Bypass Aware Instruction Scheduling for Register File Power Reduction

Sanghyun Park
Seoul National University, Korea
shparkid@compiler.snu.ac.kr

Aviral Shrivastava
University of California, Irvine
aviral@ics.uci.edu

Nikil Dutt
University of California, Irvine
dutt@ics.uci.edu

Alex Nicolau
University of California, Irvine
nicolau@ics.uci.edu

Yunheung Paek
Seoul National University, Korea
ypaek@ee.snu.ac.kr

Eugene Earlie
Strategic CAD Labs, Intel
eugene.earlie@intel.com

Abstract

Since register files suffer from some of the highest power densities within processors, designers have investigated several architectural strategies for register file power reduction, including "On Demand RF Read" where the register file is read only if the operand value is not available from the bypasses. However, we show in this paper that significant additional reductions in the register file power consumption can be obtained by scheduling instructions so that they transfer the operands via bypasses, rather than reading from the register file. Such instruction scheduling requires the compiler to be cognizant of the bypasses in the processor pipeline. In this paper, we develop several bypass aware instruction scheduling heuristics varying in time complexity, and study their effectiveness on the Intel XScale processor pipeline running MiBench benchmarks. Our experimental results show additional power consumption reductions of up to 26% and on average 12% over and above the register file power reduction achieved through existing techniques.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages—Code Generation, Compilers, Optimization, Retargetable Compilers

General Terms algorithms, measurement, performance, experimentation

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.

Keywords Architecture-sensitive Compiler, Bypass-sensitive, Forwarding Paths, Operation Table, Power Consumption, Processor Bypasses, Register File, Reservation Table

1. Introduction

Reducing the power consumption of register files is very important due to two main reasons. The first reason is that register files may consume a substantial portion of the power budget of modern microprocessors [15, 25], therefore reducing register file power consumption reduces the total processor power consumption. For example, in the Motorola M.CORE architecture, the register file consumes 16% of the total processor power and 42% of the data path power [6]. Azavedo et al. [2] observed that the register file power may reach 25% of the total processor power when running embedded applications. This amount would be even higher if the associated clock tree is taken into account.

The second and the more important reason is that due to the comparatively small size of the register file, the power density (power per unit area) of the register file is very high. In fact, register file is one of the most important hotspots in some of the commercial processors [4, 7]. As a result the register file is highly prone to "heat stroke" [9]. "Heat Stroke" occurs if the temperature of any part of the chip increases beyond a critical limit. Expensive packaging, heat sinks and other cooling solutions are required to avoid "heat stroke". However if a heat stroke occurs, the processor has to be stopped, and let cool, before it can resume execution. Since, typically the time it takes to cool is an order of magnitude more than the time taken to heat the component, avoiding heat stroke is of utmost importance. Again, since register file is one of the most important hotspots in the processors, reducing register file power is crucial to reduce the chances of a "heat stroke".

The situation is exacerbated by the trend of increasing register file power consumption. Modern microarchitectural techniques, e.g. register renaming, and compilation techniques, e.g. software pipelining, aim to improve the processor performance. Increase in performance often comes with more frequent usage of register file, and therefore increased register file power consumption. In addition, the trend of implementing large register files (e.g. in VLIW processors) is making the situation worse. Thus register file power reduction techniques are critically required not only to reduce the total processor power, but also to avoid possible heat strokes in processors.

Recognizing the need and importance of reducing register file power consumption, several techniques have been proposed. In particular, Tseng et al. [22] observed that existing processors perform an anticipatory register file and bypass read to obtain all the possible values of the source operand. The decision of which value will be finally used is made later. They found that on average 36% of the source operand values are transferred via bypasses in pipelined and bypassed processors. This implies that 36% of the values read from the register file are discarded and are never used, wasting substantial energy. They propose to reduce this register file power wastage by first deciding whether the value from the register file will be used or not. Register file read is performed only if the value from the register file will be used. In this paper we call architecture that employ this architectural technique as *On Demand RF Read* architectures.

Although, *On Demand RF Read* architectures are effective in reducing the register file power, there is scope for further reduction in the register file power consumption in such architectures by scheduling instructions so that the instructions read the operands from the bypasses, rather than from the register file. This requires the compiler to be cognizant of the bypasses in the processor. In this paper, we propose several bypass-aware instruction scheduling techniques aimed at reducing the register file power consumption.

The main results of this paper are -

- Our experiments on the Intel XScale processor pipeline with *On Demand RF Read*, executing benchmarks from the MiBench suite show that our proposed bypass-aware compilation techniques can further reduce the register file power consumption of an architecture already optimized architecture for register file power consumption by average 12% and up to 26%.
- Our proposed RFPN2 instruction scheduling is an effective heuristic to reduce register file power consumption. RFPN2 can reduce on average 10% register file power, with minimal performance loss (average 2%), and within reasonable compilation time.

The next section, Section 2, we do a brief survey of previous approaches for register file power reduction and show how our contributions are novel. Section 3 describes the experimental framework and empirically demonstrates the effectiveness of *On Demand RF Read* on the XScale processor pipeline. Section 4 describes the our bypass-aware compilation approach. Section 5 investigates the scope of register file power consumption by intra basic block instruction scheduling. Section 6 and Section 7 propose and evaluate two instruction scheduling heuristics with varying time complexities, aimed at reducing register file power consumption. Finally in Section 8, we summarize our work and present interesting future dimensions.

2. Related Work

As mentioned before, educing register file power consumption has two-fold importance: first, it reduces the power consumption of the whole processor, and second, it reduces the chances of "heat stroke", which can have a catastrophic impact. To start with, [25, 23] evaluate the register file power consumption and it's implications. Research on reducing the register file power consumption can broadly be classified into two categories: the first set of techniques aim at reducing the register file power by using less number of registers, while the second set of techniques aim to reduce the frequency of accesses to the register file.

2.1 Reduce the number of registers required

Reducing the number of registers required by an application enables the use of smaller register files, and thereby reduces the register file power consumption. Reducing the number of registers required by a program has mainly been a compiler forte. Several instruction scheduling techniques e.g. [24, 12, 5] have been proposed. Most of these techniques propose to schedule instructions so as to minimize the number of overlapping live ranges. In contrast to these compiler techniques our technique does not reduce the number of registers that a program uses, but reduces the frequency of register file accesses.

Among architectural techniques, [1] propose an interesting mechanism temporarily put the unused registers in a low-power state, reducing the register file power consumption. In addition, register renaming techniques e.g. [21] can modify the register requirements of an application independent of the compiler. In contrast of these architectural techniques, we propose a compiler technique that can be used in conjunction with these architectural techniques

2.2 Reduce number of accesses to register file

Apart from the obvious relationship between the frequency of accesses to the register file and its power consumption, reducing the number of accesses to the register file reduces the power consumption of the register file by enabling the use of register file with lesser number of ports. Owing to the quadratic dependence of the register file power on the number of ports, this is a very attractive option. In addition, reducing the frequency of accesses to the register file decreases

the performance penalty associated with reducing the number of ports in the register file. Combined, both of these reasons have lead to a wealth of architectural techniques to exploit this.

For example, [3] suggests the use of hierarchial register file organization to reduce the register file size, and a banked organization to reduce the port requirements. [11] observed that most register lifetimes are short. They propose to buffer the results between the functional units and the register file. [17] propose a technique of decoupled renaming to avoiding bank conflicts in a multi-bank register file. [16] adds small auxiliary memory structures to reducing the number of read and write ports. Tseng et al. [22] propose several register file and pipeline design modifications to reduce the register file power consumption. However, in this paper, we propose a compiler technique for register file power reduction.

Although compilers may influence the frequency of accesses to the register file by optimizations like dead code elimination, or register spilling etc., there has been no work in instruction scheduling targeted to reduce the frequency of accesses to register file. This is mostly because most existing processors read the register file anticipatorily. They read all the source operands from the register file, as well as the bypasses, regardless of whether or not they will be used. The decision of which value will be used is done later. In such a case, there is no scope of reducing the frequency of access to register file simply by instruction scheduling.

2.3 On Demand RF Read

Recently *On Demand RF Read* architecture was proposed by Tseng at al. in [22]. They propose to first compute whether the source operand value is present in the bypasses or not. Register file is then read if and only if the operand value from the register file will be used. They found that on a single issue MIPS-II architecture, running benchmarks from SpecInt95, 36% of operand values come from bypasses. Park et al. [17] also analyzed the this architectural feature and reported that on 8-issue SimpleScalar architecture, running benchmarks from SpecInt2K, 50-70% operand values come from bypasses.

Both [22], and [17] have evaluated and found this architectural feature effective in reducing the register file power consumption. Such an architecture provides an opportunity for the compiler to affect the number of access to the register file by reordering the instructions, and thus further reduce the register file power beyond what is achieved by the architectural technique alone.

2.4 Bypass Aware Compilation

Despite the possibility of reducing the number of accesses to the register file in *On Demand RF Read* architectures, no instruction scheduling techniques have been proposed till now. This is because, instruction scheduling to enable instructions to obtain their operands from the bypasses (instead of the register file), necessitates a bypass-aware compiler. Recently a bypass-sensitive instruction scheduling mechanism was presented by Shrivastava et al. in [20]. They used Operation Tables to implement bypass-sensitive scheduling in order to generate better performing code for partially bypassed processors. We employ the same Operation Table based approach to detect whether an operand is being transferred via bypass or not, and develop bypass-aware instruction scheduling technique for register file power reduction. Our instruction scheduling techniques can effectively perform instruction scheduling even for partially bypassed processors.

3. On Demand RF Read

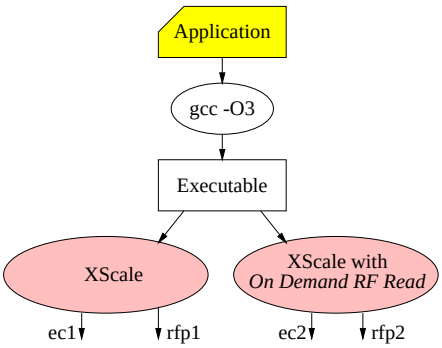


Figure 1. Experimental Framework to establish the effectiveness of *On Demand RF Read*

In this section, we describe our experimental setup, and empirically establish the effectiveness of *On Demand RF Read* architectural technique on the Intel XScale processor.

We perform our experiments on the Intel XScale architecture [14]. The design point is representative of processors targeted at relatively high-end, but low-cost, low-power embedded applications, including wireless and handheld devices. We simulate benchmarks from the MiBench suite [8], since these represent the intended target applications for the Intel XScale processor. The Intel XScale is a partially bypassed, 7-stage superpipeline, which is implemented in $0.18\ \mu$ technology, and operates at a maximum frequency of 1000 MHz.

As shown in Figure 1, we compile the applications using a GCC cross compiler for the Intel XScale, with all the performance optimizations turned on. The executable generated (in Figure 1) is then simulated on our cycle accurate simulator of the Intel XScale. The cycle accurate simulator structurally models the XScale pipeline in great detail, and has been validated against the 80200 evaluation board [13]. The simulator gives $ec1$, the number of execution cycles, and $rfa1$, the number of register file accesses, for the whole application.

The power model for the RF is generated using eCACTI cache power models [18], assuming a $0.18\ \mu$ technology.

We modified CACTI so that it can estimate the access time, energy, and area of small memory structures such as a multi-ported register files, which do not require the tags found in cache memories. CACTI provides us with ae, dynamic energy per RF access. The register file power consumption $rfp1$ is computed as $rfp1 = \frac{ae \times rfa1}{ec1}$.

As shown in Figure 1, we modify the base Intel XScale architecture to implement *On Demand RF Read*. We designed the logic to find out whether the operand is coming from the bypasses, and read register file only if it is required. We synthesized this logic using Synopsys Design Compiler-2001.10 [10] and 0.8 μ library *lsi_10k*, and linearly scaled the delay for 0.18 μ technology. We synthesized the *On Demand RF Read* logic for minimum delay. The delay of the *On Demand RF Read* logic was 0.8 ns, which is less than 1 ns (cycle time at 1000 MHz). Thus this logic can be comfortably implemented as an extra pipeline stage, just before the *Register Fetch (RF)* pipeline stage in the Intel XScale.

We also used Synopsys to estimate the power consumption of the *On Demand RF Read* logic. The power consumption of the logic is 756 μW, which is negligible ($< 1\%$) as compared to the register file power, so we do not consider it in our calculations.

We modified our XScale cycle accurate simulator (in Figure 1) with an extra pipeline stage before *Register Fetch (RF)*, and simulate the generated executables to obtain $ec2$, the number of execution cycles, and $rfr1$, the number of register file reads. The register file power, $rfp2$, is computed using the same formulas.

The *on Demand RF Read* architecture reduces the register file power consumption by 40%, with only 1% loss in performance. Thus, *On Demand RF Read* is an effective architectural feature to reduce the RF power. In the next sections we develop instruction scheduling algorithms for this architecture, and demonstrate their effectiveness on an already optimized architecture.

4. Bypass-Aware Compilation

Although *On Demand RF Read* is an effective architectural technique to reduce the register file power consumption in processors, in such architectures, there is scope for further reducing the register file power consumption by scheduling the instructions so that the value of their source operands are present in the bypasses. Since the register file is read only when the source operand value is not present in the bypasses, such scheduling can reduce the register file usage and therefore reduce register file power consumption. However, this requires the compiler to know exactly when an instruction bypasses the results, which source operands can read them, and when the result is written back in the register file. In architectures that have all the possible bypasses i.e., completely bypassed, two numbers $(l_1, l_2 \mid l_1, l_2 \in I,\ and\ 0 < l_1 < l_2)$ are required for each operation; where l_1 is the number of cycles after issuing the instruction, the result is computed, and l_2 is the number of cycles after issuing the instruction, the result is written in the register file. In a completely bypassed processor, the result of an instruction is available to every source operand in cycle l iff $l_1 \leq l \leq l_2$. When $l < l_1$, the result cannot be used, and when $l > l_2$, the result is available from the register file, until it is overwritten.

Although complete bypassing is superior for performance, it results in significant increase in the power consumption, area, and wiring complexity of the processor [19]. Owing to their stricter power, cost and complexity constraints, partial bypassing is more popular in embedded processors. In processors with partial bypassing, the analysis described in the previous paragraph becomes much more complex.

Shrivastava et al. [20] proposed the concept of Operation Table to perform bypass-sensitive instruction scheduling. An Operation Tables defines all the resources and registers that an operation uses in each cycle of its execution. It also defines which and when the operands are read, written and bypassed, to detect both the resource and data hazards in a given schedule of instructions.

To drive our instruction scheduling algorithms, we use a cost function, *OTGetInstrCost(S, f)*, which should give us the number of operands of instruction f that are read from bypasses, when f is scheduled as a next instruction in an already existing partial schedule S. A partial schedule here means an ordered list of instructions. We modify the operation *AddOperation* in [20] to do this. Another cost function that our algorithms need is *OTGetScheduleCost(S)*, which provides us the number of operands transferred by bypasses in the whole schedule. This is simply computed by repeatedly using *OTGetInstrCost(S, f)*, for each instruction of the schedule.

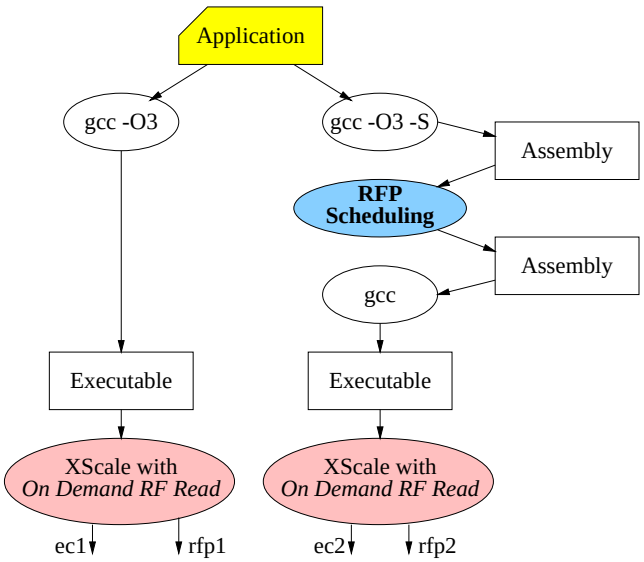
In the next sections, we will use these cost functions to drive our instruction scheduling techniques aimed at reducing register file power consumption.

5. Scope of RF Power Reduction

5.1 Experimental Framework

Now we perform experiments to estimate the scope of register file power reduction by intra-basic block instruction scheduling. As shown in Figure 2, first we compile the applications using GCC with all performance optimizations on. The executable generated is simulated on the Intel XScale Cycle Accurate Simulator with *On Demand RF Read*. The simulator gives the number of execution cycles, $ec1$, and the register file power, $rfp1$.

For our bypass-aware instruction scheduling, we first compile the benchmarks using the GCC cross compiler with the same high performance options to obtain the assembly code. The generated assembly is the input to our instruction scheduling algorithms. We read the assembly file, generate the instruction list, control flow graph, data flow graph and



Register File Power Reduction = (rfp1-rfp2)/rfp1
Performance Improvement = (ec1 - ec2)/ec1

Figure 2. Experimental Framework for instruction scheduling experiments

other compiler data structures. We then perform instruction scheduling at a basic block level.

We try all the legal permutations of instructions, allowed by the data dependencies. We use the function *OTGetScheduleCost(S)* to estimate the number of source operands that will be available from the bypasses. The schedule that maximizes the cost is chosen. such scheduling is performed for each basic block. The transformed assembly file is assembled and linked by the GCC compiler and executable is generated. The executable is simulated on the same Intel XScale Cycle Accurate Simulator with *On Demand RF Read*. The simulator gives the runtime, $ec2$, and register file power $rfp2$.

5.2 Instruction Scheduling

The functioning of our instruction scheduling algorithm is very different from traditional instruction scheduling algorithms which aim to improve performance. Tradtional performance oriented instruction scheduling algorithms try to separate dependent operations as much as possible, and insert non-dependent instructions in between them. This is done to ensure that even if the first instruction is blocked (due to data hazard or pipeline hazard), the dependent instruction does not have to stall. For example it is beneficial to separate the load instruction and the instruction that uses its result - in case there is a cache miss, the effective memory latency will be reduced by the "distance" between the two instructions. This technique is popularly known as "load hoisting".

In contrast, our instruction scheduling technique tries to schedule dependent instuctions close to each other so that they can transfer the dependent operand through the bypasses. As a result, if the first instruction is delayed (e.g. because of a cache miss), the memory latency will not be hidden and there will be a performance loss. However this performance loss should be small owing to the high hit rates of caches, and the fact that separating the dependent instructions hide only a very small fraction of the memory latency.

5.3 Effectiveness

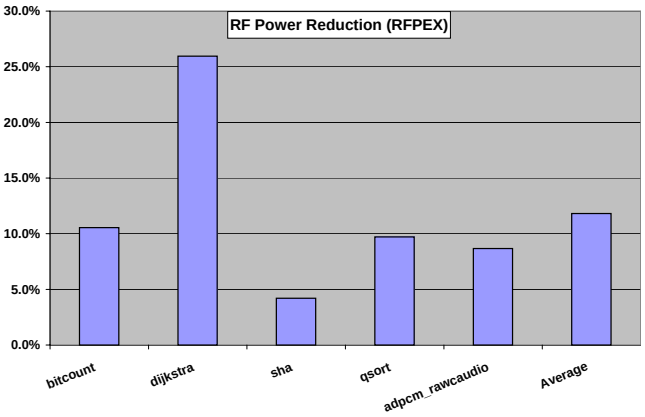


Figure 3. Reduction in RF Power Consumption

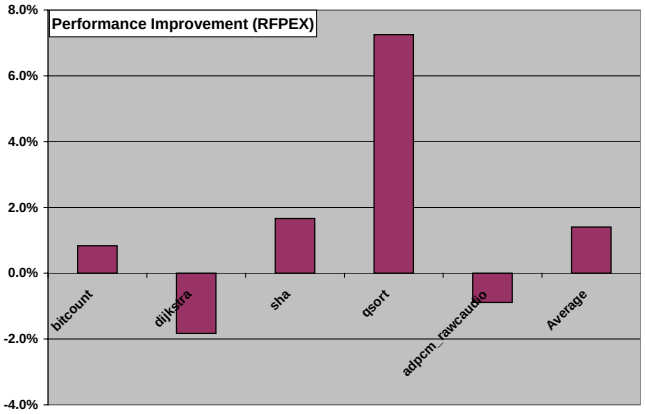


Figure 4. Performance Improvement

Figure 3 plots the percentage reduction in register file power The graphs show that there is scope of up to 26%, and average 12% register file power reduction by bypass-aware instruction scheduling techniques. It should be noted that these register file power reduction is in addition to the 40% reduction that *On Demand RF Read* already achieves, and are obtained without any additional architectural support. It is also worth mentioning here that even though 12% register file power reduction may not reduce the total processor power significanty, it is very important to reduce the register file power to reduce the chances of "heat stroke". Therefore even 12% register file power reduction is significant and important.

Figure 4 plots the percentage reduction in runtime, or performance improvements. As described previously in this section, our technique should only cause slight peformance degradations. The performance improvements that we see in some benchmarks is because our bypass-aware compiler, generates better schedules than GCC, which is unaware of the bypasses present in the Intel XScale [20]. The graph in Figure 4 shows an average 1.4% performance improvement.

Our scheduling does not consider inter-basic block effects. Considering those should decrease the register file usage at the basic block boundaries, and thereby further reduce the register file power consumption. However, there is definitely much scope of register file power reduction via instruction scheduling.

RFPEX is an exhaustive algorithm, and has exponential time complexity. It takes hours to compile for most benchmarks, and it could not schedule the benchmarks *susan* and *rijndael* in two days. As a result in the next two sections, we investigate simpler scheduling heuristics.

6. RFPN Scheduling

Heuristic RFPN(BasicBlock U)
01: $S = \phi$
02: **while** ($U \neq \phi$)
03: $F = getNextInstructions(S, U)$

/* get best next instruction */
04: $maxCost = 0$
05: $maxCostInstr = \phi$
06: **foreach** ($f \in F$)
07: $cost = OTGetInstrCost(S, f)$
08: **if** ($cost > maxCost$)
09: $maxCost = cost$
10: $maxCostInstr = f$
11: **endIf**
12: **endFor**

13: $S += maxCostInstr$, $U -= maxCostInstr$
14: **endWhile**
15: **return** S

Figure 5. Heuristic RFPN

Figure 5 shows our RFPN scheduling algorithm. This is a greedy algorithm to schedule instructions within a basic block. U is the ordered set of *unscheduled* instructions, while S is the ordered set of *scheduled* instructions. Initially U is full, while $S = \phi$ (line 01). In each iteration of *while-loop* (lines 02-14), one instruction is selected from U and moved to S (line 13), until U is empty and S contains all the instructions. Finally the ordered set of scheduled instructions S is returned (line 15). In each iteration of the *while-loop*, first the set of instructions, that are ready to be scheduled is computed using the data dependency information (line 03). We term the set of instructions that can be scheduled next as the *frontier set* F. For each instruction in the frontier set F, the function *OTGetInstrCost* is used to find out how many source operands will be transferred via bypasses if the instruction is scheduled next (line 07). The *for-loop*, (lines 04-12) does this.

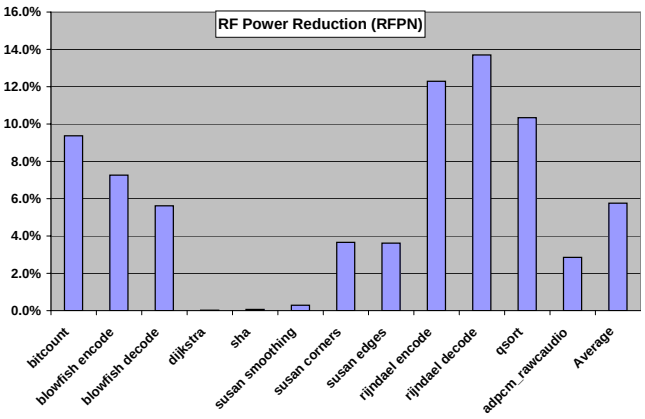


Figure 6. Reduction in RF Power Consumption

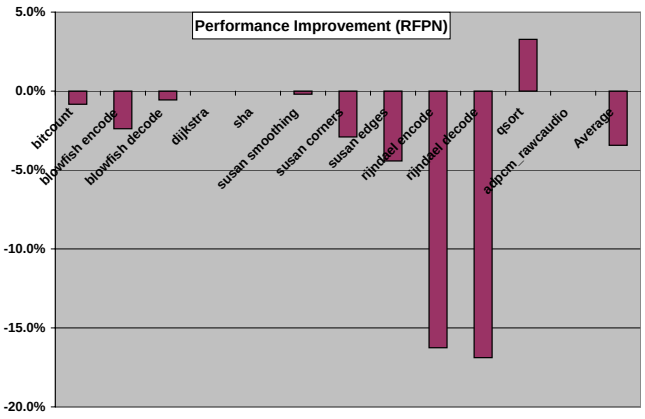


Figure 7. Performance Improvement

Figure 6 plots the percentage reduction in register file power, rfp, and Figure 7 plots the percentage improvement in performance, percentage increase in execution cycles (ec). The graphs show that there is an average 6% register file power reduction, at 3.5% performance loss. It takes only a few seconds to compile for each application. Next we investigate more complicated heuristics to achieve more reduction in the register file power consumption.

7. RFPN2 Scheduling

Next we propose and evaluate RFPN2 scheduling algorithm described in Figure 8. This algorithm takes as input U, the ordered set of unscheduled instructions, reorders them in S, the ordered set of *scheduled* instructions, and returns it. Initially U is full, while $S = \phi$ (line 01). In each iteration of the *while-loop* (lines 02-28), first the set of instructions that

Heuristic RFPN2(BasicBlock U)
01: $S = \phi$
02: **while** ($U \neq \phi$)
03: $F = getNextInstructions(S, U)$

/* Find the best next instr */
04: $maxCost = 0$
05: $maxCostInstr = \phi$
06: **foreach** ($f \in F$)
07: $S' = S + f, U' = U - f$

/* Assuming f, find the best schedule by RFPN */
08: **while** ($U' \neq \phi$)
09: $F' = getNextInstructions(S', U')$
10: $maxCost' = 0$
11: $maxCostInstr' = \phi$
12: **foreach** ($f' \in F'$)
13: $cost' = OTGetInstrCost(S', f')$
14: **if** ($cost' > maxCost'$)
15: $maxCost' = cost'$
16: $maxCostInstr' = f'$
17: **endIf**
18: **endFor**
19: $S' += maxCostInstr', U' -= maxCostInstr'$
20: **endWhile**

/* Assuming f, S' is the best schedule */
21: $cost = OTGetScheduleCost(S')$
22: **if** ($cost > maxCost$)
23: $maxCost = cost$
24: $maxCostInstr = f$
25: **endIf**
26: **endFor**

/* maxCostInstr is the best next Instr */
27: $S += maxCostInstr, U -= maxCostInstr$
28: **endWhile**
29: **return** S

Figure 8. Heuristic RFPN2

are ready to be scheduled, or (the *frontier* set), F, is computed using the data dependency information (line 03). Each instruction ($f \in F$) is chosen and a schedule is generated (lines 08-20), much like the greedy heuristic RFPN in Figure 5. All the schedules are then compared using OTs (line 21) to find out which schedule results in highest number of operands being transferred via bypasses (lines 21-26. The first instruction of the best schedule ($maxCostInstr$)is chosen, and added to S, and removed from U (line 27). Finally S, the ordered set of scheduled instructions is returned (line 29).

Figure 9 plots the percentage reduction in register file power, rfp, and Figure 10 plots the percentage improvement

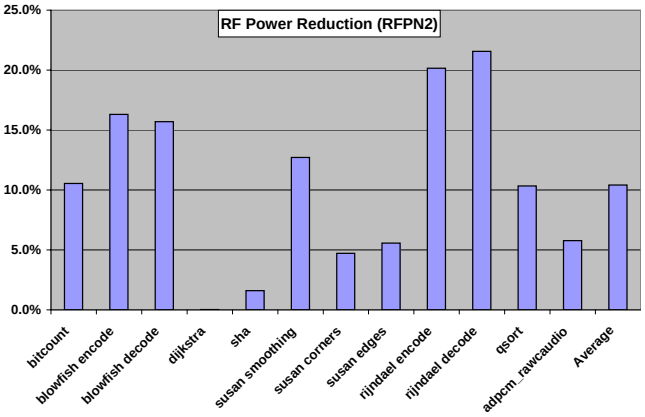


Figure 9. Reduction in RF Power Consumption

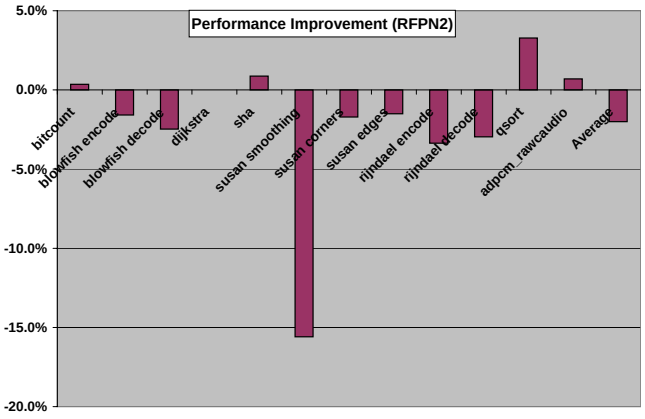


Figure 10. Performance Improvement

in performance, or percentage increase in execution cycles (ec). On an average, RFPN2 is able to reduce the register file power by 10.5%, at a minimal 2% loss of performance. It takes of the order of only a few minutes to compile for each application. We think that RFPN2 is a good "overall" heuristic.

8. Summary and Future Work

Register file power consumption has been widely recognized as a very important problem, not only to reduce the total power consumption of processor, but also to prevent "heat strokes". Consequently, register file power reduction has received a lot of research focus. *On Demand RF Read* is an architectural technique that reduces register file power consumption by reading registers only if the source operand value is not present in the bypasses. In this paper, we demonstrated that in architectures that employ this feature, there is scope for further register file power reduction by scheduling instructions so that they transfer operand values via bypasses, instead of reading them from the register file. This requires the compiler to be aware of the bypasses present/absent in the processor pipeline. As a result in this paper we proposed several bypass-aware instruction

scheduling techniques aimed at register file power reduction. Our experiments on the Intel XScale processor pipeline with *On Demand RF Read*, running MiBench benchmarks show that up to 26% and on average 12% register file power can be reduced. Further, one of our scheduling techniques RFPN2 is an effective heuristic to reduce the register file power consumption (10% on average) without much loss in performance (2% on average), and within reasonable compilation time.

Note that this reduction in the register file power adds up with that achieved by the hardware alone. Also even though 12% register file power reduction may translate into only a very small fraction of the total processor power consumption, the significance and importance of the register file power reduction lies in reducing the chances of "heat stroke".

Although even within basic block scheduling results in significant reduction in register file power consumption, we plan to investigate more sophisticated loop scheduling algorithms to fully explore the scope of register file power reduction by instruction scheduling. Other interesting dimensions of this work include reducing the number of read ports in the register file, and customizing processor bypasses to reduce both the bypasses as well as the register file power consumption.

9. Acknowledgements

This work was partially funded by Intel Corporation, UC Micro (03-028), SRC (Contract 2003-HJ-1111), and NSF (Grants CCR-0203813 and CCR-0205712), MIC (Ministry of Information and Communication),Korea, under the ITRC (Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031), MoST(M103BY010004- 05B2501-00411), IP/SOC, KRF contract D00191,Korea Ministry of Information and Communication under Grant A1100-0501-0004.

References

[1] J. L. Ayala, A. Veidenbaum, and M. López-Vallejo. Power-aware compilation for register file energy reduction. *Int. J. Parallel Program.*, 31(6):451–467, 2003.

[2] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints in the copper framework, 2002.

[3] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 237–248, Washington, DC, USA, 2001. IEEE Computer Society.

[4] J. Deeney. Thermal modeling and measurement of large high power silicon devices with asymmetric power distribution. In *International Symposium on Microelectronics*, 2002.

[5] A. Eichenberger and E. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proceedings of MICRO*, pages 338–349, 1995.

[6] D. R. Gonzales. Micro-RISC architecture for the wireless market. *IEEE Micro*, 19(4):30–37, 1999.

[7] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. The impact of increasing microprocessor power consumption. In *Intel Technology Journal*, 2001.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop in workload characterization*, 2001.

[9] J. Hasan, A. Jalote, T. Vijaykumar, and C. Brodley. Heat stroke: Power-density-based denial of service in smt. In *In Proceedings of International Symposium on High-Performance Computer Architecture*, 2005.

[10] http://www.synopsys.com/products/logic/design_compiler.html. *Synopsys Design Compiler*, 2001.

[11] Z. Hu and M. Martonosi. Reducing register file power consumption by exploiting value lifetime.

[12] R. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–267, 1993.

[13] Intel Corporation, http://www.intel.com/design/iio/manuals /273411.htm. *Intel 80200 Processor based on Intel XScale Microarchitecture.*

[14] Intel Corporation, http://www.intel.com/design/intelxscale/ 273473.htm. *Intel XScale(R) Core: Developer's Manual*.

[15] A. Kalambur and M. J. Irwin. An extended addressing mode for low power. In *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*, pages 208–213, New York, NY, USA, 1997. ACM Press.

[16] N. S. Kim and T. Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 172–182, New York, NY, USA, 2003. ACM Press.

[17] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 171–182, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[18] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. In *WRL Technical Report 2001/2*, 2001.

[19] A. Shrivastava, N. Dutt, A. Nicolau, and E. Earlie. Pbexplore: A framework for compiler-in-the-loop exploration of partial bypassing in embedded processors. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1264–1269, Washington, DC, USA, 2005. IEEE Computer Society.

[20] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 194–199, New York, NY, USA, 2004. ACM Press.

[21] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), 1967.

[22] J. H. Tseng and K. Asanovic. Energy-efficient register access. In *SBCCI '00: Proceedings of the 13th symposium on Integrated circuits and systems design*, page 377, Washington, DC, USA, 2000. IEEE Computer Society.

[23] L. Wehmeyer, M. K. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan. Analysis of the influence of register file size on energy consumption, code size, and execution time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1329–1337, 2001.

[24] H.-S. Yun and J. Kim. Power-aware modulo scheduling for high-performance vliw, 2001.

[25] V. Zyuban and P. Kogge. The energy complexity of register files. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 305–310, New York, NY, USA, 1998. ACM Press.