processing, energy efficiency

1. Introduction

In recent years, much research has aimed at relieving the increasing problems of power consumption in modern processors. Figure 1 shows a comparison between recent implementations in terms of power/area efficiency based on research by Tobias, et al.[1]. More specifically, Application-Specific Integrated Circuits (ASICs) provide perfect power efficiency via their specialized support for individual programs, while their high power efficiency comes at a cost of a corresponding inflexibility and long design cycles, which impede their popularity considerably. On the contrary, the General-Purpose Processor (GPP) and its up-scaled version, Many-Core Architecture (MCA), can easily function for long and continuous service in existing programs since

its backward compatibility is well maintained via Instruction Set Architecture (ISA). However, with the general purpose (GP) target, MCAs lack an ability to tune for good energy efficiency as measured by MIPS/W. Due to the integration overhead, an MCA will have even less MIPS/W than its building block GPP. Furthermore, there is another constraint now growing in importance known as the utilization wall, by which, due to high power or heat dissipation, not all integrated transistors can be switched on simultaneously.

The general way in which special hardware, including ASIP, FPGA, and ASIC, adapt for a high MIPS/W execution is to tune hardware for the program hotspots, which are also known as program algorithm kernels. In the case of general programs, about 90% of the execution time will be consumed by only 10% of the program codes, usually in a loop fashion. These loop executions will thus represent the major behaviors of the program, including performance and power consumption. Recently, as an alternative to the above special architectures like ASICs, Coarse-Grained Reconfigurable Architecture (CGRA) has grown in popularity as a way to provide a medium performance/power value and flexibility by reconfiguring the execution in a Functional Unit (FU) array. This is done mainly by re-constructing FU interconnections to represent individual data paths [2]–[4]. However, most current CGRAs still require a special compiler to instruct good reconfiguration. This requirement leads to a compatibility issue similar to that of ASICs. To address these weak points, which are already too familiar, in both general and special hardware, we have designed a Linear Array Pipeline Processor (LAPP) [5]–[7]. This LAPP takes the structure of tightly coupled general/special parts to maintain the best features of the two modules. This hybrid combination provides both the backward compatibility to use existing compilers and high energy efficiency for individual applications.

This research extends the work in paper [8]. The main contributions here are as follows:

1. We have zoned a general purpose reconfigurable accelerator as an LAPP, according to the modular active behaviors in the accelerator. Power states are adaptively applied to each module according to their individual power features and coarse-grained utilization information. Minimal power consumption at each working stage, such as reconfiguring and hotspot acceleration, can be achieved by using this zoned design.

Manuscript received January 11, 2014.

Manuscript revised April 26, 2014.

[†]The authors are with Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.

^{*}Presently, with SHARP Corp.

^{**}Presently, with Fujitsu Labs.

^{***}Presently, with The University of Tokyo.

a) E-mail: yaojun@is.naist.jp

DOI: 10.1587/transinf.2014PAP0025

Fig. 1 Recent implementations, compared according to power and area efficiency in 130nm CMOS technology.

2. A detailed investigation of the power models and the corresponding power consumption, was carried out for each individual power state by combining the circuit and the architectural simulation approaches. The power consumption data indicates that the LAPP achieves 9.4x the MIPS/W of a many-core processor on a similar hardware scale.

The rest of this paper is organized as follows. Section 2 introduces the structure and the execution modes of LAPP. Section 3 introduces the corresponding preferable power states in each LAPP hardware zone. Based on a practical circuit estimation, Sect. 4 presents a detailed study of the power efficiency of LAPP by both the optimal acceleration and the aggressive application of low-power techniques. Section 5 concludes the paper.

2. LAPP: A Linear Pipeline Processor and Its General/Special Execution Models

2.1 Basic Structure of LAPP

Figure 2 shows the basic architecture of LAPP [5]–[7]. It contains a VLIW stage as the frontend for the purpose of backward compatibility. This frontend part makes up the general purpose (GP) section inside this architecture, which has exactly the same structure and functionality as a VLIW processor for taking binaries generated by conventional compilers like GCC. Though other GP architectures, like the scalar single-issue processor, the out-of-order pipelines, or the vector processor, can all serve as the frontend in this general/special purpose hardware, the reason that we specifically choose VLIW is its ability to achieve instruction level parallelism (ILP) within a relatively low hardware budget, as seen in Table 1. There will later be a tightly coupled special hardware attached to this frontend, which must have the same ISA specification as the frontend. Therefore, for efficiency in both the front and the back ends, a low ILP or a high-cost ILP is generally not preferred.

The special purpose (SP) hardware part is denoted as the array stages in Fig. 2. The vertically arranged array stages are added into the processor by extending the

Table 1 Frontend candidates for our general/special purpose hardware.

Frontend	Pros.	Cons.
Simple scalar	HW overhead: –	ILP: N/A IPC ≤ 1
Out-of-Order superscalar	dynamic ILP: + IPC $\approx \sqrt{\text{ways}}$	Complexity: scheduling & etc. HW: +++
Vector	static ILP: + IPC: highest when app vectorizable	Complexity: vector LD/ST HW: ++ (multi-port mem, regfile)
VLIW	static ILP: + IPC: medium	Complexity: medium (decoder) HW: + (multi-port regfile)

Fig. 2 Module structure of LAPP [7].

EXEC (Execution) and LSU (Load/Store unit) units in the VLIW stage, which can be used later to achieve a significant acceleration under the Array-Execution mode. In Fig. 2, three array stages have been depicted as an example, while this number can increase until the hardware limitation is reached. The basic idea for acceleration is to map each VLIW instruction in a loop kernel, cycle by cycle, onto the FUs in each array stage [7]. As a result, instructions of different loop iterations can exist simultaneously inside the array and the executions will be accelerated. As shown in Fig. 2, the FU array takes a two-dimensional form, in which each row can map multiple instructions of the same execution cycle, and the vertical direction is used for a full speed-up by exploiting parallelism between different loop iterations.

Figure 3 presents an example of how LAPP maps the hotspot, which is a loop kernel inside the special hardware FU array of LAPP. The loop kernel is first compiled into

```

// hotspot in C
for (i=N; i>0; i--)
  b[i] = a[i] + a[i-1] + a[i+1];

// hotspot in VLIW
// gr5, gr6 are the two base addresses of a, b
loop: ld @(gr5, #0), gr12; subc gr7, #1, gr7, icc; //VLIW 1
      ld @(gr5, #-4), gr13; beq icc, #1, exit; //VLIW 2
      ld @(gr5, #4), gr14; addi gr5, #4, gr5; //VLIW 3
           add gr12, gr13, gr12; //VLIW 4
           add gr12, gr14, gr12; //VLIW 5
      st gr12, @(gr6, #0); addi gr6, #4, gr6; bra loop; //VLIW 6

```

Fig. 3 Example of mapping a hotspot in LAPP.

a VLIW binary, in which each VLIW instruction can be sequentially executed inside the frontend. When the frontend detects this hotspot, its mapping logics, introduced in paper [7], will map the kernel into the FU array, as shown in Fig. 3. Note that the FUs in the frontend VLIW stage are also used to map the first loop instruction in this data-flow-graph (DFG) of this loop kernel, since the FUs share the same specifications. One special requirement is that the mapped loop kernel should not have data dependencies across loop iterations, so that the execution of each loop iteration can be largely overlapped. As in the example of Fig. 3, in the first cycle, VLIW 1 of the first iteration mapped on the VLIW frontend can be executed, while in the second cycle, VLIW 2 of the first iteration can start in array stage 1. However, in this second cycle, since the VLIW stage is empty now, it can thus start the execution of VLIW 1 of the second iteration. In this way, the throughput of this special hardware is loop-iteration completion for each cycle after the FU array is filled, thus resulting in a very high ILP under the array-based execution.

2.2 Execution Modes in LAPP

Overall, the proposed general/special hardware LAPP has three execution modes for both binary compatibility and execution acceleration. These modes are Normal-Execution, Array-Setup and Array-Execution [5]–[7], each of which function on different hardware modules inside the whole structure, as follows:

1. Normal Execution: LAPP uses its general purpose frontend to work on non-parallelizable parts of the VLIW program. The VLIW processor (denoted as the VLIW stage in Fig. 2), including the traditional IF, ID, RR, EXEC, MEM and WB pipeline stages, together with the register file (RF) and L1 cache (L1\$), is enabled and used cycle by cycle. The array stages, includ-

ing the mapping logic (mapper, designated as MAP), are not used in this mode.

2. Array Setup: When a parallelizable loop kernel DFG is detected, LAPP starts to prepare the mapping of the DFG into the FU array. In this mode, all FUs, including execution and memory access units are stopped, while the IF and ID stages inside the VLIW frontend still work to fetch and interpret the mapping information of the loop kernel DFG. The MAPs inside the ID stage and inside all array stages work to put the corresponding kernel into the FU array, following a hardware DFG mapping scheme as introduced in paper [7]. After the DFG mapping, an FSM-based control logic will start to prefetch the data for array execution into the L1\$, which can later guarantee a cache miss-free array execution. The IPC of this Array-Setup is zero, lasting a period of N cycles (N is the loop size) for mapping and M cycles (M is the data fetch size) for data preparation. For each innermost loop kernel DFG, this mapping in the Array-Setup is only required once, which thus comes to a trivial performance cost. The data fetching is always required even in a normal processor and does not really cause any additional performance degradation.
3. Array Execution: After the mapping is finished and data are ready inside the L1\$, LAPP can start a cache miss-free and high IPC array-based execution. This execution is independent of the IF, ID, RR of the VLIW frontend. Only the mapped FUs and L1\$ are required for this mode. Note that the register file (RF) provides the initial values—loaded in the first iteration—for the registers inside the FU array. After the first iteration, the physical registers are self-updated for instructions like `gr7++` and `ld @(gr5, 0), gr12`, because either `gr7` or `gr5` in these examples are continuously incremented inside the loop execution.

3. Power Reduction Model

With different hardware modules being used inside LAPP for each individual LAPP execution mode, the units in LAPP are actually locally zoned for better power use and, in addition, the optimal power control requires only simple control logics. More specifically, LAPP is friendly to power-gating based low-power technologies, and dynamic voltage scaling (DVS) is also able to find easy application in the locally zoned hardware modules of LAPP. For example, in traditional processors such as MCAs, the architectures do not allow gating of the FUs over most of the time, since the FUs are generally in use. In GPPs, only the instruction cache can be gated when a cache miss occurs, as shown in Fig. 4. Alternatively, the LAPP architecture has been designed to work effectively with unit gating over a sufficiently long period so as to conceal the gating penalty, which leads to an effective power saving. Low-power techniques such as Power Gating (PG), Clock Gating (CG) and Dynamic Volt-

Fig. 4 Gating of I1\$ in GPP.

age Scaling (DVS) are used to reduce power consumption in LAPP.

3.1 Power States of LAPP

Using the above methodologies in LAPP, we divided the power states of FUs into seven categories, as follows:

Power ON Active State: This is the normal working state when a unit is active in use. Here, the FU consumes dynamic power when calculating the different source values. We use the Synopsys HSPICE tool to model this power, assuming that for each cycle there is a 50% change in the input data.

Data Inactive State: The power supply is on and the clock is on, but the input data in the FU is unchanged. Since the data in the FU is unchanged due to pipeline stalls, the power consumed in this state is less than the power consumed in the active state.

Power Startup State: This is the power warming-up state when a previously PG-ed unit turns to the ON state. The power consumed in this state is the power required for the start-up of the unit.

Power ON Leak State: The power supply is ON, but the clock is OFF, and the FU is not in use. This represents the power state when CG is applied to the unit. In this state, power is consumed due to the leakage current.

Power OFF Leak State: In this state the power supply starts the transition from ON to OFF, at which point the power supply begins to drop gradually from full strength to near 0. A small amount of power will be consumed before the unit is fully black due to the PG.

Power Sleep State: The power supply voltage of the FUs is reduced using DVS.

Power OFF State: The power supply to the FU is completely OFF, and no power is consumed.

Table 2 summarizes these power states by listing the power ON/OFF and clock activity information in each state. We use the Synopsys HSPICE to give a detailed study of the power in each state, as is shown in Fig. 5. A 180nm technology was used to get the power data shown in this figure. More specifically, the power consumed in the *Power Active State* is taken as the base for the power consumed in other states. When PG is applied, the power state changes from active state to *Power OFF Leak State*. Here 0.2% power is consumed due to the leakage current before the unit fully transits to *Power OFF State*. When the unit restarts, the FU

Table 2 Power states.

State	Power	Clock	Data
Power ON Active	ON	ON	active
Data Inactive	ON	ON	unchanged
Power Startup	↑	OFF	N/A
Power ON Leak	ON	OFF	N/A
Power OFF Leak	↓	OFF	N/A
Power Sleep	standby	OFF	N/A
Power OFF	OFF	OFF	N/A

Fig. 5 Power in each state considering gating overhead (180nm).

is turned to *Power Startup State*, and 8% of the power is consumed due to startup overhead. Here, if the data is inactive, to reduce the power consumption, the clock is not turned on, and hence the FU goes to *Power ON leak state*. In this state, it consumes only 0.2% of the power of the active state, because the leakage current is on. When the data become active, the clock is also on, and the FU goes to *Power Active State* and performs the required operation. If the FU suddenly stops running due to stalls, the FU turns to the *Data Inactive state*, and consumes 75% of the power consumed in the active state.

3.2 Applying Low-Power Techniques to LAPP

Since the hardware modules are locally zoned into different utilization parts in LAPP under different execution modes, we describe here how the three technologies, PG, CG, and DVS, are used in each unit of the LAPP:

RF, I1\$: As introduced in Sect. 2, once the instruction mapping is completed and the processor turns to Array-Execution, RF and I1\$ are not accessed until the mode changes to Normal-Execution again. Then the unused RF and I1\$ are put into sleep mode using DVS, and power consumption is reduced, as is shown in Fig. 6 (a). The reason that PG is not applied for these two memory structured units is that PG damages the data and accordingly requires other high-cost methods to maintain the correct state by additional saving and re-loading.

IF, ID: As in the case of I1\$, IF and ID are not used in the Array-Execution. These units are turned off using PG, as shown in Fig. 6 (b).

Mappers (MAP in Fig. 2): Mappers are used to allocate instructions to the respective units. Therefore these units are switched on only in Array-Setup mode, when

(a) Power reduction model for RF, I1\$ (only DVS is applied)

(b) Power reduction model for IF, ID (Gating)

(c) Power reduction model for MAPs (Gating)

(d) Power reduction model for EXEC, Propagation registers, L0\$ (Gating)

Fig. 6 Power Reduction Model for each unit in LAPP.

the instructions need to be mapped. In other modes, Mappers are turned off using PG as shown in Fig. 6 (c).

L1\$, L0\$, Propagation Registers: The L1\$ is generally used in all the execution modes. But when a cache miss occurs, CG is applied to L1\$ until the required data arrives. L0\$ buffers and the propagation registers (in Fig. 2) are not used when the LAPP is in Array-Execution mode. Therefore, they are turned off by applying PG as shown in Fig. 6 (d).

EXEC: The EXEC units in the FU array are used for acceleration under the Array Execution. For a known DFG, only necessary FUs need to be active while the other parts can easily be put into power OFF state by PG, following the same scheme as in paper [9]. The prefetch time span overlaps and conceals the wakeup time span of the execution unit, as shown in Fig. 6 (d).

Figure 7 presents a summary of the above LAPP zones according to the above activity behaviors and the application of the low power techniques. The zone activity, power consumption data and corresponding power states are inte-

Fig. 7 Module zones of LAPP to match activities, using the Array-Execution phase as an example.

grated in our cycle-accurate simulator to achieve a practical study of the power efficiency of LAPP.

4. LAPP Power Efficiency Results

4.1 Circuit Size

LAPP is designed to have both general and special purpose sections, and it uses many FUs for data processing acceleration. Before making a comparison with GP MCAs, we provide a HW scale estimation of LAPP by implementing the hardware in HDL and synthesizing it with the Synopsys Design Compiler, based on a 180nm cell library, under a timing constraint of 100MHz. We estimate the power parameters of each unit by the Synopsys Prime Time PX. The circuit size and the power parameters are shown in Table 3. The second column there shows the circuit size in gates, and the third column gives the estimated power of each unit. Columns 4, 5, 6 show the number of units that exist in a traditional processor and in LAPP. The traditional processor contains a PC, an IF, an ID, an RF, an I1\$, an L1\$, an Effective Address Generator (EAG), three Arithmetic Logic Units (ALUs), four Media Units (MEDIAs) and a Branch execution unit (BRC). LAPP also has the same units in the VLIW stage. Other stages (i.e., excluding VLIW stage) in the LAPP contain only EXEC, MAP, SEL, an LSU and an L0\$ buffer. Finally, the circuit size of the power gating circuit (PGC) is calculated as 15% of the total area [9].

The ability of LAPP to map a DFG of an innermost loop kernel is defined by its vertical FU array depth. Our benchmark suite, which will be introduced in Sect. 4.2, requires a maximum 36-stage LAPP. Accordingly, we use 36 stages—a one-stage VLIW frontend and 35 array stages—as our LAPP hardware size. Table 4 shows the total area of a 180nm 36-stage LAPP, which is about 6.78×10^6 NAND2 gates. This HW scale is at the level of a 9-core MCA, whose building blocks are actually the VLIW frontend of the LAPP. This HW scale will be used in Sect. 4.2 to normalize the performance and the power consumption for both the LAPP and the MCA platforms.

Table 3 Area and power estimation results.

Unit	Circuit size [gates]	Power [mW]	Number of Units		
			NC	VS	AS
PC	685	1.58	1	1	0
IF	42,908	53.30	1	1	0
ID	26,247	22.20	1	1	0
RF	87,388	37.00	1	1	0
I1\$	205,853	93.40	1	1	0
L1\$	268,837	143.28	1	1	0
EXEC					
EAG	4,396	2.21	1	1	1
ALU	10,522	5.82	3	3	3
MEDIA	5,809	4.41	4	4	4
BRC	4,744	0.77	1	1	1
LSU	11,345	1.18	1	1	1
Mapper	23,961	26.30	0	1	1
SEL	24,896	25.56	0	0	1
L0\$	22,820	4.04	0	1	1
NC	707,204	388.84			
VS	753,985	varies			
AS	146,963	varies			
PGC of VS	113,098	varies			
PGC of AS	22,044	varies			

*NC: Normal CPU *VS: VLIW stage *AS: Array Stages

*PGC: Power Gating Circuit

Table 4 Area estimations for each architecture.

		gates
ManyCores	Traditional CPU $\times$ 9 cores	6,364,837
LAPP	36stages(Stage0, 35stages)	6,782,335

4.2 Evaluation

We use mainly image processing programs for our performance and power evaluation, as these applications are parallelizable across the loop iterations and thus fulfill the acceleration requirements for most special hardware, including LAPP. The image processing programs used for our evaluation are Frame Interpolation (FI), Median Filter (M), Edge Detection (E) and Stereo Matching (SM). We split the Frame Interpolation program into three small programs as FI-1, FI-2 and FI-3. FI-1 works on the Sum of Absolute Difference (SAD) calculation, FI-2 provides the calculation to find the smallest value of SAD, and FI-3 recreates the interpolated image. The total size of the image in the benchmark programs is 320×240 pixels, where the size of each pixel is 4 bytes. In LAPP, we consider the maximum data that can be used in one loop kernel to be 3×3 pixels, and the input data difference between two iterations to be 3 pixels at maximum. Table 5 summarizes the other parameter values used in the cycle-accurate simulator.

4.3 Performance and Power Evaluation

Figure 8 gives the performance results by comparing LAPP to a 9-core MCA which is at the same level of HW size. Note that the performance of this 9-core MCA is obtained by simply calculating 9 times the single-core GPP. Usu-

Table 5 Parameters of LAPP simulator.

VLIW stage	
Decode Width	max. 8 instructions/cycle
General Registers(GR)	32
Media Registers(MR)	32
Data transfer speed (with external cache)	8bytes/cycle
Instruction Cache	4 ways 16KB (64bytes/line)
L1 Data Cache	4 ways 16KB (64bytes/line)
L1\$ $\rightarrow$ L0\$ Data transfer rate	16bytes/cycle
Store Buffer	4 entries
Array stages	
Instruction Mapping Speed	2cycles/Stage
Inter L0\$ Data propagation rate	4 ways 256B (16bytes/cycle)
L1\$ $\rightarrow$ L0\$ Data transfer rate	16bytes/cycle
L0\$ $\rightarrow$ LSU Data transfer rate	4bytes/cycle
Store Buffer	1 entry

Fig. 8 Performance results: LAPP vs. a 9-core MCA.

ally such ideal up-scaling of the performance is not possible in multi-core architectures. However, as we are focusing mainly on image processing applications, which can easily be divided into subgraph processing, this 9x GPP performance may be attainable in the MCA.

The benchmarks are listed in an order of decreasing loop kernel size in Fig. 8. As was introduced in Sect. 2, LAPP accelerates the program kernel, which is the hotspot, i.e., the innermost loops, by mapping the loop body DFG into its FU array. The throughput of the LAPP FU array is per cycle loop-iteration completion when the DFG can be held inside the array. It can therefore be expected that a high IPC can be reached when processing large loop DFGs. Our performance data in Fig. 8 show this trend perfectly, since the largest loop-based FI-2 has the greatest performance speed-up—2.5x the result of MCA—while the performance in the smallest FI-3 loop of LAPP is lower than the MCA result. This is because the additionally introduced Array-Setup mode in LAPP will consume several cycles for loop body mapping. In addition, LAPP requires an array pipeline filling period, in which data processing flows from the first array stage to the final one. When the loop body is not large enough, the performance impact from these warming up periods becomes visible, as in benchmark FI-3. Overall, LAPP outperforms the 9-core MCA by 1.42x according to the benchmarks we studied. Considering that this speedup is achieved by assuming an ideal performance-scalable MCA, we believe that the special hardware backend in LAPP can

Fig. 9 Power results: LAPP vs. a 9-core MCA.

suffice as a candidate to exploit ILP and loop iteration parallelism in the parallelizable applications.

Similarly, the power consumption of both LAPP and the 9-core MCA have been simulated by our cycle-accurate simulator, which embedded, specifically, the practical power state and the corresponding value of each unit. Figure 9 shows those power consumption data. From this figure, it can clearly be observed that LAPP largely decreased the power consumption of the program hotspot execution, since the average power consumed in LAPP is 13% of the 9-core MCA result. Specifically, studying the individual applications, the power of MCA does not vary much when the benchmark changes. This is because all the hardware modules in an MCA, from the instruction fetch to the writeback, are required to be active for all the execution cycles. The only chance to apply a low-power execution mode is under long-penalty stalls such as L2 cache misses. On the other hand, LAPP uses special hardware for the program hotspots by first translating them into instruction mappings. Low-power techniques such as power gating (PG), as in paper [9], is easily used to achieve a maximum power reduction. After applying PG, for individual applications, the power consumption on LAPP is roughly proportional to the loop kernel sizes, as is indicated by the power data in Fig. 9.

Table 6 presents a more detailed energy consumption analysis of the individual hardware in the two architectures. The energy in this table is broken-down into the frontend, I1\$, L1\$, register file, and Execution parts. The data listed are normalized by the power data of these individual modules in the 9-core MCA. As was introduced above, we use the same VLIW design, whose breakdown power data are listed in Table 3, to construct the 9-core MCA. Note that each individual module in the 9-core MCA consumes a different amount of power, so that the sum of these breakdown percentage data do not reach 100%. This table shows that each unit in the LAPP consumes much less power than in an MCA.

Specifically, the energy consumed by the frontend and I1\$ in the LAPP is reduced considerably from the MCA results to 5.55% and 4.23% respectively. This is because LAPP uses the mapped loop DFGs during the acceleration, so that the program counter updates in the IF stage and the corresponding accesses to the I1\$ are therefore un-

Table 6 Ratio of energy consumption to Many-core [%].

Kernel (# of VLIWs)	frontend	I1\$	L1\$	Reg.	Exe.
FI-2 (33)	7.14	4.09	12.6	43.3	51.1
SM (31)	6.07	4.09	12.9	37.8	34.3
N (26)	14.4	5.03	11.6	33.4	44.6
M (23)	4.02	4.09	10.5	30.2	34.2
FI-1 (20)	3.11	3.98	10.2	16.9	25.2
E (14)	3.30	3.86	10.9	18.2	17.5
FI-3 (10)	2.75	4.23	10.2	15.7	16.9
Average	5.55	4.23	11.1	33.2	37.4

necessary. The approximately 5% energy consumption is mainly in the Normal-Execution and Array-Setup modes. As 90% of most program executions are in the hotspots, these two parts, IF and I\$, are put into sleep mode, mainly in our general/special-purposed LAPP, which thus results in a large energy reduction. The utilization of L1\$ follows a low-power technique similar to that of the I1\$. However, since data prefetching and data storing are still necessary in the Array-Setup mode to enable a data cache miss-free execution and to write back the generated data, the activity of L1\$ is slightly higher than that of I1\$.

The register files and the execution units consume more energy than the above parts. In this simulation, the register file also includes the physical registers inside the FU array, which are used for pipelining and data forwarding among the array stages. Also, the register files and the execution units are the most active units inside the LAPP, since they are also largely used in the Array-Execution mode to complete the calculations inside the loop kernels. It can be observed that the utilization of these two parts is proportional to the loop kernel sizes, denoted as the number of VLIW instructions inside the loop body. As an example, the energy consumption of these two parts reaches 50% in FI-2, and is only 15% in small loops like FI-3. Though the calculations in these hotspots should be the same using either MCAs or LAPP, an LAPP can identify the unused FUs by understanding the loop DFGs, after which it puts these FUs into power-gated mode in the Array-Execution. These unused FUs can be aggressively gated for a relatively long period in the acceleration mode, which is impossible in most general purpose platforms like MCAs. This can thus be regarded as the reason why even for large loop DFG such as FI-2, we can still achieve a 50% energy reduction. The overall energy reduction in LAPP is 14% of the MCA result, as is shown in Table 6.

4.4 MIPS/W in LAPP and MCA

In addition to the above 180nm data, we also tried to gather power data by using a 65nm library. The power data of the proposed power states under 65nm are given in Fig. 10. As 65nm has a larger leakage current, a power overhead higher than in a 180nm technology will occur under each individual low-power state and during the power state transitions.

The graph in Fig. 11 shows the performance per power consumption value (i.e. the power efficiency, measured as MIPS/W) of the LAPP, as compared to the 9-core MCA of

Fig. 10 Power states in 65nm and 180nm.

Fig. 11 Performance/Power Consumption (LAPP/MCA).

the same HW scale. The bars on the left are the power efficiency values of the LAPP in 180nm. Combining the data in both Figs. 8 and 9, we find that the MIPS/W of LAPP is around 6x to 14x that of the MCA MIPS/W, as is shown in Fig. 11. For individual applications, the relationship between the loop size and MIPS/W is not visible, which may be due to other factors, including the FU utilization rate and the L1\$ data reuse ratios in the DFGs. Overall, our results showed that the power efficiency of the LAPP based on 180nm process technology is 10.8 times that of an MCA of the same area. This value is lowered slightly to a 9.4x level when a 65nm technology is assumed. Note that these energy efficiency achievements are obtained by using the same VLIW binaries, and that, in addition, the performance of the MCA is assumed to take a perfect performance scaling factor. Therefore, our general/special-purposed LAPP can be regarded as having an optimized parallelism and power reduction.

5. Conclusion

In this paper, by tightly coupling a general purpose frontend and an FU array-based backend, we introduced an accelerator, LAPP, for the purpose of achieving high power efficiency without losing backward binary compatibility. LAPP speeds up conventional VLIW programs by effectively mapping instructions onto LAPP's FU array and executing a whole loop kernel in parallel. The high-parallel execution grants a large performance increase. In addition, this loop kernel mapping scheme can easily work with PG and other power reduction techniques. Using the power models introduced in this paper, we controlled, under a fine granularity, the application of low-power techniques in LAPP's

modules. Our results showed that the power efficiency of the LAPP based on 180nm process technology is 10.8 times that of an MCA of the same area.

We also investigated the effectiveness of an LAPP in 65nm process technology. Even though the 65nm technology's high leakage current makes many low-power techniques less effective than in the 180nm technology, the power efficiency of the LAPP is still 9.4 times that of a MCA of the same area. This indicates a prospective reason for using the LAPP even in future processing technologies.

Acknowledgments

This work is supported by VLSI Design and Education Center (VDEC), University of Tokyo with the collaboration of Synopsys Corporation, Cadence Design Systems, and Mentor Graphics. This work is supported by KAKENHI (No. 24240005, No. 24650020, and No. 2370060), and the STARC IS program.

References

- [1] T. Noll, T. Sydow, B. Neumann, J. Schleifer, T. Coenen, and G. Kappen, "Reconfigurable components for application-specific processor architectures," in *Dynamically Reconfigurable Systems*, pp.25–49, Springer Netherlands, 2010.
- [2] F. Bouwens, M. Berekovic, B.D. Sutter, and G. Gaydadjiev, "Architecture enhancements for the ADRES coarse-grained reconfigurable array," *HiPEAC'08*, pp.66–81, Jan. 2008.
- [3] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study," *DATE*, pp.1224–1229, 2004.
- [4] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder, "Scaling to the end of silicon with EDGE architectures," *Computer*, vol.37, no.7, pp.44–55, 2004.
- [5] T. Nakada, K. Yoshimura, S. Shitaoka, S. Ooue, D.V. Naveen, and Y. Nakashima, "A linear array accelerator for image processing," *IPSI Trans. Advanced Computing Systems*, vol.5, no.3, pp.74–85, May 2012 (in Japanese).
- [6] T. Iwakami, K. Yoshimura, T. Nakada, and Y. Nakashima, "Improvement of FU array accelerator by time division execution," *IPSI Trans. Advanced Computing Systems*, vol.5, no.4, pp.13–23, Aug. 2012 (in Japanese).
- [7] K. Yoshimura, T. Iwakami, T. Nakada, J. Yao, H. Shimada, and Y. Nakashima, "An instruction mapping scheme for FU array accelerator," *IEICE Trans. Inf. & Syst.*, vol.E94-D, no.2, pp.286–297, Feb. 2011.
- [8] N. Devisetti, T. Iwakami, K. Yoshimura, T. Nakada, J. Yao, and Y. Nakashima, "LAPP: A low power array accelerator with binary compatibility," 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, Los Alamitos, CA, USA, pp.854–862, 2011.
- [9] L. Zhao, D. Ikebuchi, Y. Saito, M. Kamata, N. Seki, Y. Kojima, H. Amano, S. Koyama, T. Hashida, Y. Umahashi, D. Masuda, K. Usami, K. Kimura, M. Namiki, S. Takeda, H. Nakamura, and M. Kondo, "Geyser-2: the second prototype CPU with fine-grained run-time power gating," *Proc. 16th Asia and South Pacific Design Automation Conference*, pp.87–88, 2011.

Jun Yao received the B.E., M.E. degrees from Tsinghua University in 2001 and 2004 respectively. He received the Ph.D. degree from Kyoto University in 2009. Since 2013, he has been an associate professor in Graduate School of Information Science, Nara Institute of Science and Technology. His current researches are in computer architecture field, especially low-power consumption and high-reliable architectures. He is a member of IEEE, ACM and IPSJ.

Yasuhiko Nakashima received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Kyoto University, Japan in 1986, 1988 and 1998 respectively. He was a computer architect in the Computer and System Architecture Department, FUJITSU Limited in 1988–1999. From 1999 to 2005, he was an associate professor in the Graduate School of Economics, Kyoto University. Since 2006, he has been a professor in the Graduate School of Information Science, Nara Institute of Science and Technology. His

research interests include processor architecture, emulation, CMOS circuit design, and evolutionary computation. He is a member of IEEE CS, ACM and IPSJ.

Naveen Devisetti received the B.E. degree from Osaka Electro-Communication University, Japan in 2010. He received the M.E. degrees from Nara Institute of Science and Technology, Japan in 2012. He is currently in SHARP Corp., since 2012. His research interests are in reconfigurable processors.

Kazuhiro Yoshimura received the B.E. degree from Ryukoku University, Japan in 2007. He received the M.E. and D.E. degrees from Nara Institute of Science and Technology, Japan in 2009 and 2012, respectively. He is currently in FUJITSU LABORATORIES LTD. since 2012. His research includes an energy efficient SMT processor with heterogeneous instruction architectures and a linear array based reconfigurable processor. He is a member of IEEE and IPSJ.

Takashi Nakada received the B.E., M.E., and Ph.D. degrees from Toyohashi University of Technology, Japan in 2002, 2004 and 2007 respectively. From 2007 to 2011, he was an assistant professor in Graduate School of Information Science, Nara Institute of Science and Technology. since 2012, he has been a project assistant professor in Graduate School of Information Science and Technology, the university of Tokyo. His research interest include processor architecture and related simulation technologies. He is a member of ACM and IPSJ.

He is a member of ACM and IPSJ.