Energy-Efficient Instruction Scheduling Exploiting Memory Access Slack

Akihiro Chiyonobu Toshinori Sato
Department of Artificial Intelligence, Kyushu Institute of Technology

Abstract

Current microprocessors require both high performance and low-power consumption. In order to reduce energy consumption with maintaining computing performance, we propose to utilize the information regarding instruction criticality. Microprocessors we are proposing have two types of functional units distinguished in terms of their execution latency and power consumption. Only critical instructions are executed on power-hungry functional units, and thus the total energy consumption can be reduced without severe performance loss. In order to achieve large energy reduction, it is required to execute instructions on power-efficient units as frequently as possible. In this paper, we propose a new instruction scheduling method utilizing cache miss information over the above mentioned scheduling technique. As a performance gap between microprocessors and main memories is increasing, it is possible that critical instructions are executed in power-efficient units as well as non-critical ones while main memory access is occurring. Our simulation results reveal that the modified instruction scheduling achieves 27.3% ED^2P reduction with 1.4% performance degradation.

1 Introduction

Traditionally, microprocessor performance is improved without considering the increase in its power and energy consumption. One of the ways to improve performance is to utilize a lot of functional units by speculative execution. If a branch miss-prediction occurs, useless energy is wasted. Large cache and large instruction window consume much energy. Today, power and temperature on a chip become a critical problem. It can be said that future microprocessors require both high performance and low-power consumption.

Most current microprocessors execute instructions in an out-of-order fashion in order to reduce the execution time of a program. The execution time is determined by the microprocessor's computing power and by dependences between instructions executed on the microprocessor. The critical path is the longest path in a data flow graph (DFG), where each node represents an instruction and each arc represents a dependence between instructions, and it determines the execution time of the program[9]. Figure 1 shows an example of a DFG. In this example, its critical path consists of instructions `I:0->I:3->I:6->I:7->I:8` when every instruction's latency is assumed to be one cycle. However, in an actual execution of the program, we concern that main memory access latency impacts critical path.

As well known, there is a tremendous performance gap between main memory and microprocessor. It is called the memory wall problem[17]. Figure 2 shows how the gap is increasing[7]. In this figure, vertical axis shows performance improvement on a logarithmic scale and horizontal axis shows years. As you can see, microprocessor performance is improved by 55% per year, but the improvement in memory performance is only 7% per year. If a load instruction is a hit in a cache, the requested data is delivered to the microprocessor immediately. If it is a miss, a memory access occurs and the microprocessor is stalled. We concern that the memory access latency strongly affects critical path. Because instructions dependent on a cache miss instruction cannot start until the requested data are delivered from the main memory.

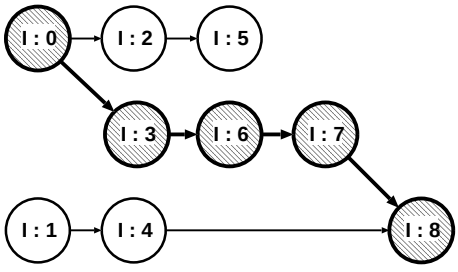

Figure 1. Critical Path

Microprocessors can continue to execute until their instruction window becomes full. In other words, there

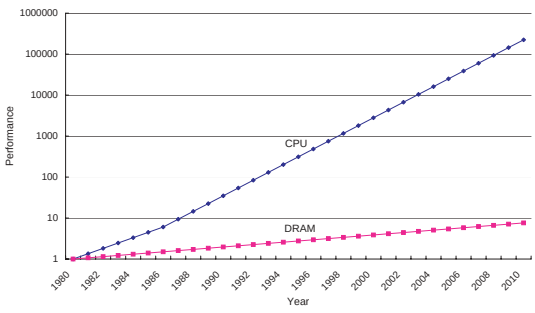

Figure 2. Performance Gap Between Memory and Microprocessor[7]

are a lot of in-flight instructions in the microprocessor, even when the memory access occurs. Instruction criticality depends on every load instruction. When it is a hit in the cache, the critical path statically identified is still dynamically critical. Otherwise, the weight of each arc in the DFG changes, and thus the other path becomes critical. Instructions dependent upon the missed load can not be executed. Only instructions independent of the missed load are executed during the memory access. As the speed gap between processors and memories is significantly large, the dependent instructions are on the critical path and the independent ones are not. An example is shown in Figure 3. In this figure, `I:0` and `I:1` are load instructions. Critical path consists of `I:0->I:3->I:6->I:7->I:8`, when both `I:0` and `I:1` are a hit in the data cache as we have seen in Figure 1. But if `I:1` is a miss the critical path turns into `I:1->I:4->I:8`.

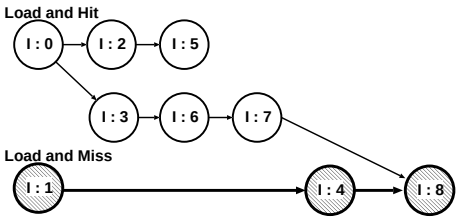

Figure 3. Exchange of the Critical Path

Considering these situations, in this paper, we propose a new instruction scheduling method utilizing cache miss information. The remainder of this paper is organized as follows: Section 2 explains how critical path information is created and proposes our instruction scheduling techniques. Section 3 describes the evaluation methodology, and Section 4 presents simulation results. Section 5 surveys related works. Lastly, Section 6 provides our conclusions.

2 Energy-Efficient Instruction Scheduling Methods

We proposed that a microprocessor has several functional units distinguished in terms of their execution latency and power consumption, and that instructions on the critical path, which determines the execution time of the program, are executed in fast and power-hungry units and instructions on the non-critical path are executed in slow and power-efficient units[3]. Using this scheduling strategy, we can reduce microprocessor energy consumption while maintaining its performance.

2.1 Path Information Table

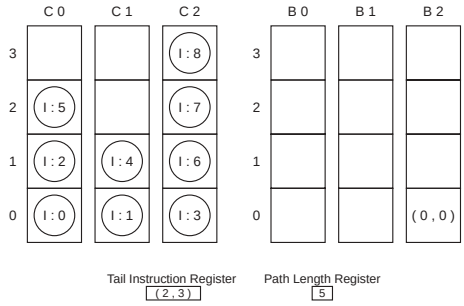

Figure 4. Path Information Table

In order to identify if each instruction is critical or not, we utilize path information table (PIT) proposed by Kobayashi at el.[9]. PIT is shown in Figure 4, where instructions in Figure 1 are registered. PIT consists of two tables, each of which has several FIFOs. Here, we call the first table chain table, and the other one branch table. As we can guess from their names, each FIFO in the chain table expresses a chain of dependent instructions. Each entry in the FIFO has a pointer to its corresponding instruction in the instruction window. The pointer is kept in the table until the instruction leaves the instruction window. When a branch is found in the DFG or when the current FIFO is full, the next FIFO is used to express the succeeding instruction chain. The branch table keeps a pointer which connects the branch with the root tree as shown in Figure 4. In this figure, (0, 0) in `B2` means that `I:3` in `C2` is a branch from `I:0` in `C0`. In `B0` and `B1`, there are no pointers. This means their corresponding instruction chain in `C0` and `C1` are independent of other instruction chains. Every entry in the branch table is also released when its corresponding

instruction is issued. There are two registers in PIT. One is the tail instruction register. It keeps which instruction in the chain tables is the tail instruction in the current critical path. The other is path length register. It keeps the length of the critical path. In other words, referring these registers, we can find when the tail instruction finishes. The contents in these registers are updated every time when new instructions are dispatched. From the explanations above, we can see that critical path is identified by referring PIT.

2.2 Dynamic Functional Units Gating

We propose an energy-efficient instruction scheduling technique, which we call dynamic functional units gating (DFUG). It exploits performance imbalance between microprocessor and main memory as explained in Section 1. Critical path information is utilized by our baseline scheduling technique[3]. Instructions on critical path are executed on fast and power-hungry functional units, while non-critical instructions are executed on slow and power-efficient ones. In this paper, we improve this scheduling strategy to be more power-efficient.

We are interested in main memory access, because it strongly affects instruction criticality. DFUG exploits its characteristics and switches over two modes. One is normal mode. In this mode, one from two different functional units is selected according to instruction criticality as explained above. The other is DFUG mode. In this mode only slow and power-efficient units are utilized. In other words, fast and power-hungry units are gated. Every L2 cache miss turns the processor into DFUG mode from normal mode. When the requested data are delivered from main memory, the mode returns into the normal mode from DFUG mode. The mode transition is shown in Figure 5. We have two switching policies. One is the baseline DFUG, which is already explained. Under this policy, every L2 cache miss and every cache refill change the instruction scheduling mode. The other is the enhanced DFUG (E-DFUG). Pipelined microprocessors that allow out-of-order execution need not stall on a cache miss. They can continue to fetch instructions from instruction cache while waiting for the data cache to return the missing data. A non-blocking cache[5] allows the data cache to continue to supply cache hits during a miss. This optimization reduces the miss penalty by being helped during a miss instead of ignoring the requests of microprocessor. Under such optimization, multiple L2 cache misses can be occur. In such case, DFUG mode continues until the last missing data is obtained.

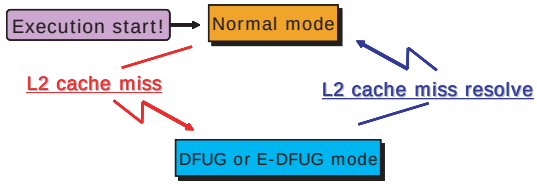

Figure 5. Mode Transition

3 Evaluation Methodology

This section describes our processor model and benchmark programs, which are used in simulations.

We use SimpleScalar Tool Set[1] for our base simulation environment. Since accurate memory hierarchy simulation is required, we use the SimpleScalar memory extension[2]. Each instruction scheduling method explained in Section 2 is implemented in detail. Table 1 shows the processor configuration. The fast functional units can execute most integer operations in one cycle, while the slow functional units execute operations in two cycles. In the rest of this paper, *functional units* means *integer units* (ALU). In this evaluation, we assume the followings. Our processor model has two fast and four slow ALUs, which is the best configuration when processor has six ALUs. It is revealed in our previous work[4]. Both the fast and slow ALUs can share their circuit design, while each transistor's size and threshold voltage might be optimized independently. According to the data-sheet of Intel Pentium M processor[8], we assume two configurations of the supply voltages for the fast and slow ALUs. Table 2 shows those combinations. Power consumption due to leakage current is out of consideration. It is remained for the future study.

Instruction set architecture (ISA) is the SimpleScalar/PISA ISA, which is an extension of MIPS R10000 ISA. We use seven programs from SPEC CPU2000 benchmark suite using *ref* input set. There are listed in Table 3. We fast forward two billion instructions and simulate 500 million instructions for each program. We do not count NOP instructions.

4 Evaluation Results

We use processor performance and energy-delay-square product (ED^2P) as metrics for our evaluation. For performance, higher bar means higher processor performance. For ED^2P, lower bar indicates higher power efficiency. We evaluate four models, **0f/6s**, **DFUG**, **E-DFUG**, and **PIT**. **0f/6s** means that all ALUs are slow.

Table 1. Processor Configuration

Fetch Bandwidth	8 instructions
Branch Predictor	1K-set 4-way set-associative BTB, 4K-entry 8-history-length gshare predictor, 64-entry return address stack, 6-cycle miss penalty, updated at commit stage
Insn. Windows	64-entry instruction queue, 32-entry load/store queue
Issue Width	8 instructions
Commit Width	8 instructions
Functional Units Latency (total/issue)	6 Int, 4 FP, 2 Ld,St fast iALU 1/1, slow iALU 2/1, iMUL 3/1, iDIV 20/19, fADD 2/1, fCMP 2/1, fCVT 2/1, fMUL 4/1, fDIV 12/12, fSQRT 24/24,
Insn. Cache Data Cache	64KB, 2-way, 64B blocks, 1-cycle latency 64KB, 2-way, 64B blocks, non-blocking load, hit under miss, 3-cycle latency
L2 Cache	unified, 1MB, 8-way, 64B blocks, 11-cycle latency
Bus Width	8B
Main Memory Latency	130-cycle latency

Table 2. The combinations of supply voltage and clock frequency[8]

Config A	Processor Clock	1.6GHz	800MHz
	Processor Core V_{dd}	1.484V	1.036V
Config B	Processor Clock	1.2GHz	600MHz
	Processor Core V_{dd}	1.276V	0.956V

Table 3. Benchmark programs

Benchmark	*input set*
gzip	input.source
vpr	net.in arch.in
gcc	166.i
parser	ref.in
vortex	lendian1.raw
bzip2	input.source
mcf	inp.in

4.1 Miss Rates of the Benchmark Programs

First, we evaluate cache miss rate of each benchmark program. We show the miss rate of each cache in Table 4. `mcf`, `bzip2`, and `parser` are programs whose cache miss rate is relatively higher than that of the others. We expect that there are many opportunities where DFUG and E-DFUG are applied. `gcc` has large DL1 cache miss rate, but its L2 cache miss rate is small. Therefore, it is expected that there are few opportunities where DFUG and E-DFUG are applied. For other programs which have small cache miss rate, it is expected that our techniques might be useless.

4.2 The Impact on Performance

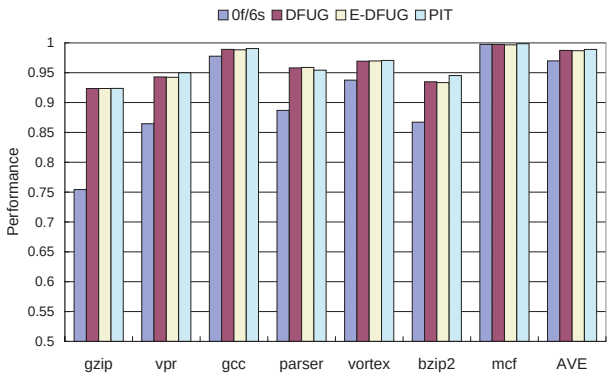

Figure 6. Performance Result

Second, we evaluate processor performance when DFUG and E-DFUG are used. This section investigates how DFUG and E-DFUG affect processor performance. Figure 6 shows processor performance. In this figure, vertical axis shows processor performance and horizontal axis shows each benchmark program and the average of results. Each performance is normalized by the result when all ALU in the processor are fast. Each program has different performance degradation. In the case of **0f/6s**, the performance is degraded by up to 24%, and on the average of 3%. When instructions are only scheduled by PIT, processor performance is degraded by 7.7% in the worst case and by 1.2% on average. In **DFUG**, the worst performance degradation is 7.7% and the average is 1.2%. E-DFUG degrades processor performance by 7.7% in the worst case and by 2.4% on the average. As you can see, performance degradation of **6s/0f** varies, and thus we can say that **0f/6s** is not a good choice for improving energy effi-

Table 4. Cache Miss Rate

	gzip	vpr	gcc	parser	vortex	bzip2	mcf	average
DL1 cache	3.5%	6.4%	17.1%	4%	1.8%	3.5%	40%	10.9%
L2 cache	3.4%	5.6%	8.2%	16.7%	1.7%	20.1%	52.3%	15.3%

ciency. In the case of `mcf`, processor performance is almost maintained. Table 4 shows that `mcf` originally has large cache miss rate. Hence, processor performance is dominated by the cache miss behavior. Also for other programs, performance degradations from PIT is very small. Therefore, it can be said that our scheduling techniques do not have any serve impact on performance.

4.3 Instruction Breakdown

Third, we investigate which ALU is used by each instruction. The breakdown is shown in Figures 7–9. In those figures, **cp_fast** indicates the percent of instructions that are identified as critical and are executed on fast ALU. **ncp_fast** indicates the percent of instructions that are identified as non-critical and are executed on fast ALU. **cp_slow** indicates the percent of instructions that are identified as critical and are executed on slow ALU. **ncp_slow** indicates the percent of instructions that are identified as non-critical and are executed on slow ALU. **DFUG_slow** indicates the percent of instructions that are executed on slow ALU while L2 cache miss occurs. In the case of DFUG, compared with PIT, the utilization of fast ALU is reduced by 3.7% on average. As expected in Section 4.1, our techniques are most effective for `mcf` and the utilization of fast ALU is reduced by 12.9%. In the case of `bzip2` and `parser`, the utilization of fast ALU is reduced by 4.4% and by 3.2%, respectively. As for **E-DFUG**, the utilization of fast ALU is reduced by 4.8% on average. The largest reduction in the utilization of fast ALU is found in `mcf` and it is 18.9%. In the case of `bzip2` and `parser`, it is decreased by 5.2% and by 3.4%, respectively.

As the utilization of fast ALU is reduced, our scheduling techniques are applied correctly to improve energy efficiency. As we have already seen, the reduction in the utilization of fast ALU does not diminish processor performance.

4.4 The Impact on Energy Efficiency

Lastly, we evaluate how our scheduling techniques affect ED^2P. Figure 10 shows ED^2P. In Figure 10, right part shows results when `Config A` in Table 2 is used, and the left part shows results when `Config B` in Table 2 is used. First, we focus on `Config A`. In the comparison with **PIT**, ED^2P is reduced by 1.5% and by 2% in the cases of **DFUG** and **E-DFUG**, respectively. In the case of `mcf`, ED^2P is reduced by 6.8%, and 10% in **DFUG** and **E-DFUG**, respectively. As we have seen, the reduction in the utilization of fast ALU is largest. On the contrary, in the case of `bzip2`, ED^2P is not reduced at all. In the case of `parser`, ED^2P is reduced by 2.7% and by 3% in **DFUG** and **E-DFUG**, respectively. As expected in Section 4.1, ED^2P is reduced in `mcf` and `parser`. However, it is not reduced in `bzip2`. This is because **DFUG** and **E-DFUG** affect the execution in the normal mode.

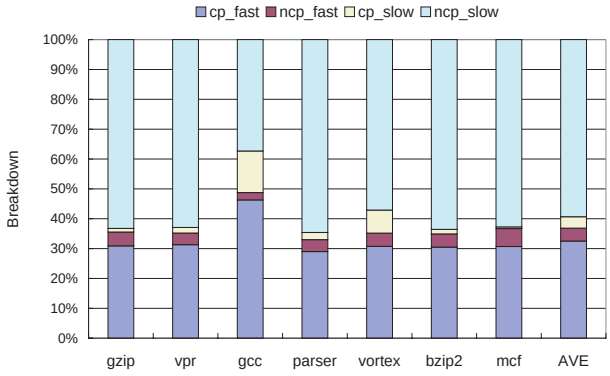

Figure 7. Instruction Breakdown of PIT

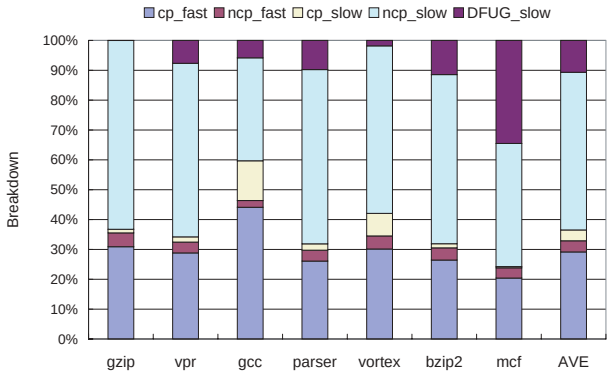

Figure 8. Instruction Breakdown of DFUG

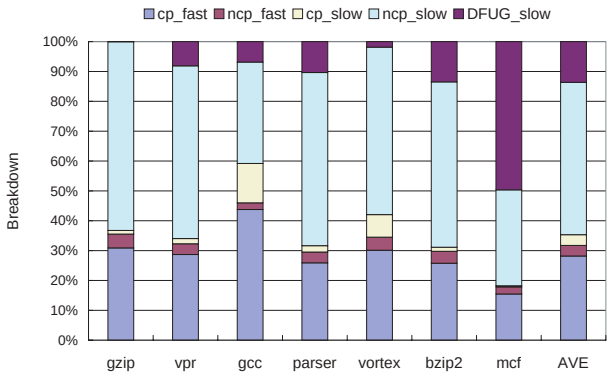

Figure 9. Instruction Breakdown of E-DFUG

When the scheduling mode returns to the normal mode from DFUG mode, a lot of instructions identified by PIT as critical are executed on slow units. This seriously diminishes performance as shown in Figure 6. This degrades ED^2P.

Next, we consider `Config B`. In this configuration, DFUG reduces ED^2P by 1.1% and E-DFUG improves it by 1.6%. In the case of `mcf`, ED^2P is decreased by 6% and by 8.6% in comparison with **PIT**. In the case of `bzip2`, ED^2P is increased by 0.5%. In the case of `parser`, ED^2P is reduced by 2.5% and 2.8% in **DFUG** and **E-DFUG**, respectively. We think that the reason of the increase in ED^2P is same with the one explained above.

For both `Config A` and `Config B`, we confirm that ED^2P is improved on average. We are currently investigating why ED^2P is degraded for some benchmarks.

5 Related Work

The critical path is a chain of dependent instructions, which determines the number of cycles executing the program. And thus, the performance of the processor is limited by the speed at which it executes the instructions along the critical path. If we can identify which instructions are critical, we can accelerate their execution by any means. Critical path prediction[3, 6, 16] is the technique for identifying critical instructions dynamically. Exploiting information regarding instruction criticality is effective not only for improving processor performance but also for reducing power consumption[14, 15]. Pyreddy et al.[14] use the profile-based heuristics proposed by Tune et al.[16] for identifying critical instructions. From a profile run, each instruction is marked as critical or non-critical. When the program is executed, the critical instructions are executed on fast and power-hungry functional units while the non-critical ones are executed on slow and power-efficient units. They concluded that dual pipeline had the potential for low power without suffering performance loss, but did not make any measurement of power savings. In contrast, Seng et al.[15] utilized a dynamic mechanism. They proposed to use the critical path predictor to identify non-critical instructions, and reported significant gains in the ratio of performance and power density. However, they only utilized critical path information and did not use L2 cache miss information for instruction scheduling.

Marculescu [13] proposed cache miss driven Dynamic Voltage Scaling (DVS) technique where the supply voltage is lowered when the processor detects L2 cache misses. Li et al. [11] proposed Variable Supply–Voltage scaling (VSV). VSV scales down the supply voltage and carries out independent computations at lower speed during an L2 cache miss. In VSV, the supply voltage is not scaled down when instruction level parallelism (ILP) is high. DVS method can be applied for Multiple Clock Domain microarchitecture [12]. It divides processor chip into individual clock domains exploiting globally asynchronous locally synchronous technique. Kondo et al. proposed dynamic processor throttling (DPT) for power efficient computations[10]. DPT dynamically detects the performance imbalance between the processor and main memory. They scale down the supply voltage and clock frequency to redress the imbalance. However, they did not mention critical path.

6 Conclusions

A performance gap between microprocessors and main memories is increasing. That gap often makes the microprocessor stalled. On the other hand, current microprocessors require both high performance and energy efficiency. Considering those, we are investigating to reduce energy consumption exploiting the performance imbalance between microprocessor and main memory. In this paper, we proposed the instruction scheduling techniques, which we call DFUG and E-DFUG, to reduce energy consumption with maintaining processor performance.

We evaluated the effect of our techniques using the SimpleScalar tool set and SPEC CPU2000 benchmark programs. Our evaluation results showed that ED^2P can be reduced by 27.3% (`Config A`) while performance is degraded by 1.4% on the average. From our evaluation results, it is concluded that our techniques can reduce energy consumption with maintaining computing performance.

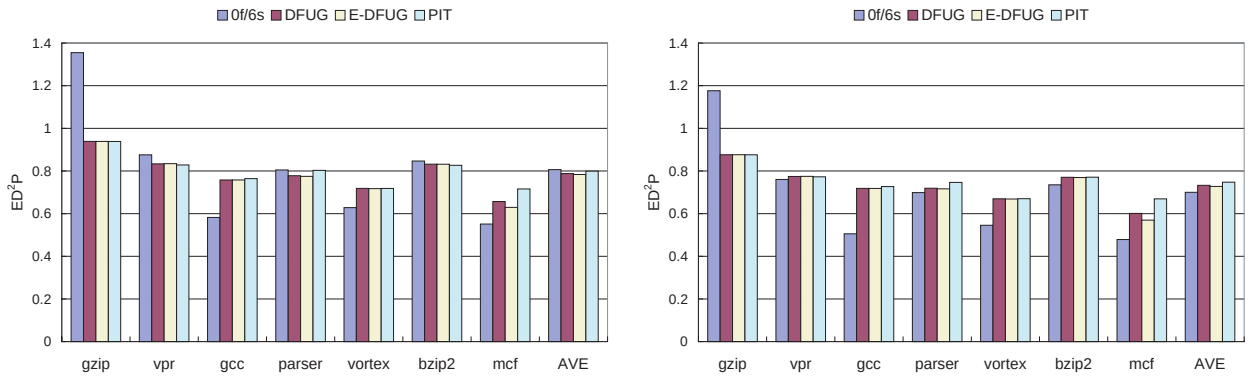

Figure 10. Energy–Efficiency Results

Acknowledgments

The authors thank Doug Burger for providing them with the SimpleScalar Memory Hierarchy Extensions kit. This work is supported in part by the grants from Japan Society for the Promotion of Science (No.1630019, No.176549).

References

[1] D. Burger, T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report CS-TR-97-1342, Computer Science Department, University of Wisconsin Madison, June 1997.

[2] D. Burger, A. Kagi, M. Hrishikesh, "Memory Hierarchy Extensions to the SimpleScalar Tool Set", Technical Report TR99-25, Department of Computer Science, University of Texas at Austin, April 1999.

[3] A.Chiyonobu, T.Sato, I.Arita, "Correlation-based Critical Path Predictors for Low Power Microprocessors", International Workshop on Innovative Architecture for Future Generation High–Performance Processors and Systems, January 2003.

[4] A. Chiyonobu, T. Sato, "Investigating Heterogeneous Combination of Functional Units for a Criticality–based Low–power Processor Architecture", 3rd International Symposium on Information and Communication Technologies, June 2004.

[5] K. I. Farkas and N. P. Jouppi, "Complexity/performance tradeoffs with non-blocking loads", 21st International Conference on Computer Architecture, April 1994.

[6] B. Fileds, S. Rubin, R. Blodik, "Focusing Processor Policies via Critical-Path Prediction", 28th International Symposium on Computer Architecture, July 2001

[7] J. L. Hennessy, D. A. Patterson,"Computer Architecture: A Quantitative Approach", Morgan Kaufmann, May 2002.

[8] Intel Corporation, "Intel Pentium M Processor Datasheet", April 2004.

[9] R.Kobayashi, H.Ando, T.Shimada, "Instruction-Issue Mechanism for a Clustered Superscalar Processor Focusing on a Critical Path in a Data Flow Graph", 13th Joint Symposium on Parallel Processing, June 2001 (in Japanese).

[10] M. Kondo and H. Nakamura, "Dynamic Processor Throttling for Power Efficient Computations", Workshop on Power-Aware Computer Systems, Dec 2004.

[11] H. Li, C-Y. Cher, T. N. Vijaykumar, K. Roy, "VSV: L2-Miss-Driven Variable Suppl-Voltage Scaling for Low Power", 36th International Symposium on Microarchitecture, December 2003.

[12] G. Magklis, M. L.Scott, G. Semeraro, D. Albonesi, S. Dropsho, "Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor", the 30th International Symposium on Computer Architecture, June 2003.

[13] D. Marculescu, "On the Use of Microarchitecture-Driven Dynamic Voltage Scaling", Workshop on Complexity–Effective Design, June 2000.

[14] R. Pyreddy, G. Tyson, "Evaluating Design Tradeoffs in Dual Pipelines", Workshop on Complexity-Effective Design, June 2001.

[15] J. S. Seng, E. S. Tune, D. M. Tullsen, "Reducing Power with Dynamic Critical Path Information", 34th International Symposium on Microarchitecture, December 2001.

[16] E. Tune, D. Liang, D. M. Tullsen, B. Calder, "Dynamic Prediction of Critical Path Instructions", 7th International Symposium on High Performance Computer Architecture, January 2001.

[17] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious", Computer Architecture News, March 1995.