POWER-AWARE REGISTER RENAMING

COMPUTER ENGINEERING GROUP TECHNICAL REPORT 01-08-02

Andreas Moshovos[‡]
Electrical and Computer Engineering
University of Toronto
moshovos@eecg.toronto.edu

Abstract

We propose power optimizations for the register renaming unit. Our optimizations reduce power dissipation in two ways. First, they reduce the number of read and write ports that are needed at the register alias table. Second, they reduce the number of internal checkpoints that are required to allow highly-aggressive control speculation and rapid recovery from control flow miss-speculations. To reduce the number of read and write ports we exploit the fact that most instructions do not use the maximum number of source and destination register operands. We also use intra-block dependence detection logic to avoid accessing the register alias table for those operands that have a RAW or a WAW dependence with a preceding, simultaneously decoded instructions. To reduce the number of internal checkpoints we propose out-of-order control flow resolution as an alternative to the conventional method of in-order resolution. We study our optimizations for a set of integer applications from the SPEC2000 and for a multimedia application from mediabench and for an aggressive 8-way superscalar. We find that it is possible to reduce the number of read ports from 24 to 12 and the number of write ports from 8 to 6 with a minor slowdown of 0.5% (worst case slowdown of 2.2%). Furthermore, we find that out-of-order control flow resolution achieves performance that is within 0.6% of that possible with an infinite number of checkpoints (worst case slowdown of 0.8%). We model the maximum power dissipation of the rename unit for a 0.18um process and demonstrate that our methods can reduce overall rename power by 42%.

1 Introduction

Power dissipation has emerged as a first design consideration even for high-performance processors. In such designs, the concern is maximum power dissipation that ultimately impacts cost or even the feasibility of the design. This is in contrast to mobile systems where the consideration is battery life. Naturally, most work in power-aware techniques for high-performance processors has focused on those units that account for a large fraction of overall on-chip power dissipation such as the level one caches, the datapath, the register file and the instruction scheduler.

Since silicon is not a good heat conductor, another not so well understood parameter, *power density*, is also becoming similarly important to power dissipation alone. In particular, there are units that while do not consume a large fraction of overall power, do exhibit extremely high *power density.* These are units whose overall power dissipation is disproportionately large compared to their area. Consequently, such units create hot spots and their temperature tends to significantly exceed that of other on-chip units. A high operating temperature can lead to transient or permanent faults.

In modern microprocessor designs a unit that exhibits both high power density and also dissipates a considerable amount of on chip power is the register rename unit [7]. For example, the Pentium Pro rename unit dissipates about 4% of the overall on-chip power [8]. Moreover, its power-density is among the four highest on the chip. It falls just behind that of the decode and execution units. Motivated by these observa-

[‡]*Currently with the Hellenic Armed Forces.*

tions in this paper we propose a set of simple, yet effective techniques for reducing the power of the rename unit.

Several approaches to reducing power density may be possible. For example, we may seek to maintain the same power and delay while distributing the specific logic over a larger area. In this work, we focus on reducing the power dissipated by the unit. We propose power-aware optimizations that aim at reducing (1) the number of read and write ports at the register alias table (RAT), and (2) the number of control-flow related checkpoints necessary for achieving high-performance. All optimizations reduce the number of resources that need to be accessed per cycle and hence reduce power dissipation.

To reduce number of RAT ports we first observe that conventional renaming units are designed for the worst case where the maximum number of instructions is simultaneously renamed and where all instructions use the maximum number of source and destination register operands. However, it is understood that not all instructions use the maximum number of source and destination register operands [6]. Moreover, the rename unit may not need to always rename the maximum number of instructions to sustain high-performance. Finally, we observe that it is possible to exploit intra-block RAW and WAW dependences to avoid accessing the RAT for the corresponding registers. To reduce the number of checkpoints held on branches we observe that conventional designs use in-order control-flow resolution. That is, a branch remains unresolved until *it* and *all* preceding branches decide their directions. While this was an appropriate design decision for small scale superscalars, the number of checkpoints grows quickly for wider and deeper processors. This in turn increases the power dissipation whenever branches are renamed or when miss-speculations occur. To maintain high-performance while drastically reducing the number of required checkpoints we propose *out-of-order control-flow resolution*. With this technique a branch is resolved and the correspong checkpoint is discarded as soon as the branch decides its direction.

We study the register renaming requirements of several ordinary programs. These include integer applications from the SPEC2000 and a multimedia application. We demonstrate that our methods can reduce the number of RAT ports to half of that of a conventional design and the number of checkpoints by about 33%. We use WATTCH, hand-created and synthesized layouts to estimate the power dissipation of the various units comprising the renaming unit and demonstrate that our methods can reduce power by 42%. Furthermore, we show that the multimedia application places much higher demand on the renaming unit than the other applications. To the best of our knowledge this is the first work that studies the register renaming requirements of ordinary programs, that proposes out-of-order branch resolution.

The rest of the paper is organized as follows. In section 2 we review the operation of a conventional register renaming. In section 3 we use this discussion to introduce our power optimizations. First we describe the port reduction optimizations (section 3.1) and then we describe out-of-order control flow resolution (section 3.2). In section 4 we present the benchmarks and our performance simulation methodology. In section 5 we study the performance impact of reducing the number of RAT ports. In section 6 we do the same for the control-flow resolution method. We report our power measurement methodology and the resulting power reduction in section 7. We review related work in section 8. We summarize our findings in section 9.

2 Register Renaming

In this section, we review a commonly used register renaming scheme. This description forms the foundation upon which we can later describe our optimizations. In the interest of space we will use the term *renaming* to refer to register renaming.

2.1 Renaming a Single Instruction

Renaming is used to remove name dependences (WAR or WAW) and to support speculative execution. Figure 1(a) illustrates the steps required to rename a single instruction. Without the loss of generality we

assume a typical load/store ISA where an instruction may have up to one destination register (*ARd*) and up to two source registers (*ARs1* and *ARs2*). The register names as used by the instruction refer to the *architectural* register name space. Renaming maps these names into a different *physical* register name space that is larger. Renaming entails the following actions: (1) Finding the current physical names *PRs1* and *PRs2* for the source registers ARs1 an ARs2. (2) Finding the current physical name *PRd* for the destination register ARd. (3) Assigning a new physical register for the destination register ARd. Actions (1) and (3) are used to remove false dependences and to facilitate out-of-order scheduling later on. Action (2) is required to support speculative execution. If this instruction is squashed the destination register has to be restored to its previous mapping.

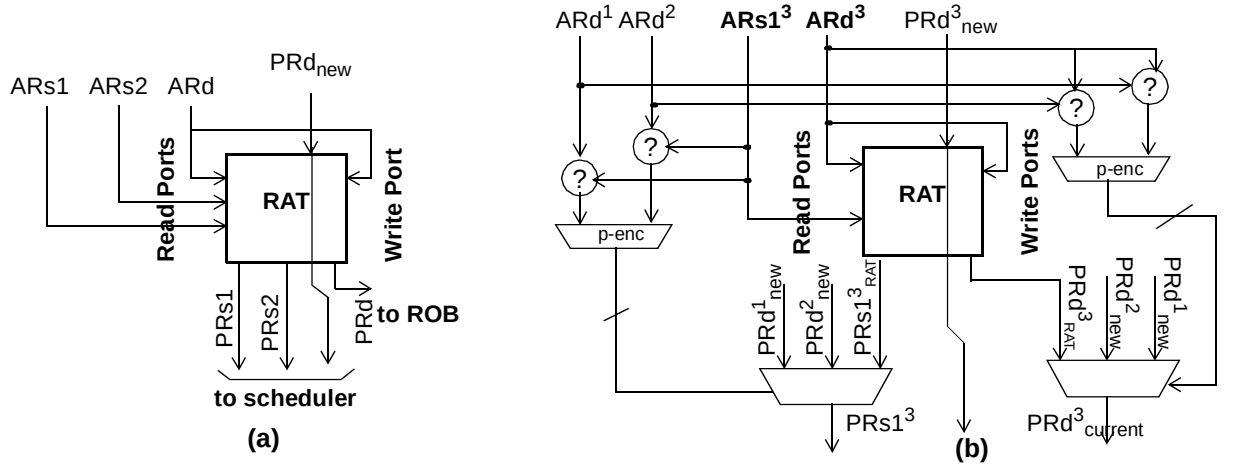

Figure 1: *(a) Renaming of a single instruction. (b) Renaming multiple instructions simultaneously. Shown is the logic only for renaming one of the source registers and the destination register of the third instruction in a superscalar processor.*

The core of the renaming activity takes place in the *register alias table* (RAT). The RAT holds the current mapping of architectural to physical registers. There are two common implementations, one is based on a simple SRAM table and the other is based on a CAM (see related work section). For the purposes of this study we focus on the former. In this scheme, renaming proceeds into two phases. In the first phase, actions (1) and (2) take place where the RAT is used to locate the current physical names for the source and destination registers (*PRs1*, *PRs2* and *PRd* respectively). In phase two, the RAT is updated to reflect the new mapping PRd_{new} of the destination register (if any exists). The new physical name is provided by the register free list which can be implemented in a variety of ways. The current mapping for the destination register *PRd* is stored into the *reorder buffer* (ROB). It will be used to restore the RAT into a consistent state if this instruction is squashed. The renamed instruction is then shipped to the scheduler where it will await execution. Finally, the register free list is updated accordingly to reflect that the PRd_{new} name is no longer available (names are released on commit or on squash).

Because branch miss-predictions are relatively frequent, a second checkpointing mechanism is used to facilitate the rapid restoration of renaming state. In this case the RAT is capable of checkpointing all register mappings every time a branch is renamed. If a branch is miss-speculated the corresponding checkpoint is used to restore *all* mappings simultaneously. Similar checkpointing mechanisms are incorporated in all related structures including the register free list and the ROB. This facility is typically implemented as a circular buffer which is physically located next to each RAT bit cell. The depth of this circular buffer limits the number of checkpoints that can be held and hence it sets an artificial limit on the number of branches that can be simultaneously unresolved. For example, in the MIPS R1000 four such checkpoints were available [1]. As we will see in section 6, many more are required for future generation processors. Note that since these snapshots are taken only for branches, they cannot be used to restore the renaming state on other squashing events (e.g., page faults or interrupts). To handle these events, we use the mapping records

held into the ROB. For example, in the MIPS R1000 the updates of up to four instructions could be undone per cycle.

In this work we focus on power optimizations for the RAT. As we explain next, when renaming multiple instructions the number of ports and hence the size and power of the RAT increases rapidly. Future work could explore power optimizations for the register free list and the ROB.

2.2 Superscalar Renaming

Typical high-performance processors already rename multiple instructions per cycle. It is to be expected that future generation processors may have to rename even more. When renaming multiple instructions simultaneously, the RAT-scheme we have just described needs to be extended to cope with multiple instructions and with the dependences that may exist among them.

As we explained, to rename a single instruction that may have up to two source registers and up to one destination register, a RAT with three read and one write ports is needed. To rename *N* instructions, the RAT needs to have *3xN* read ports and *N* write ports. Up to *2xN* read ports are needed to locate the current mappings for the source registers and the other *N* read ports are needed to retrieve the current mappings for the destination registers prior to renaming them. These mappings will be stored into the ROB. Accordingly, the ROB needs to have *N* write ports. The *N* write ports are used to update the RAT and its internal checkpoints with the the up to *N* new mappings for the destinatiom registers. Consequently, the register free list needs to be capable of providing up to *N* physical register names per cycle.

Renaming still proceeds into two phases where in the first we read the current mappings for all registers and in the second we record the new mappings for all destination registers. Since we are renaming a *block* of multiple instructions simultaneously it is possible that *intra-block* dependences exist. In particular, we need to correctly handle intra-block RAW and WAW dependences. In existing superscalar renaming organizations, all RAT reads proceed in parallel. As a result, the physical register names provided by the end of the RAT read cycle do not reflect any renames from instructions within the current block. The physical register names returned by the RAT are the ones that the registers had before *any* of the instructions in the block. To appropriately handle intra-block RAW dependences an additional unit is used which operates in parallel with the RAT. This unit operates as follows. The unit compares any source register names with the destination register names of all preceding instructions. For example, as shown in Figure 1(b), the first source register name (architectural) $ARs1^3$ of the third instruction, is compared with the destination registers ARd^1, ARd^2 of the first and second instructions respectively. The results of the comparisons are priority encoded and are used to drive a multiplexer that selects the appropriate physical register name. This has to be the most recent physical register assigned to the specific architectural register. In priority sequence this can be either the new physical register assigned to the destination register of the second instruction (PRd^2_{new}), that of the destination register of the first instruction (PRd^1_{new}), or if no match exists, that returned by the RAT (PRd^2_{RAT}). Note that while in the figure we show a priority encoder followed by a multiplexer, in an actual implementation, the comparison signals will be priority gated and then used to drive directly the pass gates implementing the multiplexer.

In addition to the logic that detects intra-block RAW dependences, similar logic is used to track intra-block WAW dependences. This is needed so that the mapping information stored in the ROB (for miss-speculation recover purposes) correctly reflects the most recent mapping for each register. In this case, for each destination register name, a set of comparators is used to compare it with the all preceding destination registers. The comparison results are prioritized to select the one that corresponds to the instructions that is closest in program order, or the result of the RAT read if no intra-block WAW dependence exist.

In the second phase, the new mappings for all destination registers are written into the RAT. Note that all rename updates need to become visible by the RAT even in the case of an intra-block WAW dependence between two instructions A and B. While subsequent instructions will not need the mapping created by

instruction A, recall that the RAT incorporates a checkpointing mechanism for handling branch miss-speculations. The mapping created by A will be stored in a separate checkpoint if a branch appears in between A and B.

3 Power Optimizations

We propose power optimizations that fall into two categories. The first set of optimizations reduces the number of read and write ports needed by the RAT. The second optimization, reduces the number of branch RAT checkpoints needed to sustain high-performance.

3.1 Reducing the Number of RAT Read and Write Ports

Conventional renaming hardware, very much like many other high-performance designs, is built to accommodate worst case demand. In particular, a conventional RAT has *3xN* read ports and *N* write ports since it is designed to simultaneously rename *N* instructions that have two source registers and one destination register. For small degrees of superscalarity, such a design is very appropriate since it is simple and fast. However, as we move towards wider superscalars, the delay and power of a full-blown RAT increase creating an opportunity for revisiting this design decision. To reduce the number of RAT ports we exploit the following two opportunities: First, not all instructions read two source operands or write a register. Second, when an intra-block RAW or WAW dependence exists it is not necessary to read the mapping stored in the RAT for the corresponding source or destination register. The correct name in this case is that created for the destination register of the earlier in program order instruction.

Our optimizations reduce the number of RAT ports so that it can sustain high performance compared to a full-blown RAT. Since the number of RAT ports is reduced, it is possible that we will not be able to simultaneously decode a sequence of instructions whose RAT read requirements exceed the number of RAT ports. However, as we show in section 5, it is possible to drastically reduce the number of RAT ports and still sustain high performance for ordinary programs.

It is well understood that many instructions do not require two source registers or may not produce a register result. Many read only a single source registers (e.g., addition with an immediate), while others do not read any source registers (e.g., move immediate). Some instructions do not write a register (e.g., branches or stores). It is possible to reduce the number of RAT ports by first detecting which opcode fields represent actual register identifiers and then by routing only these to the RAT. Upon accessing the RAT we need to reshuffle the results into position for writing into the ROB and the scheduler. This detection and routing process is not without power and delay overheads. As we show in section 7, the power of the routing network is an order of magnitude smaller than that of the RAT for an 8-way superscalar (we have implemented and simulated the routing logic in a 0.18um technology). As far as delay overheads are concerned there are two mitigating factors: First, since we are reducing the number of RAT ports we are also reducing the latency of the RAT compared to a full-blown implementation. It is very likely that the overhead of the routing network will be offset by the decrease in RAT latency. Furthermore, future superscalar processor most likely will be heavily pipelined, accordingly, including an additional pipeline stage to accommodate for the routing network and register identifier detection logic will have a small impact on performance (as we show in section 5). Note that additional opportunities for reducing the latency of this optimization exist such as including pre-decode bits in the I-Cache or embedding this information in the trace cache if one exists.

It is possible to further reduce the number of RAT read ports by exploiting intra-block RAW and WAW dependences. In particular, we do not need to read the mapping stored in the RAT for a source register that reads the result created by a preceding instruction within the block (RAW dependence) or the previous mapping for a destination register that overwrites the destination register of a preceding instruction within the same block (WAW dependence). In the extreme case, we may use *full* intra-block RAW and WAW dependence detection and RAT read elimination. That is, we may first detect *all* intra-block RAW and

WAW dependences and then proceed to access the RAT. In this case, the intra-block dependence detection logic that *already exists*, now operates in series with the RAT as opposed to in parallel. The same routing network we introduced for selecting the actual register operands only can be used for preventing registers with WAW or RAW dependences from accessing the RAT. Operating all of the dependence detecion logic in series with the RAT will obviously increase overall rename latency. As we show section 5.3, however, most RAW dependences are between adjacent instructions. Accordingly, we could implement *partial* RAW and WAW intra-block dependence detection and RAT read elimination. For example, we could wait only for the comparisons between adjacent instructions to finish before we access the RAT. In this case, the RAT read path is increased only by a single level of comparators that operate in parallel. The increase in latency will be much smaller in this case.

3.1.1 Performance Implications

It is important to note that the performance implications of the aforementioned optimizations can be fairly complex. In particular, the average number of register operands that are simultaneously renamed can be a deceiving metric. What is important is how the new RAT design impacts the *performance critical path*. Consider the following two examples assuming an 8-way superscalar. In the first case, assume that all instructions are independent. To sustain maximum performance, the RAT should be capable to rename *all* eight instructions in parallel, otherwise it will be a performance limiter. If, however, the instructions form a dependence chain even a RAT that can decode a single instruction per cycle will offer maximum performance. As we show section 5.4, while on the average very few register operands are used, the RAT needs to be able to rename a much larger number of registers to achieve acceptable performance.

3.2 Reducing the Number of Branch Checkpoints

A checkpoint is created for each branch as soon as it is renamed and it remains allocated as long as the branch is *unresolved*. Conventional branch checkpointing mechanisms use *in-order control-flow resolution.* That is, a branch remains unresolved until (1) it decides what direction it will follow, and (2) all preceding branches do the same. The use of in-order control-flow resolution in high performance processors was justified since it is simple. It results in a checkpointing mechanism that operates as a circular buffer that can be distributed and embedded along with the information bits it needs to checkpoint (e.g., the RAT contents). Furthermore, the control-flow resolution logic is also relatively simple. However, as we move towards deeper instruction windows, the number of checkpoints required increases. As we show in section 6, for an 128-entry window, we may need even up to 32 checkpoints to sustain acceptable performance for all programs we studied. Creating RAT bit cells that include a 32-entry circular buffer becomes questionable impacting both power and performance (i.e., by elongating the wordlines).

To reduce the checkpoints we propose using *out-of-order control-flow resolution,* a simple, yet effective alternative. In this scheme, a branch is resolved as soon as *it* decides which direction it will follow. That is, we do not require that all preceding branches decide which direction they will follow also. To understand why this scheme is correct one has to consider the two possible scenarios where a branch B is allowed to resolve and discard its checkpoint out-of-order. In the first, eventually all preceding branches resolve correctly (i.e., no miss-speculation). In this case, the checkpoint held by B is not needed. In the second scenario, a preceding branch resolved incorrectly and squashes all subsequent instructions. It uses its own checkpoint to restore the RAT state. Hence, in this case also B's checkpoint is not needed. Consequently, in all possible cases, a branch that decides its direction can safely discard its own checkpoint.

While there are no correctness implications, there are implementation and possibly performance implications. In the event of a non-control-flow related exception, we will have to use the ROB to incrementally restore the renaming state. When all branch checkpoints are available, it may be possible to reduce the number of steps required to restore the state. By first restoring the checkpoint associated with the branch that immediately follows the offending instruction, we can reduce the number of updates that have to be restored by the ROB. However, note that if the checkpoints are implemented as a circular buffer, we may

still have to perform several shifts before we reach the appropriate branch checkpoint. The last point allows us to explain a potential performance advantage of out-of-order resolution. Since the checkpoint mechanism is no longer a circular queue, it can immediately restore the appropriate checkpoint.

The checkpoint mechanisms can no longer be a circular queue. Instead a regular register file-like structure is required. Embedding such a structure along with each bit becomes questionable for several reasons including the need for separate decoders per RAT bit. Accordingly, we assume an implementation where the checkpointing cells are embedded next to each RAT row. This increases the height of the RAT row albeit only slightly (e.g., for 128 or 256 physical registers) compared to the conventional circular buffer implementation that requires several control signals (a horizontal wire per RAT bit needs to be sent across to the checkpointing file). Furthermore, since the checkpointing storage is not embedded into each RAT bit, the length of the RAT wordline is kept at a minimum, decreasing power and delay.

4 Analysis Overview and Methodology

We start our analysis of our power-aware renaming techniques in section 5 where we investigate the effect of reduced read and write ports. We report statistics on the number of simultaneously decoded instructions, the number of intra-block dependences (RAW and WAW) and the distance between dependent instructions. Finally, we report the performance impact of reducing both read and write ports. In section 6 we study our out-of-order control-flow resolution method and show that it can significantly reduce the number of checkpoints needed to achieve high performance. In section 7 we explain our power measurement methodology and report the power savings offered by our methods.

To study our register renaming power optimizations, we use Simplescalar v3.0 [9] to simulate an aggressive wide-issue, 8-way dynamically-scheduled superscalar processor. Our processor is deeply pipelined to appropriately reflect modern processor designs. Table 1 depicts the base processor configuration parameters. Table 2 depicts the SPEC2000 programs we study. We focused on primarily integer programs. We also included *mpeg2encode* from mediabench as a representative of modern media-oriented applications. We simulated up to 2 billion instructions per benchmark after skipping a sufficient number of instructions to exclude the initialization. The binaries were compiled for the MIPS-like PISA architecture using GNU's gcc v2.9.

Branch Predictor	16k GShare+16K bi-modal with 16K selector 2 branches per cycle	**Fetch Unit**	Up to 8 instr. per cycle 64-entry Fetch Buffer 2 branches per cycle Non-blocking I-Cache
Instruction Window Size	128 entries RUU-like	**Load/Store Queue**	64 entries, 4 loads or stores per cycle Perfect disambiguation
Issue/Decode/Commit Bandwidth	any 8 instructions / cycle	**Functional Unit Latencies**	same as MIPS R10000
DL1/IL1 Geometry	64Kbyte/32byte blocks/4-way SA	**UL2 Geometry**	1Mbyte/64byte blocks/8-way SA
L1/UL2 Access Latencies	3/16 cycles	**Main Memory**	Infinite, 100 cycles
Stage Latencies	8 cycles from branch predict to decode stage 5 cycles for decode and renaming + overhead of detecting and routing register operands where appropriate 6 cycles from writeback to commit 10 cycles branch misprediction penalty.		

Table 1: *Base processor configuration.*

5 Reducing the Number of RAT Ports

In this section, we present experimental results that demonstrate to what extend we can exploit the actual number of register operands and the existence of intra-block dependences to reduce the number of RAT ports.

App.	Label	Sim. Insts.	Skip	Base IPC
164.gzip	*gzp*	1.57B	1B	1.67
175.vpr	*vpr*	2.00B	4B	0.99
176.gcc	*gcc*	1.59B	0	1.13
181.mcf	*mcf*	2.00B	2B	0.24
197.parser	*prs*	1.88B	3B	1.27
255.vortex	*vor*	2.00B	4B	2.69
256.bzip	*bzp*	2.00B	3B	1.93
300.twolf	*twf*	2.00B	5B	0.77
mpeg2encode	*mpg*	1.13B	0	0.99

Table 2: *Applications used in our studies. Reported from left to right are the resulting committed instruction counts, the number of instructions we skipped prior to measuring program behavior and the base IPC.*

5.1 Number of Simultaneously Renamed Instructions

The first relevant behavior is the number of instructions that are simultaneously renamed by the RAT. A conventional design for an 8-way superscalar is capable of renaming up to 8 instructions. It is interesting to know to what extend this peak capability is used in practice. Figure 2 shows the distribution of the number of simultaneously decoded instructions per program. The range is zero (none) through eight (peak). On the average, more than half of the time, the rename logic sits idle. There are several reasons why this is so. The two more relevant for most programs are branch miss-predictions and I-Cache stalls since for most programs the window is rarely full. The relatively low utilization in mcf is the result of the high data cache miss rate exhibited by this program. In this case, the window is very often full waiting for memory data. Overall, while the rename logic often has no instructions to rename, when there are instructions it operates mostly at its peak.

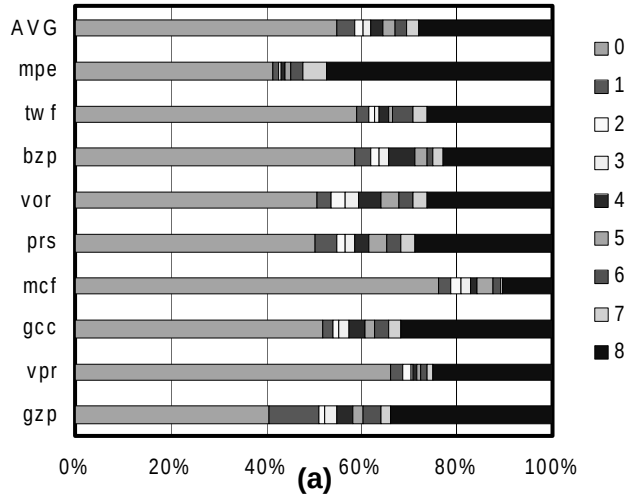

Figure 2: *Distribution of the number of instructions that are simultaneously renamed.*

5.2 Number of Source and Destination Registers

The number of simultaneously decoded instructions is only loosely related to the number of register operands that need to simultaneously access the RAT. As we explained in section 2.2, source register operands need to determine their current physical register location while destination register operands need to first determine the current physical register location of the registers they overwrite and then update them to point to the newly allocated physical registers. For our ISA and for the 8-way superscalar configuration,

there may be up to 16 source register operands and up to eight destination register operands. Figures 3(a) and 3(b) show the distribution of the number of simultaneously renamed source and destination register operands respectively. The R0 operand is not included in the operand counts since this register does not need to be renamed.

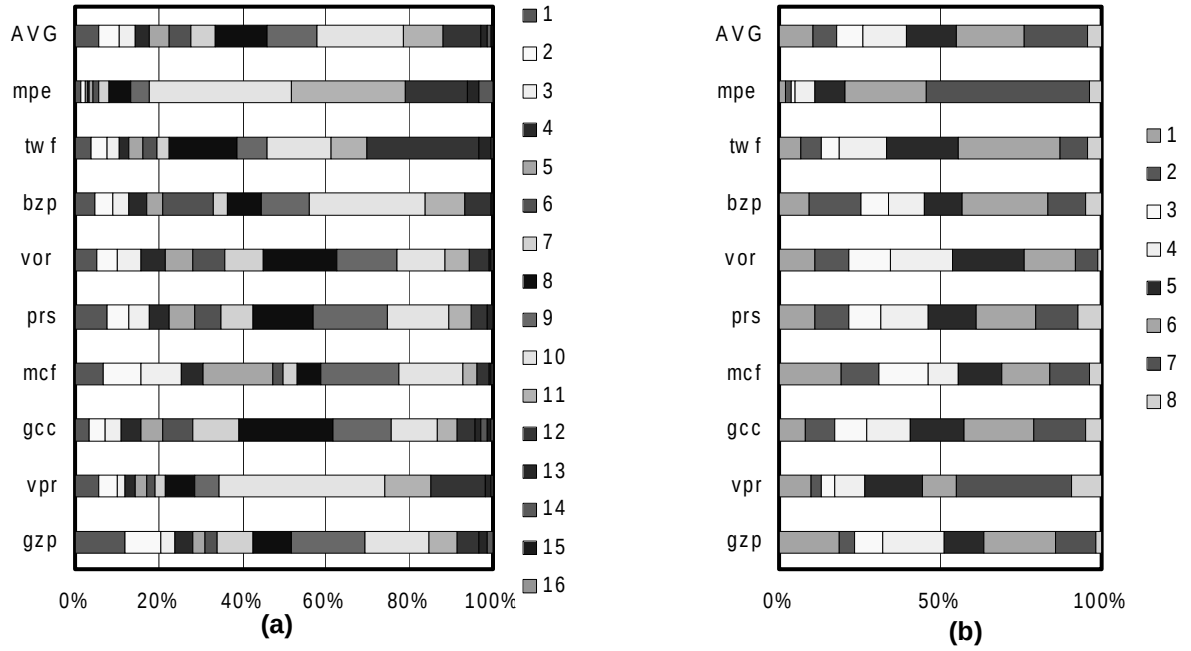

Figure 3: *(a) Distribution of the number of simultaneously renamed source register operands. (b) Distribution of the number of simultaneously renamed destination register operands. Both distributions include cycles during which there is at least one instruction available for renaming.*

As shown in figure 3(a), on the average, about 58% of the time, up to nine source register operands access the RAT, while about 90% of the time, only up to 11 source register operands exist. The percentage of time all 16 read ports for source register operands are used is virtually zero. The programs appear to fall into two groups. The first includes mpeg2encode, twolf and vpr that exhibit a much larger number of source register operands. The second group exhibits a much smaller number of source register operands per rename block. Focusing on the number of destination operands and on figure 3(b) we can observe that about 75% of the time only up to six operands are simultaneously renamed. Mpeg2encode and vpr exhibit significantly different behavior than the other programs having a much higher percentage of time where seven or eight destination operands exist.

The results of this section provide a first indication that it may be possible to reduce the number of RAT ports. We show that in practice, most of the time up to 11 source and six destination register operands exist. As we explained in section 3.1.1, this result is only an indication. The actual tolerance to a reduced number of RAT ports can be determined when the effects of execution and the critical path are accounted for.

5.3 Intra-Block Dependences

In addition to determining and routing only the actual source and destination register operands, we may further exploit intra-block RAW and WAW dependences to further reduce pressure for RAT read ports. Figure 4 reports the fraction of intra-block RAW dependences. This is measured as the number of source register operands that have a RAW dependence with a preceding, simultaneously renamed instruction. On the average, 42% of source registers have a RAW dependence and hence need not access the RAT. Figure 4(b) shows the distribution of the number of RAW dependences per cycle when there is at least one

instruction to be renamed. On the average, 40% of the time (sum of bars greater than three) there are at least four RAW dependences. As before, mpeg2encode, vpr and twolf behave quite differently than the other programs exhibiting a much larger number of intra-block RAW dependences. This result suggests that it may be possible to compensate for their larger number of register operands (that we observed in section 5.2) by exploiting their higher fraction of interact block dependences. Overall, rarely we see eight or more intra-block dependences. Finally, figure 4(c) shows the dependence distance distribution for intra-block RAW dependences. We define *dependence distance* as the distance in dynamic instructions between the producer and the consumer. If the dependence is between adjacent instructions the dependence distance is one. On the average, 55% of the dependences are among adjacent instructions, while this percentage rises to 87% when we consider distances of up to three instructions. This result suggests that we may be able to get most of the benefits by detecting dependences between adjacent instructions prior to accessing the RAT. This is important as the latency of these comparisons will be smaller than that needed to detect all dependences. Furthermore, these comparisons could be used to gate the comparators for longer distances (such an investigation is beyond the scope of this paper). Detecting all possible dependences would require up to seven comparators for the operands of the last instruction.

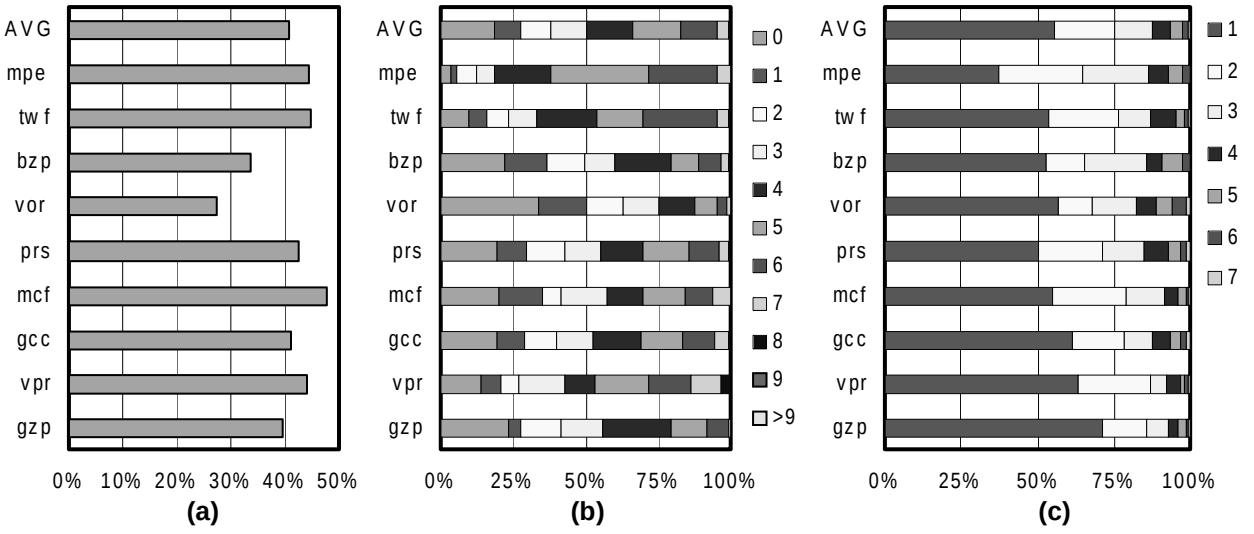

Figure 4: *(a) Fraction of intra-block RAW dependences. (b) Distribution of intra-block RAW dependence count. (c) Distribution of the dependence distance for intra-block RAW dependences.*

Figures 5(a) through (c) report statistics for intra-block WAW dependences. All measurements are for cycles during which there is at least one instruction available for renaming. Part (a) shows the fraction of destination registers that have a WAW dependence. On the average, about one very three do. Part (b) shows the distribution of the number of intra-block WAW dependences. About 45% of the time there are at least two WAW dependences. Three or more WAW dependences exist only for about 23% of the time. Finally, part (c) shows the dependence distance distribution. Here, two comparators per destination register are needed to detect more than half of the WAW dependences (sum of bars 1 and 2). The fraction of WAW dependences between adjacent instructions is relatively low. These dependences exist when a register result is immediately overwritten. Note that this does not mean that the result is useless. It could be that it is both a source and a destination for the second instruction.

The results of this section demonstrated that there is significant additional opportunity for reducing the number of RAT ports. About 40% of source registers have an intra-block RAW dependence and about 20% of them have a RAW dependence with their immediately preceding instruction. Finally, about one every three instructions has an intra-block WAW dependence.

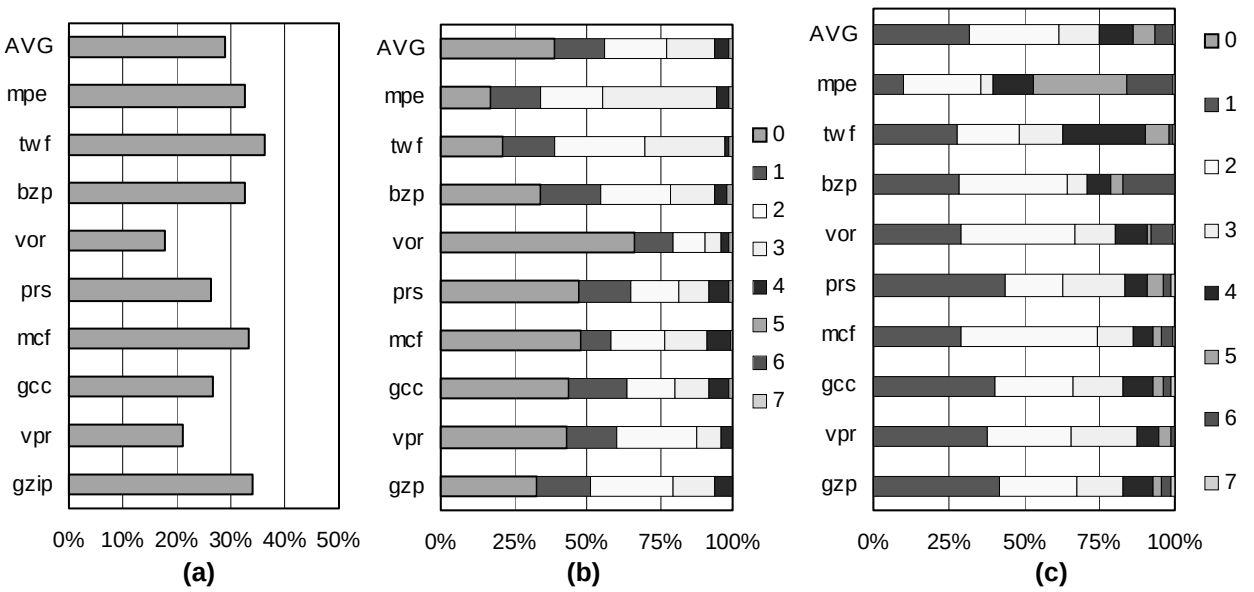

Figure 5: *(a) Fraction of intra-block WAW dependences. (b) Distribution of intra-block WAW dependence count. (c) Distribution of the dependence distance for intra-block WAW dependences.*

5.4 Performance With Reduced RAT Ports

We first investigate the impact of reduced write ports and then proceed to study the impact of reduced read ports. Write ports are used during the second phase of renaming to write the new physical registers assigned to each destination operand.

Figure 6(a) reports the performance slowdown over the base processor when the number of RAT write ports is reduced from eight down to six or four (in the interest of space we omit intermediate design points). With the exception of mpeg2encode, performance stays within 2% of the base even when the RAT has half the write ports. This result justifies the inclusion of mpeg2encode in our benchmark suite. Had we restricted our attention to SPEC2000, we would have concluded that four write ports result in virtually identical performance. Performance slowdowns are higher for mpeg2encode, vpr and gzip. The first two programs exhibited a much larger average number of destination operands as we have seen in section 5.2. Gzip has a relatively low branch prediction accuracy. In this program, when a miss-prediction occurs it is important to quickly fill as much of the window as possible. With a reduced number of write ports additional stalls are introduced impacting performance. Similar phenomena can be observed for other programs that have relatively weak branch prediction accuracy such as gcc. In the rest of the experiments we will use a RAT with six write ports. On the average, the slowdown is less than 0.1% and in the worst case of mpeg2encode about 0.6%.

Figure 6(b) reports performance slowdown when we also reduce the number of read ports. We report results for 20, 16, 12, and 8 read ports. Here we assume that only the actual register operands are routed into the RAT and only those that do not have a RAW or WAW intra-block dependence of *any* distance. Even with 16 or 12 read ports performance is virtually unaffected for all programs except vpr and mpeg2encode. For those two programs, slowdown remains at an acceptable 1.8% and 2.2% respectively. While we omit these results in the interest of space we report that a renaming unit with 12 read ports that can detect dependences at a distance of up to two offers virtually identical performance with that can detect dependence of any distance.

5.4.1 Introducing Additional Pipeline Stages

So far, we assumed that our optimizations do not impact the overall latency of renaming. As we

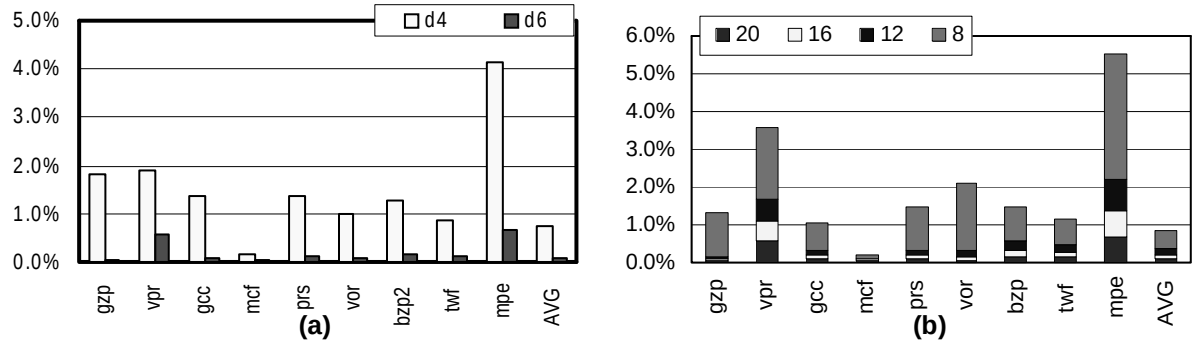

Figure 6: *Performance slowdown with reduced RAT ports. (a) Reduced write ports. (b) Reduced read ports in addition to (a).*

explained in section 2.2, this may be the case since the reduced port RAT is now faster and hence we may be able to use the extra time to perform the register and dependence detection along with all necessary routing in and out of the RAT. For completeness we report in Figure 7 the performance impact of introducing additional pipeline stages for renaming. Our base configuration uses five cycles for decode/rename. With an additional stage, performance on the average stays within 1.2%. However, specific programs behave worse than others. For example, the slowdown is about 4% for mpeg2encode. For the specific program this is comparable to a 6-way superscalar that still has higher port requirements.

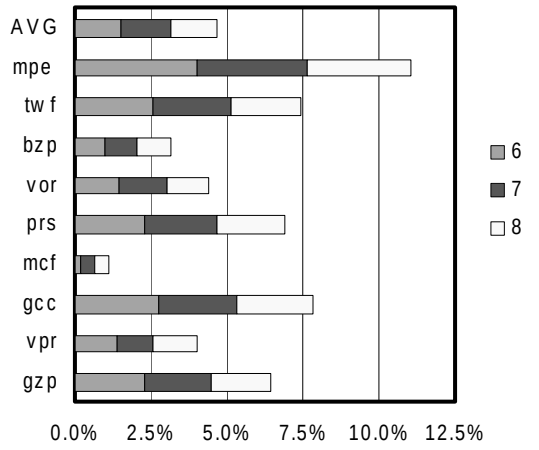

Figure 7: *Performance slowdown when the rename/decode pipeline becomes deeper. The base pipeline has five stages.*

In the rest of the experiments we will restrict our attention to a RAT with six write and 12 read ports. On the average, the slowdown compared to a RAT with eight write ports and 24 read ports is less than 0.5% and in the worst case of mpeg2encode is 2.2%. That is if we assume that it is possible to detect and route the appropriate register operands without impacting the number of pipeline stages. Assuming one additional stage, performance on the average stays within 1.7% and in the worst case is within 6% of the base (for mpeg2encode).

6 Reducing the Number of Branch Checkpoints

In the interest of space we omit branch checkpoint related statistics and report only the performance slowdown with the two control-flow resolution policies. These are the conventional in-order and our proposed out-of-order. On the average, the conventional method requires at least 16 checkpoints to stay within 5% of the base. It needs at least 24 checkpoints to approach within 1.2% of the base. On the contrary, our

out-of-order method achieves comes within 6% with just eight checkpoints and approaches within 0.3% with just 16 checkpoints. When we look upon individual benchmarks we can observe that the conventional method requires 32 checkpoints to maintain its performance close to 2.5% for mcf. In mcf, several branches use data that misses in the caches an for this reason remain unresolved for long periods of time. These branches delay the resolution of many other branches that exist in the window. Many other programs need at least 24 checkpoints to maintain performance close to the base. For example both gzip and parser exhibit a slowdown of more than 5% with 16 checkpoints.

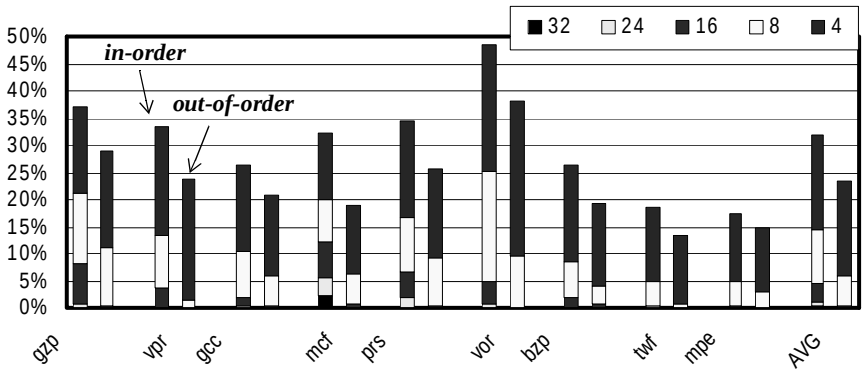

Figure 8: *Performance slowdown with in-order (left) and out-of-order control-flow resolution policies as a function of the number of available checkpoints.*

We conclude that our out-of-order policy can reduce the number of checkpoints to just 16 compared to the 24 needed on the average by the conventional in-order policy. If worst case performance is of prime importance, the conventional policy needs at least 32 checkpoints.

7 Power Reduction

To measure the power reduction possible via our methods we used a methodology similar to that used for the development of the WATTCH power models [4]. In particular we used WATTCH's register file model to estimate the maximum power dissipated by the SRAM arrays of the RAT. We extended the model to include 32 SRAM cells next to each bit along with the effect on the wordlines to account for a conventional embedded checkpointing mechanism. For this purpose we used the 0.18um model incorporated into WATTCH with a 1.8v power supply and a 1Ghz clock. WATTCH models maximum power dissipation by estimating the overall capacitance of all non-power-supply nodes in the circuit. It then assumes a predetermined activity factor to estimate the maximum power dissipation. We followed the same methodology to estimate the maximum power dissipation of the routing network that is required by our methods. We did the same for the intra-block dependence detection logic and the output multiplexers.

We used hand-layout for the routing network and a design flow for the dependence detection logic. We used TSMC's 0.18um process to hand layout the routing network for steering register operands into the RAT and for reshuffling into position the results read out of the RAT. For the layout we used Magic v7.1 and the process files provided by MOSIS for the aforementioned TSMC process. We then extracted the capacitance for all nodes. Since our goal is to demonstrate that the RAT is the main source of power dissipation, we used a pessimistic design flow for the dependence detection logic. We wrote a verilog description of this unit and then used Synopsys' Design Analyzer along with a standard logic gate library provided by the Canadian Microelectronics Corporation to synthesize the unit. We then used Cadence's Silicon Ensemble to place and route the synthesized logic. Finally, we extracted the total capacitance for all relevant nodes and used it to estimate the maximum power dissipation. We made no attempt to power optimize either the routing network nor the dependence detection unit. Furthermore, we assumed an activity factor

of 100% for the routing network and the detecion logic.

A 24 read port and eight write port RAT for 32 architectural registers, 128 physical registers containin 24 checkpoints per bit dissipates approximately 950mW. By comparison, a RAT with 12 read ports and six write ports dissipates 487mW including 16 out-of-order checkpointing units. That is, by concentrating on the RAT alone, our methods reduce maximum power by 48%. However, this reduction comes at the cost of introducing the routing logic which, however, dissipates a lot less energy. We estimated its maximum power dissipation at 17mW, or an overhead of only 1.8%. Finally, we estimated the power of dependence detection logic and the output multiplexers at 99mW. Taking the last two results into account our methods reduce the overall rename power by approximately 42%.

The results of this section demonstrate that our methods can significantly reduce rename power. Since we used maximum power dissipation as our metric the actual reduction is independent of each program's activity. This is appropriate since we did not incorporate logic that can gate unused logic such as RAT ports and comparators. With a complete layout of the rename unit it may be possible get a more refined power estimate. Nevertheless, our results hold and do demonstrate the viability and usefulness of our approach.

8 Related Work

Several register renaming techniques even ones that combine both dynamic and static features have been proposed. In this section we restrict our attention to works that describe specific implementations, scalability/delay studies and related work on reducing rename power. To the best of our knowledge this is the first work that studies the exact renaming requirements of typical programs and suggests power-related optimizations including out-of-order control-flow resolution.

There are two primary methods for implementing renaming [2]. The first method is based on an SRAM array. It was implemented, for example, in MIPS R1000 [1]. The second method is based on a CAM array. For example, this implementation was used in the Pentium Pro [3]. In this work we focused on the SRAM-based implementation. The methods we present are also applicable with modification to the CAM implementation. An investigation of the resulting power and delay implications is beyond the scope of this paper.

Parlacharla *et al.* developed delay models for the RAT [2]. Brooks *et al.,* extended these models to include maximum power dissipation. In this work, we used the same methodology to model power dissipation. Liu and Lu suggest using a hierarchical RAT [5]. A small, first-level RAT is used to hold the mappings for the most recent renamed registers. Instructions access this RAT first and only on a miss, access the second-level conventional, full-blown RAT. Our optimizations are orthogonal since they can be applied to a hierarchical rename unit also.

Finally, Kim and Smith suggest an alternative instruction set architecture based on a hierarchical register organization [6]. They observe that most results are short-lived and suggest using accumulators that are local to each functional unit. A second level of registers is provided to hold other results. Their design eliminates some of the register operand detection logic and may lead to reduced RAT pressure. Our work is motivated by similar observations about the usage of registers in ordinary programs. However, we study its implications for existing instruction set architectures.

9 Conclusion

Existing work on power-aware techniques was naturally focused on those units that account for a larger fraction of on-chip power dissipation. In this work, we were motivated by the fact that power-density related optimizations may also be relevant as they reduce the temperature generated by hot-spot units.

As a first step towards power-density optimizations we focused on the register renaming unit. In existing designs, the register renaming exhibits among the highest power densities [7]. There are different methods for reducing power density. We observed that existing rename units are designed to handle a worst case

scenario where all instructions have two source operands and one destination register. As expected, however, in practice, this is rarely the case. Moreover, by exploiting intra-block RAW and WAW dependences it is possible to further reduce the number of register operands that need to access the register alias table. Accordingly, we proposed a simple, effective method for reducing the number of RAT ports that detects and routes only actual register operands that do not have intra-block RAW or WAW dependences. We modeled both the power and performance of our method and found that it is possible to achieve performance that is on the average within 0.5% of a conventional RAT by reducing the number of read ports from 24 to 12 and by reducing the number of write ports from eight to six. In the worst case of a multimedia application, performance was within 2.2% of the conventional RAT. We also suggested using out-of-order control-flow resolution to reduce the number of checkpoints needed to achieve high-performance. We found that a conventional in-order checkpointing mechanism with 24 checkpoints offers performance within 1.2% of that with an infinite number of checkpoints. In the worst case, however, this configuration was 5.2% slower (for mcf). Our out-of-order control-flow resolution mechanism with just 16 checkpoints offered much better performance both on the average on a per individual program basis. In particular, performance was on the average within 0.6% of the configuration with infinite checkpoints and in the worst case of mcf within 0.8%.

We implemented the routing network and dependence detection logic using a 0.18um process and estimate its maximum power consumption. We found that our methods can reduce overall rename power by 42%. Finally, we studied the impact of adding additional pipeline stages (while we argued that this may not be needed) and found that performance stays within 1.2% of the base when one more stage is introduced.

References

[1] K. C. Yeager, The MIPS R1000 Superscalar Microprocessor, IEEE MICRO, April, 1996.

[2] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. International Symposium on Computer Architecture-24*, June 1997.

[3] R. P. Colwell and R. L. Steck, A 0.6um BiCMOS Processor with Dynamic Execution, In *Proc. International Solid State Circuits Conference*, Feb. 1995.

[4] D. Brooks, V. Tiwari M. Martonosi Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, *Proc of the 27th Int'l Symp. on Computer Architecture*, 2000.

[5] T. Liu and Shih-Lien Lu, Performance Improvement with Circuit-Level Speculation, in *Proc. 33rd Int'l Symposium on Microarchitecture*, Dec. 2000.

[6] Ho-Seop Kim and J. Smith, An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing, *Proc. of the 29th Int'l Symp. on Computer Architecture*, May, 2002.

[7] Personal Communication, IA-32 Development Group, Intel Corp.

[8] P6 Power Data, Slides provided by Intel Corp. to Universities.

[9] D. Burger and T. Austin, *The Simplescalar Simulation Environment*, Univ. of Wisconsin-Madison, Computer Sciences Dept. Technical Report.