Balancing Resource Utilization to Mitigate Power Density in Processor Pipelines

Michael D. Powell, Ethan Schuchman and T. N. Vijaykumar

School of Electrical and Computer Engineering, Purdue University
{mdpowell, erys, vijay}@purdue.edu

Abstract

Power density is a growing problem in high-performance processors in which small, high-activity resources overheat. Two categories of techniques, temporal and spatial, can address power density in a processor. Temporal solutions slow computation and heating either through frequency and voltage scaling or through stopping computation long enough to allow the processor to cool; both degrade performance. Spatial solutions reduce heat by moving computation from a hot resource to an alternate resource (e.g., a spare ALU) to allow cooling. Spatial solutions are appealing because they have negligible impact on performance, but they require availability of spatial slack in the form of spare or underutilized resource copies. Previous work focusing on spatial slack within a pipeline has proposed adding extra resource copies to the pipeline, which adds substantial complexity because the resources that overheat, issue logic, register files, and ALUs, are the resources in some of the tightest critical paths in the pipeline. Previous work has not considered exploiting the spatial slack already existing within pipeline resource copies. Utilization can be quite asymmetric across resource copies, leaving some copies substantially cooler than others.

We observe that asymmetric utilization within copies of three key back-end resources, the issue queue, register files, and ALUs, creates spatial slack opportunities. By balancing asymmetry in their utilization, we can reduce power density. Scheduling policies for these resources were designed for maximum simplicity before power density was a concern; our challenge is to address asymmetric heating while keeping the pipeline simple. Balancing asymmetric utilization reduces the need for other performance-degrading temporal power-density techniques. While our techniques do not obviate temporal techniques in high-resource-utilization applications, we greatly reduce their use, improving overall performance.

1 Introduction

Power density is a growing problem in high-performance processors in which small, high-activity resources such as functional units overheat. Power density increases with technology generations as scaling of clock speed, processor current, and device area further exceeds the ability of affordable microprocessor packages to dissipate heat away from the hot spots.

Two categories of techniques, temporal and spatial, can address power density in a processor. Temporal solutions slow computation and heating either at fine-granularity through frequency and voltage scaling [16] or at coarse-granularity through stopping computation long enough to allow the processor to cool before resuming at full speed [10]. The slowing down or stopping results in performance degradation. Spatial solutions reduce heat by moving computation from a hot resource to an alternate resource copy (e.g., a spare ALU) to allow cooling. Spatial solutions are appealing because they have negligible impact on performance, but they require availability of *spatial slack* in the form of spare or underutilized resource copies. Previous work focusing on spatial slack has either proposed adding extra resource copies to a pipeline [16, 11] or targeted chip multiprocessors (CMPs) without addressing power density within an individual core [14]. Unfortunately, adding extra resource copies usually increases design complexity and critical-path delay.

Previous work has not considered exploiting the spatial slack *already existing* within the resource copies of a processor pipeline. Our key observation is that in modern processors not only is there resource underutilization, but utilization can be quite asymmetric, leaving some copies substantially cooler than others. For example, a processor with four ALUs will have one ALU that is used much more often than the others. There are two key reasons for this asymmetric utilization. (1) Processor issue width is chosen for bursty issue of many instructions to achieve high performance, but in most cycles at most one or two instructions are available for issue. (2) To achieve design simplicity, hardware schedulers statically prioritize resource copies such that even though only one or two instructions may execute, the same copies are used again and again while others remain mostly idle. It may seem that asymmetric utilization would not result in substantially asymmetric heating because the copies are adjacent. In reality, these overutilized copies become substantially hotter than their underutilized neighbors because heat conducts much more vertically to the heat sink than laterally to adjacent copies [16]. In our example of four ALUs where one is hotter than the others, evenly utilizing all four ALUs distributes the power effectively over four times the area. Such asymmetry causes hotspots, necessitating the use of performance-degrading temporal techniques. Previous work [16, 14] has not detected this heating asymmetry because aggregated resource copies (e.g. all ALUs) were modeled as a single thermal block and not individually.

The issue queue, ALUs, and register files are the source of most overheating in modern processors [5, 11, 17]. We propose to balance the asymmetric utilization within these resources to reduce power density. Scheduling policies for these resources were designed for maximum simplicity in technologies where power density was not a concern; our challenge is to balance

asymmetric utilization while keeping the pipeline simple and minimally impacting processor cycle time. Previous work [16, 11] has not addressed this challenge of implementation simplicity in power-density techniques. While balancing asymmetry is a common goal for all the three resources, fundamental differences in how the resources are structured dictate different techniques for each resource. For example, the processor may continue operating using other ALUs if some ALUs are overheated, but not if any part of the issue queue is overheated. The contributions of this paper are our techniques for balancing utilization of the three resources.

Modern superscalar processors use compacting issue queues which statically assign priorities to queue entries: the head contains older, high-priority instructions and the tail contains newer instructions. When lower-priority instructions issue, compaction logic defragments the resulting empty spaces, resulting in high-energy reads and writes to these entries. Entries near the head of the queue undergo compaction only if one of their instructions issues, but entries near the tail of the queue undergo compaction when *any* instruction issues. Because of these priority policies, compaction occurs most frequently in the low-priority tail-region queue entries, creating an asymmetry in utilization. To balance this asymmetry, we divide the issue queue into two halves, and we toggle the head and tail between the halves when a substantial temperature difference builds between them. We use a detailed model of issue and compaction logic to show that this technique has minimal impact on logic complexity. This *activity-toggling* issue queue is our first contribution.

In contrast to the issue queue which is a single monolithic resource, modern processors have multiple copies of ALUs allowing more flexible exploitation of spatial slack. Processors can issue instructions to any of these ALUs, but to keep instruction select and map simple, ALUs are statically prioritized such that high-priority ALUs are used again and again even when low-priority ALUs are idle. This static priority policy results in asymmetric utilization across the ALU copies. Ideally, we would like to balance perfectly ALU utilization using round-robin priority, but this priority scheme would add substantial complexity to instruction mapping logic. Instead, as a simple alternative to round-robin, we shut down overheated ALUs by marking them as busy, forcing select and map to choose among the underutilized ALUs while the overheated ALUs cool. This *fine-grain turnoff* policy allows the other ALUs to execute instructions while some cool, in contrast to conventional temporal techniques which shut down the *entire processor* if even a single ALU (or some other resource copy) overheats. Marking resource copies as busy does not affect the critical mapping logic and minimally degrades performance. Fine-grain turnoff is our second contribution.

Processors employ register-file copies to achieve low latency and high bandwidth; read ports of a register-file copy are wired to ALUs, creating a static mapping between ports and ALUs. Because each copy is mapped to multiple ALUs, there exists a many-to-one mapping between copies (not ports) and ALUs, giving rise to two utilization symmetries: one for ports within each register-file copy, and the other across register-file copies. If this mapping were one-to-one then only the second utilization symmetry would exist and our ALU techniques would suffice for the register file. However, the many-to-one nature requires achieving both of these utilization symmetries. One easy-to-implement option for achieving utilization symmetry across copies is *balanced mapping*, which interleaves high- and low-priority ALUs to individual register-file copies (e.g., priority 1 and 3 to one copy, and priority 2 and 4 to another). Balanced mapping slows overheating of any copy by spreading the utilization among all copies, and seems like a good solution. In addition, fine-grain turnoff can be employed for register-file copies, similar to ALUs, to allow continued processor operation as long as not all register-file copies are overheated. Fine-grain turnoff for register-file copies may be implemented simply by marking busy the ALUs mapped to an overheated copy.

However, combining balanced mapping and fine-grain turnoff creates an unexpected inefficiency in port usage because neither technique achieves utilization symmetry within a copy. Consequently, we advocate a counter-intuitive strategy of *priority mapping*, which maps all high-priority ALUs to one copy and all low-priority ALUs to another copy (e.g., priority 1 and 2 to one copy, and priority 3 and 4 to another), to be used with fine-grain turnoff. Under priority mapping combined with fine-grain turnoff, only one copy is utilized heavily until it overheats at which point other copies are used while the first one cools. Thus, the combination achieves utilization symmetry both across and within copies; this combination is our third contribution. While balanced mapping heats each copy more slowly than priority mapping, balanced mapping uses ports less efficiently. We find that the slower rate of heating is offset by the lower port-usage efficiency.

While our techniques do not obviate the need for temporal techniques in high-resource-utilization applications, our techniques greatly reduce their use, improving overall performance.

The main contributions of this paper are:

- We identify that proven techniques of compacting issue queues and static priority in ALUs and register-file ports, which have been habitually used for generations due to their overwhelming simplicity, interact unfavorably with the emerging problem of power density.
- In an issue-queue constrained processor, activity toggling in the issue queue improves performance by an average of 14% in applications constrained by issue queue and 9% overall.
- In an ALU-constrained processor, fine-grain turnoff improves performance by an average of 40%.
- In a register-file constrained processor, fine-grain turnoff and priority mapping improve performance by an average of 17% over priority mapping without fine-grain turnoff and 7% over balanced mapping without fine-grain turnoff.

The rest of this paper is organized as follows. Section 2 describes balancing asymmetric utilization. Section 3 discusses our experimental methodology. Section 4 presents our results, and Section 5 discusses related work. We conclude in Section 6.

2 Balancing Resource Utilization

In this section we describe the details of techniques that

exploit spatial slack within microarchitectural resources to reduce the occurrence of hotspots. We address intra-resource hotspots in the following microarchitectural resources: issue queue, ALUs, and register-file copies. For each resource we first discuss why there is an asymmetric distribution of activity within the resource or its resource copies and therefore why there is spatial slack. We then show how the distribution can be evened out to utilize the spatial slack by applying variations to the priority schemes that do not significantly increase complexity or area.

2.1 Issue Queue

Compaction in the issue queue is frequently identified as one of the largest consumers of energy in the processor [9]. The purpose of the compaction process is to maintain a priority order of un-issued instructions: older ready instructions should be issued first. Compaction allows for the critical select logic to be simple. Without compaction, select must determine which instructions in an un-ordered list are highest priority. With compaction, priority is determined simply by position in the queue.

Unfortunately compaction is not a symmetric process. When the instruction at the head of the queue issues and is marked invalid, every instruction in the queue must be compacted down one entry, assuming head is at the bottom and tail is at the top as shown in Figure 1. If the instruction at the tail of the queue is issued, no compaction is necessary. In other words, only instructions newer than the oldest instruction issued must be updated. This policy results in entries at the tail of the queue compacting after every issue, while entries at the head remain idle for a large fraction of the time.

To understand why this asymmetric compaction behavior leads to asymmetric power consumption and therefore asymmetric power density we describe typical compaction hardware as described in [8]. Figure 1 shows a simplified version of the compaction logic for a 3-way issue processor. Each entry holds an instruction tag, 2 physical-register tags, ready bits for the two operands, and a valid bit for the entry. The instruction tag corresponds to an address in a payload RAM that holds the additional instruction information. The payload RAM is read only when the instruction issues [3]. The output of each queue entry feeds to higher-priority entries. Generally, in an n-way issue processor, compaction of up to n invalid entries (i.e., the full issue width) is supported per cycle. Supporting n invalid entries requires that each entry can move down (towards the head) a maximum of n positions in the queue, or more specifically, each entry must drive inputs to the n entries below it for all bits in the entry. Each entry must then choose its new value from the above valid entries. Each entry produces its own mux selects using global invalid counts determined by a global adder. Once each entry calculates its mux select, the value is driven across the width of the queue to the mux. Driving the mux selects across the width of the queue, and driving the entry contents down part of the length of the queue consume much more energy than the transistors implementing the compaction policy.

Optimizations reduce energy consumption by limiting when these long wires are charged. Two opportunities exist when an entry can avoid driving its long wires. 1) When an entry is invalid or there are no invalid entries below the current entry, the current entry need not charge its data output lines because no lower entries will compact from it. 2) When an entry is valid and there are no invalid entries below the current entry, the current entry need not charge its mux select lines because its state will not change. There is ample time to determine and perform the above clock gating because compaction does not occur immediately after an instruction is issued and marked invalid. Instructions must remain in the issue queue one or more cycles after being marked invalid in case there is a L1 miss and the instruction must be replayed.

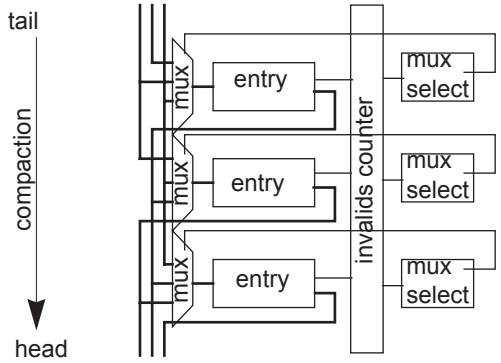

FIGURE 1: Compaction logic details

The above strategies result in the tail dissipating power on every compaction access, while the head dissipates power on only a fraction of the compaction accesses. Consequently, the head of the queue does not get as hot as the tail of the queue and there remains unexploited spatial slack.

2.1.1 Exploiting Spatial Slack in Issue Queue

We propose that this intra-resource spatial slack can be utilized by simply adjusting the position of head and tail pointers. *Activity toggling* moves the head and tail pointers to balance the activity of entries in the queue. Ideally, after migrating the head close to the hot entries, hot entries are accessed less frequently while cold entries are accessed more frequently. Moving the head and tail pointers is different from energy-saving techniques such as in [9], because those techniques only resize the queue and do not reduce activity in high-utilization regions of the queue.

Allowing for multiple positions of head and tail pointers in the queue may seem to add excessive complexity to carefully designed compaction and selection processes. As mentioned

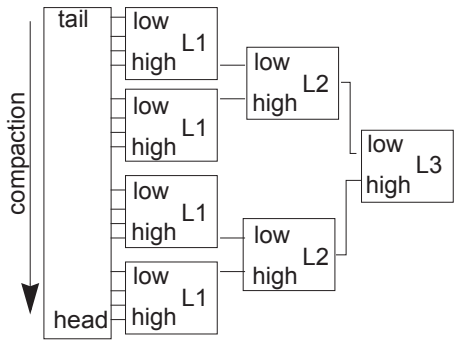

FIGURE 2: Select tree

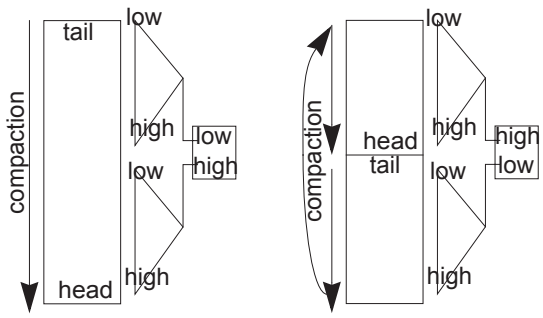

FIGURE 3: Compaction logic overview

before, the purpose of a compacting issue queue is to simplify select by encoding each instruction's priority by its location, and moving the head can break this encoding. To understand how moving the head affects the priority encoding we use Figure 2, which depicts the select tree for 1 instruction of a 16 entry issue queue and demonstrates how compaction simplifies select. In Figure 2, when an instruction is ready to issue, it raises a request bit that is sent to its L1 arbitrator. The L1 arbitrator checks all four of its inputs and if any are requesting it sends a request up the tree to its L2 arbitrator. The L2 arbitrator does the same sending a request to the root (L3) arbitrator. The L3 arbitrator is responsible for selecting only one instruction for the specific ALU hard-wired (statically mapped) to this select tree. If the ALU is ready for an instruction, the L3 arbitrator sends a grant signal back down the tree. If both its request inputs are high, the L3 arbitrator must send only one grant in priority order. In this case, the L3 node would send a grant signal to the bottom subtree because the bottom of the queue is the higher-priority head region. L2 does the same sending of a grant to the bottom-most (highest priority) L1 block that it requesting. Finally the L1 block sends a grant signal to the bottom-most (highest priority) requesting instruction. Priority can be satisfied easily at every tree node, by sending grants down to the bottom-most requesting node. The simplicity of this scheme comes from its *static* nature that the bottom-most input has the highest priority at all levels of the select tree.

In the remainder of the section we show that we can provide for another head/tail configuration, which spreads compaction heat better in the issue queue, with only simple modifications to the selection and compaction policy. In Section 4 we show that one extra compaction mode is sufficient to achieve significant performance improvements.

Good choices for the new head/tail configuration are not at first obvious. It may seem that it would be ideal to exchange the head and tail for the second configuration, but such an exchange is not realistic. Switching the head and tail would require a complete second copy of the compaction logic and wires so that instructions could be compacted in the opposite direction. In addition, every node in the tree would have to be redesigned so the high-priority end could be *dynamically* selectable between top-most and bottom-most requesting input.

Instead, we propose that the head be moved to the middle of the issue queue as depicted in Figure 3, with the tail one entry below. With this scheme the lower half of the queue holds newer instructions. Instructions still compact downward, but when they reach the bottom of the queue they wrap around and are compacted into the topmost entries of the queue. Instructions in the top half are not allowed to compact past the tail. Moving the head to the middle of the queue requires the following changes to compaction. (1) Dispatch must be able to drive instructions to the middle of the queue instead of just to the top of the queue. (2) the entries at the bottom of the queue require additional long wires to drive their contents to the top of the queue.

Maintaining the compaction direction, and placing the head in the middle of the queue requires only a minor change to the select logic. Notice that within each half, higher priority is still located at the bottom of the half. This mean that the lower select subtrees of the queue require no modifications and therefore do not increase in complexity. Only the absolute root node of the select tree which decides to grant to the top half of the queue or to the bottom half of the queue must support two modes. In the conventional head/tail configuration, the root's bottom request port is higher priority than the top request port. In the new configuration, the root's bottom request port is lower priority than the top request port.

Our scheme takes advantage of the free spatial slack by monitoring issue queue temperature and toggling compaction modes when the temperature difference between the two halves exceeds a certain threshold. We can sense temperature using on-chip temperature sensors, which [16] says are reasonable to place on-chip at resource or resource-copy granularity. In fact, POWER5 uses 24 such temperature sensors [7]. Toggling modes causes no correctness problems because priority order of instructions in the queue is not required for correctness. Immediately after a toggle, older instructions that should have higher priority may become lower priority than newer instructions. But after these older instructions issue, all instructions in the queue will stay in priority order until the next toggle. Because temperatures change slowly, at scales on the order of milliseconds, toggles are infrequent (millions of cycles) and the effect of these instructions having lower priority is negligible.

Although we move the head to combat the utilization asymmetry we are not able to guarantee moving the head will prevent overheating, because we cannot turn off completely the hot half and keep the processor running. For correctness, unless issue is completely halted, broadcast must continue to all entries and may trigger high amounts of compaction even in the head (i.e., a hot half could get even hotter). As such our technique attempts to even out the utilization and prevent a half from reaching the thermal threshold. If one does overheat we stop all issue and allow the processor to cool, which is a performance-degrading temporal technique as discussed in Section 1. In the next two subsections, we discuss resources that have independent resource copies that can be turned off entirely, unlike issue-queue halves, allowing more flexibility in utilizing spatial slack.

2.2 ALUs

While compaction leads to asymmetric utilization in the issue queue, instruction select and map leads to asymmetric utilization of the ALUs. Figure 2 shows one select tree that selects for one ALU; a superscalar has one such tree responsible for select and map for each individual ALU. Without restricting which region

of the issue queue a select tree may select from, select logic must take special precautions to ensure that multiple select trees do not select the same instruction. This requirement is handled by serializing the select trees [12]. Select happens in *static* priority order; the first tree selects, and the second masks its request signals with the grant signals of the first tree, ensuring that it can not select something already selected, and so on.

Because the select trees are constructed in a static priority order and each select tree is hard wired to a specific ALU, the ALUs are also forced into a corresponding static priority order. Consequently, if even just one instruction issues, the highest-priority ALU will always be accessed. On the other hand, the lowest-priority ALU will be accessed only in the much rarer case that the full processor width is issued. This policy results in the highest-priority ALU being accessed frequently and heating while the lowest-priority ALU is rarely accessed and stays cool. There is spatial slack in the lower-priority ALUs.

Ideally, we would like to perfectly balance ALU utilization by issuing instructions to ALUs in a round-robin order. In fact this assumption is effectively what previous research (unintentionally) modeled by treating all ALUs as one thermal block [16, 14]. However, round-robin issue is not realistic because it would require completely redesigning the select trees so they could be re-linked into many *dynamic* priority orders. Such dynamic ordering would add substantial complexity to the select trees.

Instead, we propose a much simpler solution as an alternative symmetric utilization. We propose that instead of stopping issue completely when one ALU is hot, we use fine-grain turnoff and simply stop issue to the hot ALU while exploiting the spatial slack in the remaining ALUs. Stopping issue only to the hot ALU requires informing the corresponding select tree that the ALU has overheated and that no grants should be issued from that tree. Typical select trees already support a busy signal from the ALU that prevents select. We can simply mark an ALU as busy when it has crossed the temperature threshold. When the busy signal is raised, the select tree issues no grant signals, and no requests will be masked to the lower priority select tree. Any remaining instructions will be selected by lower-priority select trees assigned to cool ALUs.

It may seem that accessing an ALU adjacent to an overheated ALU may cause the overheated ALU to get hotter, violating the purpose of the thermal threshold and causing damage. The above condition will not occur because any active ALU must be cooler than any violating inactive ALU, and heat flows only from the hotter inactive ALU to cooler, active ALUs.

Ideally overheated ALUs will cool and become active as other ALUs overheat. If that does not happen and all ALUs overheat, we resort to a temporal technique and halt issue to wait for the ALUs to cool.

2.3 Register File

Processors employ register-file copies to provide low latency and high bandwidth to the ALUs. Each ALU access requires two register reads which are provided by hard wiring the ALU to two register ports (of a copy). Because some ALUs are utilized more than others (as discussed in the previous section), some ports are utilized frequently while others are underutilized, leaving spatial

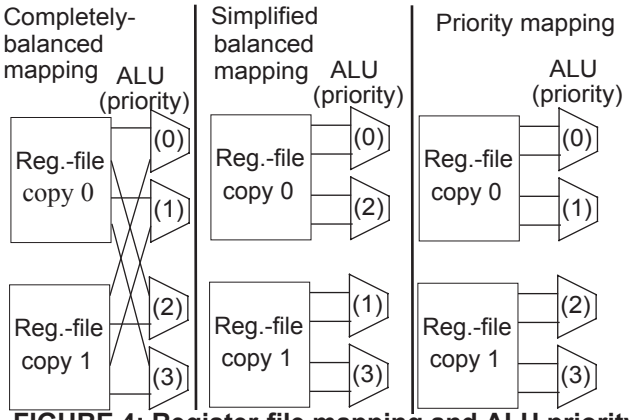

FIGURE 4: Register-file mapping and ALU priority

slack. Each register-file copy typically has (many) more than two ports, so there exists a many-to-one mapping from ALUs to register-file copies. Because of this many-to-one mapping, we are concerned with both utilization symmetry within each register-file copy (i.e., are all ports within a copy equally utilized?) and utilization symmetry across register-file copies (i.e., are all copies equally utilized?). If this mapping were one-to-one then only the second symmetry would exist and our ALU techniques would work for the register file as well. However, the many-to-one nature implies that efficient utilization of the register-file requires achieving both of these symmetries. (Register writes are inherently symmetric because all values must be written to all copies; we do not discuss register writes.)

The most-direct way to achieve symmetric utilization within and across two register-file copies (a typical number) would be to use *completely-balanced mapping* as shown in Figure 4. This mapping would ensure that one read access for every ALU went to each register copy. Unfortunately, completely-balanced mapping requires long wires between the register-file and ALU, which is undesirable because of delay and complexity.

A simpler alternative to completely-balanced mapping is *simplified balanced mapping*, also called *balanced mapping*, as shown in Figure 4. This mapping interleaves high and low-priority ALUs on each register-file copy but does not require long wires. Because balanced mapping slows overheating of any register-file copy, it achieves utilization symmetry across copies, which is the more critical symmetry if the processor must shut down when any register-file copy overheats.

If continued processor operation is allowed as long as not all register-file copies are overheated, then fine-grain turnoff for register-file copies can achieve utilization symmetry across copies, similar to ALUs. If one copy overheats then the processor can continue to use the other copy while the first one cools. Fine-grain turnoff of copies is implemented by marking busy the ALUs mapped to the overheated copy. (As in Section 2.2 and Section 2.1, if all register-file copies become hot, we halt all issue and wait for cooling.)

While it may seem that combining balanced mapping and fine-grain turnoff would result in optimal utilization symmetry, that is not the case. The key problem is that by spreading high-priority ALUs among multiple register-file copies, balanced mapping ends up overheating *all* the copies, forcing fine-grain

Table 1: Register-port mappings

Power-density	Balanced mapping	Priority mapping
conventional	symmetric across copies but not within	symmetric only within high-priority copy; not other copies
fine-grain turnoff	symmetric across copies but not within	symmetric both within and across copies

turnoff to shut down all the copies. Because each copy has multiple ports, shutting down a copy when only a few ports are overutilized results in underutilization of the copy's other ports. Shutting down many copies starves the processor of register ports even when some ports are underutilized. This unexpected inefficiency, as mentioned in Section 1, occurs because neither balanced mapping nor fine-grain turnoff target utilization symmetry within a copy.

Because fine-grain turnoff achieves utilization symmetry across copies, we replace balanced mapping by the counter-intuitive strategy of *priority mapping,* which maps all high-priority ALUs to one copy and all low-priority ALUs to another, as shown in Figure 4. Priority mapping concentrates register reads in a single copy, causing high utilization of that copy's ports (and low utilization of other copies' ports). When one copy overheats, fine-grain turnoff shuts it down and forces high utilization of the other copy and its ports. Thus, the combination of priority mapping and fine-grain turnoff achieves both symmetries. Our mapping strategies are summarized in Table 1.

Combined with fine-grain turnoff, priority mapping uses ports more efficiently than balanced mapping whereas balanced mapping heats each copy more slowly than priority mapping. However, the higher efficiency of priority mapping outweighs the slower heating of balanced mapping. Priority mapping's increased port utilization within a copy allows many more register accesses while only somewhat decreasing the heating time before a copy overheats. The heating time decrease is small because a hot copy (with high utilization) dissipates more heat per unit area than a warm copy (with low utilization), as dictated by physics. Although we consider much finer spatial granularity than [14] (register files instead of processor cores), this effect is similar to that observed in [14] which found that coscheduling carefully-chosen threads on a simultaneously-multithreaded processor (SMT) resulted in increased throughput over single-thread runs in spite of faster processor overheating.

Fine-grain turnoff causes a problem for register writes because an overheated copy may become stale. When the overheated copy cools it must contain correct register values before it can be read from. There are two simple solutions to this problem. The first is to set the thermal threshold for shutting down a copy slightly below the critical thermal threshold and allow writes to continue. Because register-files are read approximately twice as often as they are written, the cooling register file receives one-third as many accesses as normal, which is adequate to allow cooling. The second solution is to disallow writes to the overheated copy and to copy register values into the formerly-overheated copy at the end of cooling. Because cooling intervals are quite long, on the order of hundreds-of-thousands to millions of cycles, the overhead of copying register values is negligible when amortized over the cooling period.

Table 2: Processor Parameters

Out-of-order issue	6 instructions/cycle
Active list	128 entries (64-entry LSQ)
Issue queue	32-entries each Int and FP
Caches	64KB 4-way 2-cycle L1s (2 ports); 2M 8-way unified L2
Memory	250 cycles
Heatsink thickness	6.9 mm
Convection resistance	0.8 K/W
Thermal cooling time	10 ms
Maximum temperature	358 K
Frequency (GHz), voltage, and technology	4.2; 1.2V; 90nm

3 Methodology

In this section we discuss our simulation environment, design parameters, and benchmarks. We use SimpleScalar 3.0b [4] and Wattch [2] to execute the Alpha ISA. We use Wattch's aggressive clock gating to avoid unnecessary power dissipation. We use the HotSpot [16] model to extend our environment for thermal simulation, sensing temperature at 100,000 cycle intervals, substantially less than the thermal time constant of any resource, which is on the order of ms. HotSpot models both vertical and lateral heat conduction of all components. Register file copies and ALUs are turned off when they reach the maximum temperature. We toggle the issue queue policy whenever one half is hotter than the other half by more than .5 degree (before either half overheats). If any resource overheats beyond control of our techniques, we stall the processor and allow it to cool for the *thermal cooling time*, which is based on the thermal time-constant of the package. This temporal technique is similar to that used in the Pentium 4 [10]. We use a relatively high maximum temperature of 358 K. A lower temperature threshold would heat up faster making our techniques more important. So our results are conservative. Our processor parameters are listed in Table 2. Note that floating point ALUs do not represent free spatial slack in integer programs because floating ALUs can not be used for integer programs (and vice-versa). Also note that 6 integer ALUs includes arithmetic, load/store, and branch units and therefore does not provide free spatial slack for us.

We run 22 of the 26 SPEC2000 [18] benchmarks, fast-forward to the early-simpoint specified by [13], and then run 500 million instructions instead of 100 million instructions. (100 million instructions is not long enough to simulate thermal heating and cooling.) We omit four benchmarks due to long run time. We warm-up the L2 cache for the last 1-billion instructions of fast-forward.

To observe intra-resource power density variation we modify the simulator to account more accurately for energy consumption in the issue queue, ALUs and register-file copies. The following two subsections describe the circuit and floorplan

Table 3: Issue energy by component (nJ)

Compact (entry-to-entry) (per entry)	.0123
Compact (Mux select) (per entry)	.0023
Long Compaction (per entry)	.0687
Counter Stage 1 (per entry)	.0011
Counter Stage 2 (per entry)	.0021
Clock Gating Logic (entire queue)	.0015
Tag Broadcast/Match (per broadcast)	.0450
Payload RAM Access (per inst.)	.0675
Select Access (per inst.)	.0051

modifications.

3.1 Circuit Model

We modify the base simulator to model two compacting issue queues (integer and floating-point), each similar to [8, 3, 12]. Table 3 lists the power components of our issue queue model. Counter stages 1 and 2 are dynamic logic including adders and muxes as described in [8]. We assume that counter stage 1 and counter stage 2 can be selectively clock gated per entry, as described in Section 2.1. Clock gating is determined by clock-gating logic which consumes energy every cycle. We also model the wires used during compaction including entry-to-entry data wires and cross-queue mux-select wires. The entry-to-entry data wires run from each entry down to the next n higher-priority entries in an n-way issue processor. The cross-queue mux-select wires run the width of the queue and select which of the above entries should replace the current entry during compaction. Both sets of wires dissipate power only when compaction occurs. We assume that the queue entries are static memory elements and do not need to be refreshed when no compaction occurs. For power/temperature measurements we sample the power of each half of the queue (head and tail). We also model the payload RAM which is a small RAM that holds information about each instruction currently in the queue. It is written when the instruction is inserted in the queue, and read when an instruction is executed. We assume this RAM is distributed across the area of the two halves and its power dissipation likewise distributed evenly among the two segments. Similarly, we distribute the power consumed by tag broadcast, match and select to both halves of the queue because they are global queue operations. Finally when the issue queue toggles and the head moves to the middle of the queue, compaction must wrap around from one end of the queue to the other. We charge additional power (long compaction in Table 3) to each entry that must drive its data across the length of the queue. This additional power puts our activity-toggling issue queue at a power-density disadvantage when these wires must be used.

Changes to the register file are minor. We model two adjacent copies reducing the number of read ports in each by half but maintaining the number of write ports. We do not adjust the floating point register file because in Alpha, copies are not used.

3.2 Floorplan Model

We base our floorplan model on the Alpha EV6 model provided with HotSpot and scaled to 90 nm. As mentioned previously we account separately for the power of each half of the issue queue. To derive a corresponding floorplan, we divide the area of each queue into two equal parts, one representing each half. Similarly, we divide the integer register file area into 2 equal components, each representing one copy. Finally we divided the IntExec area by the 6 integer ALUs, and the FPAdd area by the 4 floating point adders in our simulated processor.

Recall that we are providing techniques for 3 different resources. Each technique targets a different resource that can be a thermal bottleneck. Architectures have different thermal bottlenecks depending on floorplan and circuit-level implementation details that are not readily available. In the Power4, the issue queue is the thermal bottleneck [5] while in the Alpha (and default HotSpot floorplan) the register file is hottest [17]. ALUs also may be a thermal bottleneck [11]. Because we cannot model this large diversity of floorplans and circuit-level implementations, we follow a simpler methodology that makes slight floorplan modifications to simulate different bottlenecks. For each of these three resources, issue queue, ALU, and register file, we scale its area such that it becomes the hottest resource for the peak-utilization applications. We fill-in the remaining area by enlarging another nearby resource. We scale area instead of power to keep the total chip power constant and ensure a fair comparison to the baseline. Figure 5a, b and c show the resulting floorplans. We believe that in ideal designs, the hottest resource would be allowed to approach the thermal threshold at steady state but should cross it only occasionally. Our scaling scheme reflects this idea well.

4 Results

In this section we present our experimental results for three different CPU models each constrained by power density of different backend resource. Section 4.1 presents results for activity toggling applied to a processor constrained by power density in the floating-point and integer issue queues. Section 4.2 discusses performance of fine-grain turnoff applied to a processor design constrained by power density in the ALUs. Finally, Section 4.3 discusses performance of fine-grain turnoff, balanced mapping and priority mapping applied to a design constrained by power density in the register file. We do not show results combining techniques for different resources because most floorplans have a single critical thermal bottleneck; however it would be possible to combine our techniques.

4.1 Issue Queue: Activity Toggling

In this section we present results for our activity-toggling scheme when applied to a CPU constrained by high power density in the issue queue. We apply activity toggling to both the integer and floating-point issue queues. We expect activity toggling to balance temperature differences between the issue-queue halves, reducing the need to shut-down the processor and increasing performance.

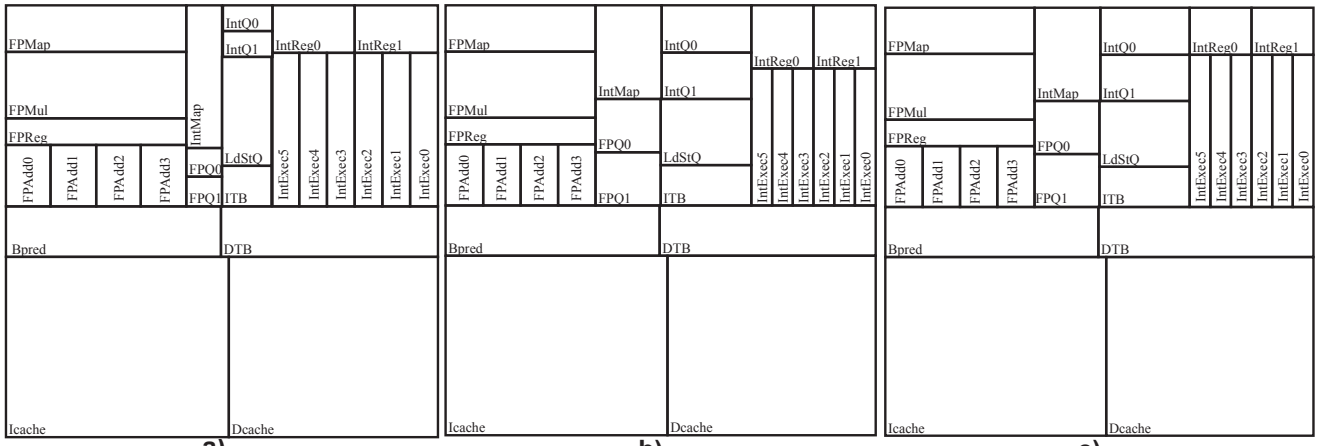

FIGURE 5: Floorplans constrained by power density in a) issue queue, b) ALUs, and c) register-file copies

To examine how activity-toggling effects issue-queue temperature we show integer issue-queue head and tail temperatures averaged across the execution time (non-overheated time) of three representative benchmarks in Table 4. The IPCs of these benchmarks are included in Figure 6 which is discussed later. *Mesa* has a 19% speedup with activity toggling, while both *facerec* and *art* have no speedup. The tables shows that for all three benchmarks, activity-toggling effectively distributes heat evenly over the two queue halves. *Art* simply never causes the issue queue to overheat, and therefore redistributing heat has no effect on performance. *Facerec* on the other hand overheats just as frequently as the base design even though activity-toggling does a good job of equalizing the two halves' temperatures. Some benchmarks such as *facerec* have high-IPC bursts of activity that cause overheating regardless of temperature balance. For other benchmarks, including *mesa*, evenly distributing heating effectively reduces processor shutdowns and produces significant speedup.

Figure 6 shows the IPC with activity-toggling (black bars) and the baseline without activity-toggling (white bars) for all of our benchmarks. Of the 22 benchmarks we simulate, 13 show speedup with activity toggling. Those that show no improvement are not limited by power density in either the integer or floating-point issue queue. Of the benchmarks constrained by issue-queue power density, *eon* shows the largest speed up of 25%. The smallest positive speedup occurs for *wupwise*, *apsi* and *gcc* each at roughly 8%. The average speedup over all benchmarks is 9%. Average speedup over just benchmarks constrained by issue-queue temperature is 14%.

As mentioned in Section 2.1 toggling is infrequent, so performance is not impacted by transiently incorrect instruction priorities in the issue queue after a head/tail swap. There are 42 head/tail swaps over the 500 million instructions executed in *eon*, corresponding to an average of 12 *million* instructions between toggles. *Bzip* toggled the most with 44 toggles and *applu* toggled the least with only 8 toggles. Frequency of toggling does not correspond to performance improvement from activity toggling; *facerec* toggles 17 times but does not speed up.

Table 4: Average temp. of issue-queue halves

Benchmark	Technique	Tail (K)	Head (K)
art	Activity-toggling	352.7	352.7
	Base	353.1	352.3
facerec	Activity-toggling	354.8	354.8
	Base	355.3	354.2
mesa	Activity-toggling	355.3	355.3
	Base	355.4	354.0

4.2 ALUs: Fine-grain turnoff

In this section we present results for fine-grain turnoff in an ALU-power-density constrained design. We apply fine-grain turnoff to both integer and floating-point ALUs. We expect fine-grain turnoff to balance ALU temperature across high-priority and low-priority ALUs almost as well as the ideal round-robin scheme, resulting in performance improvement over the base design. Our round-robin results issue instructions to ALUs in continuous round-robin priority to spread evenly accesses across all ALUs and allow fine-grain turnoff of any overheated ALU; round robin provides an upper bound on performance. As discussed in Section 2.2, round-robin would require much greater complexity than fine-grain turnoff alone.

Table 5 shows IPC and average integer-ALU temperatures of two representative integer benchmarks: *parser* which is not constrained by ALU heat and *perlbmk* which is. Parser shows no difference in IPC or ALU temperatures with or without fine-

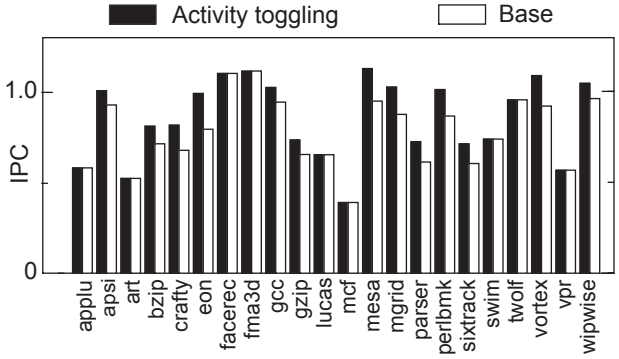

FIGURE 6: Issue-queue constrained IPC with and without activity-toggling

grain turnoff because no ALUs ever overheat. Despite not overheating we do see significant variation in the temperatures across the ALUs. The hottest ALU is over 4 degrees hotter than the coldest even though the ALUs are in close proximity to each other on the floorplan. This temperature difference is a result of better vertical heat conduction (away from the processor) than lateral heat conduction (from one ALU to the next) as mentioned in Section 1.

Perlbmk with fine-grain turnoff also shows high temperature differences between the hottest and coldest ALU, but ALU0 through ALU3 have elevated temperatures. ALU0 and ALU1 are almost at the thermal threshold meaning they likely frequently overheat and require turnoff. ALU2 and ALU3 have high temperatures because they are taking over execution of instructions that would have gone to ALU0 and ALU1. ALU4 and ALU5 remain cool, allowing the processor with fine-grain turnoff to support issue of 4 instructions even if ALU0 and ALU1 are turned off. In *perlbmk*, the baseline behavior is much different than fine-grain turnoff. For baseline, the hottest ALU is much colder than the hottest ALU in fine-grain turnoff because baseline has to stall the processor and cool whenever ALU0 reaches the temperature threshold. Fine-grain turnoff does not need to stall, and can tolerate an overheated ALU0. Moving utilization to underutilized resource copies when one copy overheats allows fine-grain turnoff to exploit more spatial slack and achieve more performance before reverting to stalling.

Both *parser* and *perlbmk* show constant low temperatures across all ALUs with round-robin, but it is interesting that equal temperature across all ALUs is not critical to achieving high IPC. In *perlbmk*, fine-grain turnoff has uneven temperatures across its ALUs and two extremely hot ALUs, while round-robin maintains evenly low temperature across all ALUs. Yet fine-grain turnoff achieves high performance. The critical aspect is preventing the whole processor from overheating and stalling, and fine-grain turnoff is as effective as round-robin. The only drawback to fine-grain turnoff's two hot ALUs is the possibility of limited issue bandwidth (because the ALUs are marked busy) in some cycles, while round-robin always has all resources available. Because ILP to sustain such high issue bandwidth over long periods is rare, IPC of fine-grain turnoff and round-robin are similar.

Figure 7 shows the IPC with fine-grain turnoff (black bars), the baseline without fine-grain turnoff (white bars), and with an ideal round-robin issue policy (hatched bars) for all of our benchmarks. Despite being much simpler, fine-grain turnoff

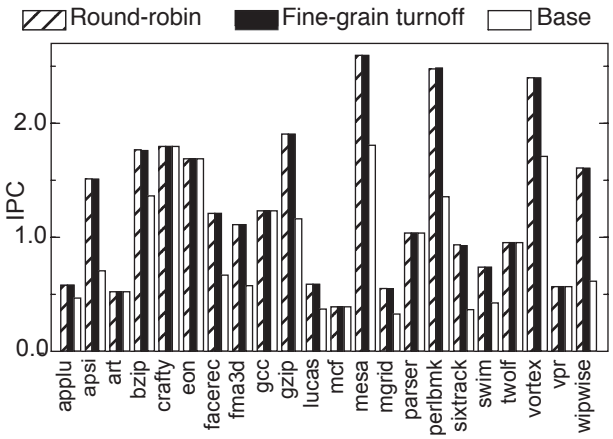

FIGURE 7: ALU-constrained IPC

approaches the IPC of round-robin to within 1%. Round robin does slightly better because it is often able to prevent any ALU from overheating, while fine-grain turnoff has to turnoff the higher priority ALUs when they overheat. Fine-grain turnoff shows significant speedup compared to the baseline that always issues to ALUs in a fixed priority order and cannot turn off individual ALUs. Fine-grain turnoff achieves average speedup of 40% across all benchmarks and 74% across only those benchmarks that are constrained by the ALU power density.

4.3 Register File: Fine-grain Turnoff and Priority Mapping

In this section we discuss results for fine-grain turnoff and port mapping in a register-file constrained design. We apply our techniques only to the integer register file because our floating-point register file does not employ copies. Without fine-grain turnoff, we expect balanced mapping to outperform priority mapping because balanced mapping at least balances register-file copy utilization (i.e., achieves symmetry across copies). With fine-grain turnoff, we expect priority mapping to outperform balanced-mapping because the combination of fine-grain turnoff and priority mapping balances port utilization within copies and register-file copy utilization (i.e., achieves symmetry within and across copies).

Table 6 shows IPC and the temperatures of the register-file copies for a representative benchmark, *eon,* for all 4 combinations described above. (Recall that the symmetry characteristics of these configurations are in Table 1.) The temperatures show that balanced mapping effectively balances heating across cop-

Table 5: Average integer ALU temperatures using different techniques

Benchmark	Technique	IPC	ALU0 (K)	ALU1 (K)	ALU2 (K)	ALU3 (K)	ALU4 (K)	ALU5 (K)
parser	Round robin (ideal)	1.0	352.6	352.6	352.6	352.6	352.6	352.6
	Fine-grain turnoff	1.0	354.8	353.8	353.2	352.4	351.0	350.4
	Base	1.0	354.8	353.8	353.2	352.4	351.0	350.4
perlbmk	Round robin (ideal)	2.5	355.0	355.0	355.0	355.0	355.0	355.0
	Fine-grain turnoff	2.5	357.2	357.0	356.5	355.2	353.0	351.2
	Base	1.4	355.4	354.9	353.8	352.8	351.3	350.4

high priority → low priority

Table 6: Average register-file copy temp. for eon

Technique	IPC	Copy 0 (K)	Copy 1 (K)
Priority-mapping + fine-grain turnoff	1.2	357.1	356.3
Balanced-mapping + fine-grain turnoff	1.1	357.2	356.9
Balanced-mapping only	0.9	357.7	357.3
Priority-mapping only	0.8	357.1	356.3

ies both with fine-grain turnoff and without fine-grain turnoff. The remaining temperature difference is due to using simplified balanced mapping instead of completely-balanced mapping as described in Section 2.3.

Because balanced mapping has more equal temperatures across copies, it is less likely to have to turn off a register-file copy. In *eon* balanced mapping with fine-grained turnoff turns off at least one register-file copy 330 times while priority-mapping turns off of at least one copy 950 times. Despite having three times more turnoffs, priority mapping achieves higher IPC, indicating better overall utilization of the register file as discussed in Section 1. It is also important to note that the temperature difference is greater with fine-grain turnoff and priority mapping than with fine-grain turnoff and balanced mapping because of increased utilization; this difference is not a problem because fine-grain turnoff switches to utilizing copy 1 if copy 0 overheats.

Figure 8 shows IPC for four configurations: fine-grain turnoff with balanced mapping (black bars), fine-grain turnoff with priority mapping (white bars), only balanced mapping (hatched bars), and only priority mapping (gray bars). Comparing IPCs of the two sets that do not have fine-grain turn off (processor must be halted if either copy becomes too hot), we see significant speedup with balanced mapping compared to priority mapping. As described in Section 2.3, balanced mapping forces symmetry in accesses across copies so it takes longer for either copy to overheat, allowing for better performance. Across all benchmarks the average speedup was 9% and across only benchmarks constrained by the register-file temperature, the speedup was 14%. Balanced mapping effectively reduces performance degradation due to register-file overheating if fine-grain turnoff is not available.

We next consider IPC improvement when fine-grain turnoff of register-file copies is available. Adding fine-grain turnoff to a priority-mapped register file improves IPC over a priority-mapped register file without fine-grain turnoff by 17% on average across all benchmarks and 30% across register-file constrained benchmarks. Fine-grain turnoff with priority mapping outperforms balanced mapping without fine-grain turnoff by 7% over all benchmarks and 14% over register-file constrained benchmarks. As discussed in Section 2.3 priority mapping plus fine-grain turnoff is the best among the 4 combinations. Combining fine-grain turnoff and balanced mapping results in lower IPC than fine-grain turnoff plus priority mapping because that combination fails to address utilization symmetry within copies. Across all benchmarks, fine-grain turnoff with priority mapping produces a 1.8% higher average IPC than fine-grain turnoff with balanced mapping. Across those benchmarks constrained by the register file, fine-grain turnoff with priority mapping produces a 3.1% higher average IPC than fine-grain turnoff with balanced mapping. *Eon* and *wupwise* show the largest speedups, 8%, from priority mapping.

5 Related work

Several previous proposals address thermal management or power density in superscalars. First we list the temporal techniques which are orthogonal to our spatial technique. These temporal techniques can all be applied if our spatial techniques are unable to prevent overheating, but they result in various amounts of performance degradation. [1] evaluates techniques to balance the rate of heat production to heat dissipation at chip-level granularity. [15] proposes fetch-throttling techniques and use of PID controllers to manage power density. [16] introduces the HotSpot temperature model used in this paper and proposes PID-controlled dynamic frequency and voltage scaling. Frequency-scaling is a temporal technique with similar perfor-

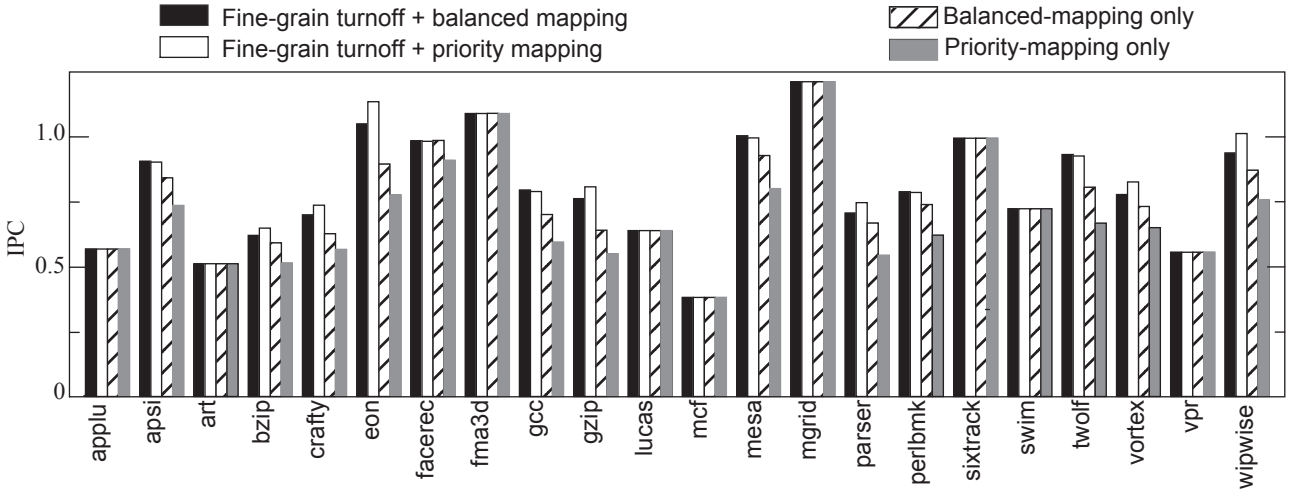

FIGURE 8: IPC of register-file constrained CPU with and without fine-grain turnoff with both balanced and priority mapping

mance-degradation as the Pentium 4's [10] thermal technique compared against in this paper. [16] also proposed dynamic-voltage scaling (DVS) and temperature-tracking frequency-scaling (TTDFS) as control mechanisms. DVS may not be practical in future circuit technologies as nominal supply voltages approach transistor threshold voltages, leaving little room for additional voltage adjustment for power-savings. Temperature-Tracking Frequency Scaling (TTDFS) allows the processor to heat above its "maximum" temperature by slowing the clock and relaxing timing constraints. As stated in [16] TTDFS is effective only if the sole limitation on power density is circuit timing. TTDFS does not reduce maximum temperature or prevent physical overheating and cannot handle large increases in temperature, which may damage the chip.

A few previous proposals have discussed spatial techniques. [16] proposes duplicating the integer register file, which would add substantial wiring and select-logic complexity. [11] proposes "ping-ponging" resource activity for various pipeline resources between duplicate resource copies within a superscalar core but does not address the scheduling-logic or wiring implications of providing these duplicates. [14] proposes a spatial technique that migrates jobs among cores in a simultaneously-multithreaded chip-multiprocessors (SMT CMPs), but does not address hotspots within the cores. [6] proposes another spatial technique that uses a clustered architecture with many complex clusters (each containing distributed rename logic, issue queues, ALUs, a register file, and distributed commit logic). Like [14], [6] does not address hotspots internal to the clusters.

6 Conclusions

We have identified that proven techniques of compacting issue queues and static priority in ALUs and register-file ports, which have been habitually used for generations due to their overwhelming simplicity, actually limit performance by causing asymmetric utilization. We have proposed three techniques, one for each of the issue queue, ALUs, and register file copies. These techniques improve performance by balancing utilization symmetrically across resource copies while keeping the implementation simple.

We show that symmetric utilization can significantly improve IPC in processors that are constrained by power density. In configurations constrained by issue queue power density, our techniques produce speedups averaging 14%. In configurations constrained by ALU resources our techniques produce speedups averaging 74%, and finally, in configurations constrained by register file power density our techniques produce speedups averaging 30%.

References

[1] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Seventh International Symposium on High Performance Computer Architecture (HPCA)*, pages 171–182, Jan. 2001.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[3] M. D. Brown, J. Stark, and Y. N. Patt. Select-free instruction scheduling logic. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO 34)*, pages 204–213, Dec. 2001.

[4] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[5] A. Buyuktosunoglu, A. El-Moursy, and D. H. Albonesi. An oldest-first selection logic implementation for non-compacting issue queues. In *15th International ASIC/SOC Conference*, pages 31–35, Sept. 2002.

[6] P. Chaparro, G. Magklis, J. Gonzalez, and A. Gonzalez. Distributing the frontend for temperature reduction. In *Eleventh International Symposium on High Performance Computer Architecture (HPCA)*, pages 61–70, Feb. 2005.

[7] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson. Design and implementation of the power5 microprocessor. In *Proceedings of the 41st Design Automation Conference (DAC)*, 2004.

[8] J. A. Farrell and T. C. Fisher. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, 1998.

[9] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA 28)*, pages 230–239, June 2001.

[10] S. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. In *Intel Technology Journal Q1 2001*, Q1 2001.

[11] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 217–222, Aug. 2003.

[12] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[13] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques (PACT2003)*, Sept. 2003.

[14] M. D. Powell, M. Gomaa, and T. N. Vijaykumar. Heat and run: Leveraging smt and cmp to manage power density through the operating system. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, pages 260–270, Oct. 2004.

[15] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *Eighth International Symposium on High Performance Computer Architecture (HPCA)*, pages 17–28, Feb. 2002.

[16] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA 30)*, pages 2–13, June 2003.

[17] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture: Extended discussion and results. Technical Report CS-2003-08, University of Virginia Department of Computer Science, Apr. 2003.

[18] The Standard Performance Evaluation Corporation. Spec CPU2000 suite. http://www.specbench.org/osg/cpu2000/.