Register File Power Reduction Using Bypass Sensitive Compiler

Sanghyun Park, Aviral Shrivastava, Nikil Dutt, Alex Nicolau, Yunheung Paek, and Eugene Earlie

***Abstract*—This paper explores, develops, and investigates several bypass-sensitive compilation techniques to reduce the register file power by reducing the access frequency to the register file. We study the effectiveness of our techniques on the Intel XScale processor, which is based on the previously proposed "on-demand register fetch read" architectural feature. Furthermore, we show that our bypass-sensitive compilation technique is effective on various partial bypass configurations.**

***Index Terms*—Bypass sensitive, compiler, forwarding paths, operation table, power consumption, register file.**

I. Introduction

Reducing the power consumption of the register files is very important due to two main reasons. The first reason is that the register files may consume a substantial portion of the power budget of modern microprocessors [1]. Azevedo *et al.* [2] observed that the register file power may reach 25% of the total processor power when running embedded applications. The second, and the more important, reason is that the register file is one of the most important hotspots in some of the commercial processors [3], [4]. As a result, the register file is highly prone to "heat stroke" [5] which occurs if the temperature of any part of the chip increases beyond a critical limit. If a heat stroke occurs, the processor has to be stopped and cooled before it can resume execution. Considering that typically, the time it takes to cool is an order of magnitude more than the time it takes to heat the component, avoiding heat stroke is of utmost importance. Again, because the register file is one of the most important hotspots in the processors, reducing the register file power is crucial to reduce the chances of a "heat stroke." On one side, as the semiconductor technology keeps scaling down, the leakage power becomes a dominant factor in the total power. Considering that the leakage power increases exponentially with the increase of temperature, it is very important to reduce not only the dynamic power but also the leakage power of the register file.

Recognizing the need and importance of reducing the register file power consumption, several advanced register file designs and architectural techniques have been proposed [6]–[8]. In particular, Tseng and Asanovic [8] observed that existing processors perform an anticipatory register file and bypass read to obtain all the possible values of the source operand. They found that on average, 36% of the source operand values are transferred via bypasses in the pipelined and bypassed processors, implying 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 and then read from the register file only if it is necessary. In this paper, we call the architecture that employs this architectural technique as on-demand register fetch (RF) read architecture. Goel *et al.* [9] also used this architectural feature and proposed a compiler-driven technique to reduce the register file power consumption of the very long instruction word processor. However, they cannot support the partially bypassed processors.

Although the on-demand RF read architectures are effective in reducing the register file power by reducing the number of the register file access (rfa), there is a 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. Recently, a bypass-sensitive instruction scheduling mechanism was presented by Shrivastava *et al.* [10]. They used operation tables to implement the bypass-sensitive instruction scheduling mechanism; however, considering that the previous techniques did not consider the impact of bypassing to the register file power, their techniques were able to reduce the number of access to the register file by, on average, 0.9%, according to our preliminary experiments. In this paper, we propose several bypass-aware instruction scheduling techniques aimed at reducing the register file power consumption by reducing the number of access to the register file. Our proposed instruction scheduling can reduce, on average, 11.4% and up to 22% accesses to the register file over and above the on-demand RF read architectures, with a minimal performance loss (less than 1% on average) and within a reasonable compilation time. In addition, our bypass-sensitive compilation technique consistently achieves high degrees of the rfa reductions across various bypass configurations, demonstrating the usefulness of our technique on any partially bypassed processor.

II. Experimental Framework

We perform our experiments on the Intel XScale architecture. The design point is the representative of processors targeted at relatively high-end but low-cost and low-power embedded applications, including wireless and handheld devices. We simulate benchmarks from the MiBench suite because these represent the intended target applications for the Intel XScale processor. The Intel XScale is a partially bypassed seven-stage superpipeline, which is implemented in 0.18-μm technology and operates at a maximum frequency of 1000 MHz. We modeled the cycle-accurate simulator of the Intel XScale in great detail, and the simulator has been validated against the 80200 evaluation board.

We also modify the base Intel XScale architecture to implement the on-demand RF read. We designed the logic to find out whether the operand is coming from the bypasses and read the register file only if it is required. We synthesized this logic using Synopsys Design Compiler-2001.10 and 0.8-μm library lsi_10k and linearly scaled the delay for 0.18-μm technology. We synthesized the on-demand RF read logic for a minimum delay. The delay of the on-demand RF read logic was 0.8 ns, which is less than 1 ns (cycle time is at 1000 MHz). Thus, this logic can be comfortably implemented as an extra pipeline stage

Manuscript received July 28, 2007; revised November 19, 2007. This work is supported in part by a grant from Microsoft, by IDEC, by the Ministry of Information and Communication (MIC), Korea, under the Information Technology Research Center (ITRC) support program supervised by the Institute of Information Technology Assessment (IITA), which is the IT R&D program of MIC/IITA [2006-S-006-01, Components/Module Technology for Ubiquitous Terminals], under Grant IITA-2006-C1090-0603-0020, by the Human Resource Development Project for IT SoC Architect, and by the Nano IP/SoC promotion group of the Seoul R&BD Program in 2007. This paper was recommended by Associate Editor L. Benini.

S. Park and Y. Paek are with the School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-600, Korea (e-mail: shpark@optimizer.snu.ac.kr; ypaek@snu.ac.kr).

A. Shrivastava is with the Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: Aviral.Shrivastava@asu.edu).

N. Dutt and A. Nicolau are with the Department of Computer Science and Electrical Engineering, University of California, Irvine, CA 92697-3435 USA (e-mail: dutt@ics.uci.edu; nicolau@ics.uci.edu).

E. Earlie is with Intel Strategic CAD Laboratory, Hudson, MA 01947 USA (e-mail: Eugene.Earlie@Intel.com).

Digital Object Identifier 10.1109/TCAD.2008.923254

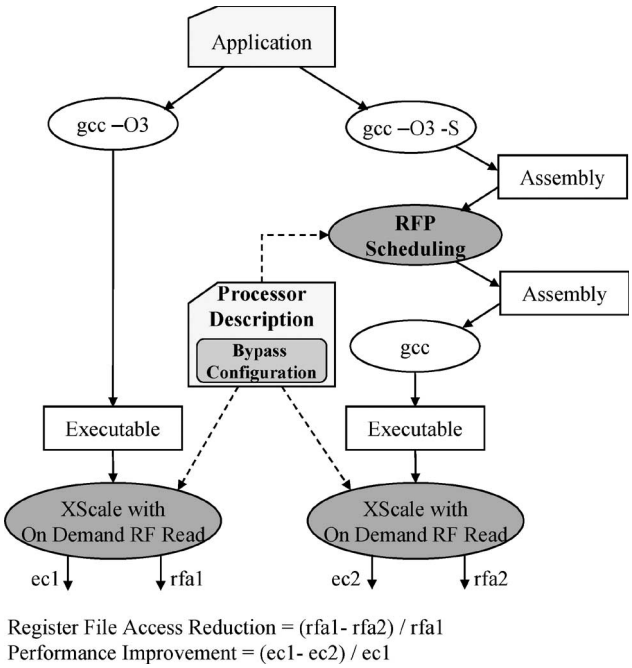

Fig. 1. Experimental framework for instruction scheduling experiments.

just before the RF pipeline stage in the Intel XScale. To estimate the power consumption of the on-demand RF read logic, we used Synopsys, and the power consumption of the logic is 756 μW, which is negligible ($< 1\%$) as compared with the register file power. Thus, we do not consider it in our calculations.

As shown in Fig. 1, we first compile the applications using GNU Compiler Collection (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 (ec), which is denoted as ec1, and the number of rfa, which is denoted as rfa1. These numbers are compared with ec2 and rfa2, which are the results of our bypass-sensitive scheduling techniques.

Our scheduling technique is applied after GCC generates the assembly code. We generate the operation tables for each assembly instruction, and then, we schedule the code with the help of the operation tables and the reconstructed data dependence graph. Considering that our scheduling techniques are intrabasic block and post-GCC algorithms, our techniques do not impact the GCC compilation stages. Although our bypass-sensitive techniques reschedule the assembly code and thus have influence on the performance, the overall performance degradation is very small because the precise modeling of the bypasses in operation tables ensures the source operands to be fetched from bypasses without delays.

III. Scheduling for RFA Reduction

In this section, we perform experiments to estimate the scope of rfa reduction by instruction scheduling. The functioning of our instruction scheduling algorithm is very different from the traditional instruction scheduling algorithms which aim to improve performance. Traditional performance-oriented instruction scheduling algorithms try to separate dependent operations as much as possible and insert nondependent instructions 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 instructions 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 performance loss. However, this performance loss should be small, owing to the high hit rates of caches and to the fact that separating the dependent instructions hide only a very small fraction of the memory latency.

A. Scope of RFA Reduction

We first developed an exhaustive scheduling algorithm RFPEX to see the scope of the rfa reduction. In this algorithm, we try all the legal permutations of instructions in each basic block and pick the best performing schedule. Each of the rightmost bars shown in Fig. 3 plots the normalized reduction in the number of access to the register file. The graphs show that there is a scope of up to 26% and, on average, 12% rfa reduction by bypass-aware instruction scheduling techniques. It should be noted that this rfa reduction is an addition to the 40% reduction that on-demand RF read already achieves and is obtained without any additional architectural support. It is also worth mentioning here that even though a 12% rfa reduction may not significantly reduce the total processor power, it is very important to reduce the frequency of access to the register file to reduce the chances of "heat stroke." Therefore, even a 12% reduction in the usage of the register file is significant and important.

The rightmost bars shown in Fig. 5 plot the normalized runtime. As described previously in this section, our technique should only cause slight performance degradations. The performance improvements that we see in some benchmarks happen because our bypass-aware compiler generates better schedules than GCC, which is unaware of the bypasses that are present in the Intel XScale [10]. The graph in Fig. 5 shows an average 1.8% performance improvement.

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

B. RFPN Scheduling

Fig. 2 shows our RFPN scheduling algorithm. This is a greedy algorithm utilized to schedule instructions within a basic block. U is the ordered set of unscheduled instructions, whereas S is the ordered set of scheduled instructions. Initially U is full, whereas $S = \phi$ (line 01). In each iteration of a "while loop" (lines 02–12), 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 13). In each iteration of the while loop, the set of instructions that are ready to be scheduled is computed first, using the data dependence 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 06). The "for loop" (lines 04–10) does this.

The leftmost bars shown in Fig. 3 plot the normalized number of rfa, and each of the leftmost bars shown in Fig. 5 plots the normalized ec. The graphs show that there is up to 13.7%, and an average 5.8%, rfa reduction, with less than 1% 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 number of rfa.

```
Heuristic RFPN(BasicBlock U)
01: S = φ
02: while (U ≠ φ)
03:    F = getNextInstructions(S, U)
 /* get best next instruction */
04:    maxCost = 0, maxCostInstr = φ
05:    foreach (f ∈ F)
06:       cost = OTGetInstrCost(S, f)
07:       if (cost > maxCost)
08:          maxCost = cost, maxCostInstr = f
09:       endIf
10:    endFor
11:    S += maxCostInstr, U −= maxCostInstr
12:  endWhile
13: return S
```

Fig. 2. Heuristic RFPN.

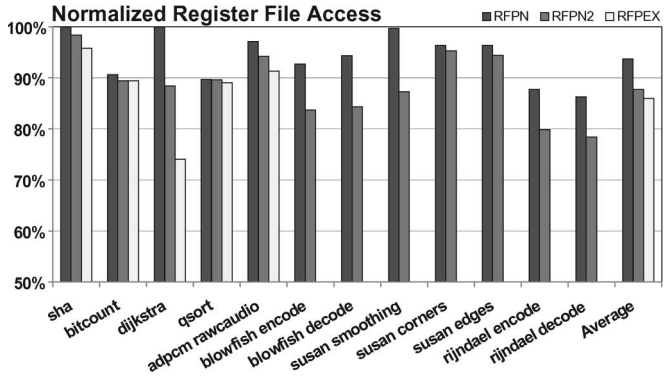

Fig. 3. Normalized rfa.

```
Heuristic RFPN2(BasicBlock U)
01: S = φ
02: while (U ≠ φ)
03:    F = getNextInstructions(S, U)
/* Find the best next instr */
04:    maxCost = 0, maxCostInstr = φ
05:    foreach (f ∈ F)
06:       S' = S + f, U' = U − f
/* Assuming f, find the best schedule by RFPN*/
07:       while (U' ≠ φ)
08:          F' = getNextInstructions(S', U')
09:          maxCost' = 0, maxCostInstr' = φ
10:          foreach (f' ∈ F')
11:             cost' = OTGetInstrCost(S', f')
12:             if (cost' > maxCost')
13:                maxCost' = cost', maxCostInstr' = f'
14:             endIf
15:          endFor
16:          S'+ = maxCostInstr', U'− = maxCostInstr'
17:       endWhile
/* Assuming f, S' is the best schedule */
18:       cost = OTGetScheduleCost(S')
19:       if (cost > maxCost)
20:          maxCost = cost, maxCostInstr = f
21:       endIf
22:    endFor
/* maxCostInstr is the best next Instr */
23:    S+ = maxCostInstr, U− = maxCostInstr
24: endWhile
25: return S
```

Fig. 4. Heuristic RFPN2.

C. RFPN2 Scheduling

Next, we propose and evaluate the RFPN2 scheduling algorithm shown in Fig. 4. This algorithm takes U as input, which is the ordered set of unscheduled instructions, reorders them in S, which is the ordered set of scheduled instructions, and returns it. Initially, U is full, whereas $S = \phi$ (line 01). In each iteration of the while loop (lines 02–24), the set of instructions that are ready to be scheduled, or the frontier set F, is first computed by using the data dependence information (line 03). Each instruction $(f \in F)$ is chosen, and a schedule is generated (lines 07–17), which is very similar to the greedy heuristic RFPN in Fig. 2. All the schedules are then compared by using operation tables (line 18) to find out which schedule results in the highest number of operands being transferred via bypasses (lines 18–21). The first instruction of the best schedule (maxCostInstr) is chosen, added to S, and removed from U (line 23). Finally, S, which is the ordered set of scheduled instructions, is returned (line 25).

Each bar shown in the middle of Fig. 3 plots the normalized rfa frequency, which is denoted as rfa, and the corresponding bars shown in Fig. 5 plot the normalized ec. On an average, RFPN2 is able to reduce the number of access to the register file by up to 22%, with an average of 11.4%, which is at a 1% minimal loss of performance. It takes the order of only a few minutes to compile for each application. We think that RFPN2 is a good "overall" heuristic. It is worth mentioning in this paper that our heuristics have no impact on the energy, considering that they have minimal overhead on the performance.

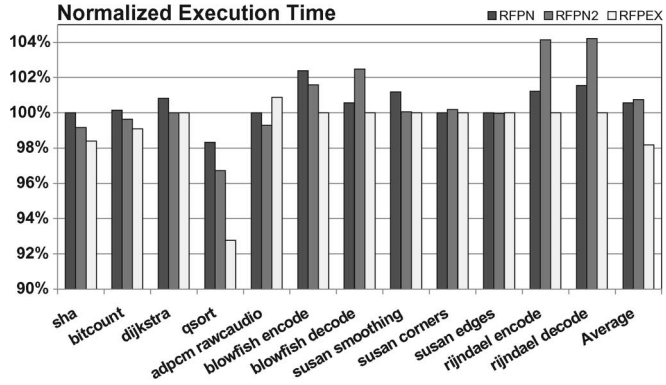

Fig. 5. Normalized execution time.

IV. RF Power Estimation

Due to the continuous technology scaling, the contribution of leakage power in the register file power is increasing. The leakage in the register file is high, not only because of the nanoscale gate dimension, but also because of its high operating temperature. As mentioned before, if the register file heats up past the critical temperature limit, a thermal emergency called heat stroke occurs, and the processor has to be stopped to let it cool. This concept, which is very similar to the stop-and-go policy proposed in [11], incurs significant performance degradation to ensure a safe operating temperature. Considering that the temperature and thus the leakage power become increasingly important portions of the total power of a chip, in this section, we model both the dynamic and leakage powers of the register file and demonstrate the effectiveness of our technique.

We model the register file in HotSpot as a single block. In addition, we model the exponential dependence of leakage power on the temperature using PTScalar.

Fig. 6 shows the normalized power consumption of the register file when we use the RFPN, RFPN2, and RFPEX algorithms. This graph shows that our heuristic RFPN2 can achieve, on average, a 10% reduction each in the register file power. Note that the power reduction in Fig. 6 is in addition to the reduction that the on-demand RF read

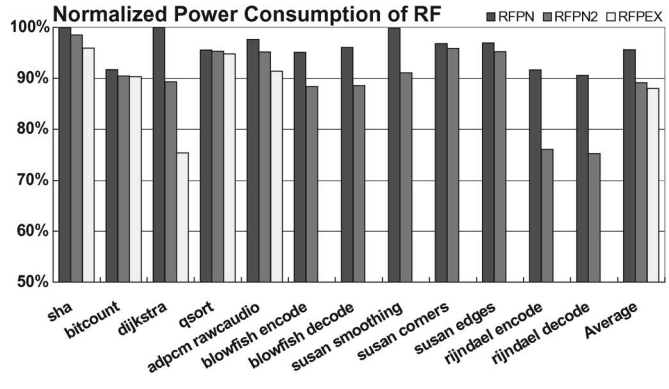

Fig. 6. Normalized power consumption (dynamic+static) of the register file.

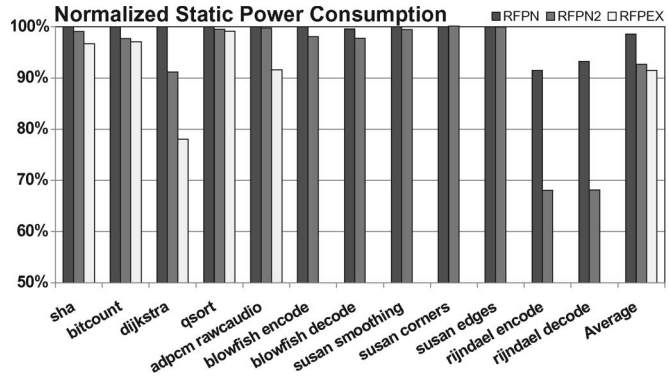

Fig. 7. Normalized static power consumption of the register file.

already achieves. In the architecture without the on-demand RF read technique, RFPN2 can achieve a 46.7% register file power reduction.

To clearly show the impact of our techniques on the leakage power, Fig. 7 shows the normalized static power consumption of the register file. The first observation from this graph is that our heuristic RFPN2 can reduce the leakage power by up to 32% and, on average, 7.3%. The second, and more interesting, observation is that the trend of the leakage power reduction on some of the benchmarks is not similar to that of the rfa reduction. For example, there is no leakage power reduction for qsort and susan benchmarks in Fig. 7, although the number of access is reduced by almost 10%, as shown in Fig. 3. This is because the temperature drops slower when it is below the critical temperature, and faster when it is beyond the critical temperature [12]. For the benchmarks mentioned above, the application size is quite small, and they do not heat the register file beyond the critical temperature. Thus, the reduction in the rfa is mostly used only for the dynamic power. However, for the rijndael benchmarks, they execute for long periods and heat the register file substantially. Therefore, our technique significantly reduces the temperature and the leakage power of the register file. Note that the normalized reduction for rijndael in the leakage power is greater than that of the rfa due to the exponential dependence of the leakage power on the temperature.

V. RFA Reduction Across Bypass Configurations

In this section, we explore the effectiveness of our bypass-sensitive compilation techniques in reducing the register file power consumption by reducing the access frequency to the register file on various bypass configurations.

We perform our experiments by modifying the bypass configuration on the Intel XScale processor. The Intel XScale has seven pipeline stages that can generate a bypass, the X1, X2, XWB, M2, Mx (referred

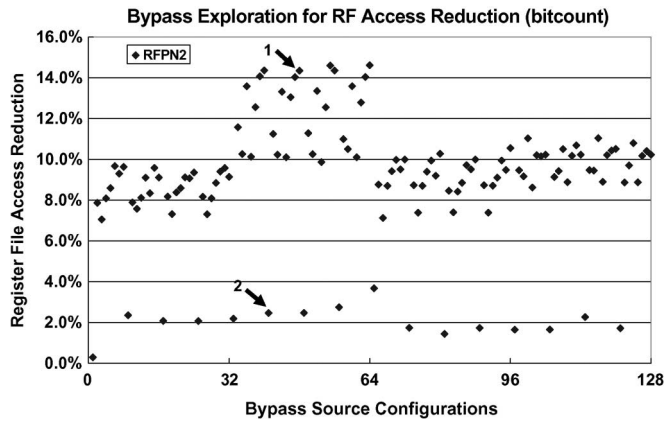

Fig. 8. All bypass explorations on bitcount (rfa).

to as MWB in this paper), D2, and DWB pipeline stages. The bypasses from these stages come to the RF stage. In the RF stage, up to four operands can be read in a cycle. Thus, in the XScale pipeline, there can be, at most, $7 \times 4 = 28$ different bypasses, and there can be 2^{28} possible bypass configurations. Although our approach can model each of these bypass configurations, it is clearly not possible to explore all of these bypass configurations. For our experiments, we explore several interesting bypass configurations from a processor architect's perspective. We assume that if a pipeline unit generates a bypass, then it will be available to all the four source operands in the RF stage. That is, if a stage bypasses, all the four operands can read the result, whereas if a stage does not bypass, none of the four operands can read the result. This constraint restrains the bypass space to just $2^7 = 128$ configurations. The configurations can be represented (encoded) by (into) a 7-b binary number. Each bit in the encoding indicates whether there are bypass connections from the corresponding pipeline stage in the ordered tuple $\langle \mathrm{DWB}, \mathrm{D2}, \mathrm{MWB}, \mathrm{M2}, \mathrm{XWB}, \mathrm{X2}, \text{and } \mathrm{X1} \rangle$.

Fig. 8 shows the percentage reduction of the number of access to the register file achieved by our RFPN2 scheduling technique over a bitcount benchmark. The bars in the graph represent how many accesses RFPN2 can reduce on the on-demand RF read architecture for each bypass configuration. We can make an important observation from this graph. That is, there is no bar which has a negative value, indicating that our technique consistently achieves good rfa reduction even if we change the bypass configuration. In fact, the reduction in the number of access to the register file for a bypass configuration shows up to 14.4% and, on average, 9%, implying the effectiveness and usability of our technique for partially bypassed processors. The second observation is that the reduction in the number of the rfa varies a lot throughout the bypass configurations. For example, design point 1 in Fig. 8 has a $(0101111)_2$ configuration, which represents a configuration which bypasses from D2, M2, XWB, X2, and X1. Furthermore, design point 2 has a $(0101000)_2$ configuration, which contains bypasses from D2 and M2. The difference between these two design points indicates that RFPN2 can generate more power-efficient code for bitcount benchmark if bypasses from the X pipeline stages are present. We can reduce the number of rfa within $\pm 2\%$ variations in the performance (from 2% reduction to 2% improvement of performance). We observed the similar trend in the case of the other benchmarks that we used.

VI. Summary and Future Work

Register file power consumption has been widely recognized as a very important issue, not only to reduce the total power consumption of processor, but also to prevent "heat strokes." In this paper, we

demonstrated that in on-demand RF read architectures, there is scope for further rfa reduction and proposed several bypass-aware instruction scheduling techniques aimed at reducing the number of access to the register file. 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% rfa can be reduced. Further, one of our scheduling techniques, which is RFPN2, is an effective heuristic to reduce the number of rfa (11.4% on average) without much loss in performance (less than 1% on average) and within a reasonable compilation time. We have demonstrated that our compilation technique consistently reduces the number of the rfa on various bypass configurations.

REFERENCES

[1] V. Zyuban and P. Kogge, "The energy complexity of register files," in *Proc. ISLPED*, 1998, pp. 305–310.

[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 *Proc. Conf. Des., Autom. Test Eur.*, 2002, pp. 168–173.

[3] J. Deeney, "Thermal modeling and measurement of large high power silicon devices with asymmetric power distribution," in *Proc. Int. Symp. Microelectron.*, 2002, pp. 300–305.

[4] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall, "Managing the impact of increasing microprocessor power consumption," *Intel Technol. J.*, vol. Q1, pp. 1–9, 2001.

[5] J. Hasan, A. Jalote, T. Vijaykumar, and C. Brodley, "Heat stroke: Power-density-based denial of service in SMT," in *Proc. Int. Symp. High-Performance Comput. Architecture*, 2005, pp. 166–177.

[6] Z. Hu and M. Martonosi, "Reducing register file power consumption by exploiting value lifetime characteristics," in *Proc. Workshop Complexity Effective Des.*, 2000.

[7] N. S. Kim and T. Mudge, "Reducing register ports using delayed write-back queues and operand pre-fetch," in *Proc. 17th Annu. ICS*, 2003, pp. 172–182.

[8] J. H. Tseng and K. Asanovic, "Energy-efficient register access," in *Proc. 13th Symp. Integr. Circuits Syst. Des. (SBCCI)*, 2000, pp. 377–382.

[9] N. Goel, A. Kumar, and P. R. Panda, "Power reduction in VLIW processor with compiler driven bypass network," in *Proc. 20th Int. Conf. VLSID, 6th Int. Conf. Embedded Syst.*, 2007, pp. 233–238.

[10] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau, "Operation tables for scheduling in the presence of incomplete bypassing," in *Proc. 2nd IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign Syst. Synthesis (CODES+ISSS)*, 2004, pp. 194–199.

[11] D. Brooks and M. Martonosi, "Dynamic thermal management for high-performance microprocessors," in *Proc. 7th Int. Symp. HPCA*, 2001, p. 171.

[12] K. Patel, W. Lee, and M. Pedram, "Active bank switching for temperature control of the register file in a microprocessor," in *Proc. 17th GLSVLSI*, 2007, pp. 231–234.

An FSM Reengineering Approach to Sequential Circuit Synthesis by State Splitting

Lin Yuan, *Member, IEEE*, Gang Qu, *Senior Member, IEEE*, Tiziano Villa, and Alberto Sangiovanni-Vincentelli, *Fellow, IEEE*

***Abstract*—This paper presents a finite-state machine (FSM) reengineering method that enhances the FSM synthesis by reconstructing a functionally equivalent but topologically different FSM based on the optimization objective. This method enables the FSM synthesis algorithms to explore a set of functionally equivalent FSMs and obtain better solutions than those in the original FSM. To demonstrate the effectiveness of the proposed method, we apply it to popular power- and area-driven FSM synthesis algorithms, respectively. Our method achieves an average of 5.5% power reduction and 2.7% area reduction, respectively, on 25 Microelectronics Center of North Carolina (MCNC) FSM benchmarks, where the proposed method is applicable. This is a significant performance improvement for the power- and area-driven FSM synthesis algorithms being used. Our method has a negligible run-time overhead, and it maintains the quality of the synthesis solutions.**

***Index Terms*—Finite-state machine (FSM) encoding, power minimization, sequential logic synthesis, state splitting.**

I. INTRODUCTION

Finite-state machine (FSM) synthesis is a well-studied problem. It consists of state minimization (SM) and state encoding (SE) procedures. SM finds a functionally equivalent FSM that has the minimum number of states. SE assigns distinct binary codes to each state of the FSM such that the sequential circuit modeled by the FSM can be efficient in terms of area, performance, and/or power.

The SM problem can be optimally solved in completely specified FSMs [6], and there are standard approaches to solving the SM problem in incompletely specified machines [9]. On the other hand, there have been many techniques to solve the SE problem based on different optimization objectives and implementation technologies.

In area-driven FSM synthesis, De Micheli *et al.* proposed an SE algorithm to minimize area in a programmable logic array implementation by generating a minimum (multivalued) symbolic cover of the FSM followed by a step of satisfying the encoding constraints [13]. Successive extensions also introduced output constraints and more efficient algorithms to satisfy the input and output encoding constraints [18], [19]. MUSTANG [3] is one of the earliest state encoding techniques for multilevel logic minimization; it assigns a weight to each pair of symbols and gives adjacent codes to pairs of states with large weight. JEDI [11] adopts a weighted graph model similar to the one in MUSTANG, but it uses a simulated annealing algorithm to perform the embedding. Instead, MUSE [4] and MIS-MV

Manuscript received October 25, 2005; revised March 8, 2006, March 10, 2007, July 24, 2007, and October 30, 2007. This paper was recommended by Associate Editor S. Nowick.

L. Yuan is with Synopsys, Inc., Mountain View, CA 94043 USA (e-mail: Lin.Yuan@synopsys.com).

G. Qu is with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA (e-mail: gangqu@glue.umd.edu).

T. Villa is with the Department of Computer Science, University of Verona, 37134 Verona, Italy, and also with the Project for Advanced Research of Architecture and Design of Electronic Systems, 00186 Roma, Italy.

A. Sangiovanni-Vincentelli is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 USA, and also with the Project for Advanced Research of Architecture and Design of Electronic Systems, 00186 Roma, Italy.

Digital Object Identifier 10.1109/TCAD.2008.923245