Heat Stroke: Power-Density-Based Denial of Service in SMT

Jahangir Hasan Ankit Jalote T. N. Vijaykumar Carla E. Brodley[1]

School of Electrical and Computer Engineering
Purdue University
{hasanj,jalote,vijay}@ecn.purdue.edu

[1]*Department of Computer Science*
Tufts University
brodley@cs.tufts.edu

Abstract

In the past, there have been several denial-of-service (DOS) attacks which exhaust some shared resource (e.g., physical memory, process table, file descriptors, TCP connections) of the targeted machine. Though these attacks have been addressed, it is important to continue to identify and address new attacks because DOS is one of most prominent methods used to cause significant financial loss. A recent paper shows how to prevent attacks that exploit the sharing of pipeline resources (e.g., shared trace cache) in SMT to degrade the performance of normal threads. In this paper, we show that power density can be exploited in SMT to launch a novel DOS attack, called heat stroke. Heat stroke repeatedly accesses a shared resource to create a hot spot at the resource. Current solutions to hot spots inevitably involve slowing down the pipeline to let the hot spot cool down. Consequently, heat stroke slows down the entire SMT pipeline and severely degrades normal threads. We present a solution to heat stroke by identifying the thread that causes the hot spot and selectively slowing down the malicious thread while minimally affecting normal threads.

1 Introduction

When a number of users share a resource in a system, some arbitration scheme must ensure fairness among the users. While it may be straight forward to arbitrate an explicitly shared resource, system design may overlook subtle forms of resource sharing. A malicious user can launch a denial of service (DOS) by exploiting such subtle forms of sharing in order to harm other users [4]. Some well-known DOS attacks are: (1) A process forks a large number of child processes, exhausting the available entries in the process table and preventing new processes from being spawned [5]. (2) A remote machine initiates a large number of TCP connections with a server, exhausting the available entries in the server's TCB table and preventing any other machines to establish TCP connections [6].

Because a DOS attack can render a system practically inoperative, DOS attacks can be detrimental to businesses that serve a large number of users (e.g., e-commerce databases and webservers). Due to the scale of possible financial loss, it is important to identify and address these attacks.

In Simultaneous Multithreading (SMT) [15] multiple threads share pipeline resources at the same time in order to achieve high throughput. Because of its high instruction throughput and low implementation cost, SMT is being adopted by the microprocessor industry. Because multiple threads share resources in SMT, there are opportunities for a malicious thread to launch a DOS attack by abusing the shared resources. Consequently, it is important to address DOS in the context of SMT. For example, [7] describes a form of DOS attack in which a malicious process repeatedly flushes the trace cache of an SMT by executing self-modifying code. Because the trace cache is shared among all the processes, the flushing degrades the performance of all threads.

In this paper we show that power density can be exploited to launch a novel DOS attack, called *heat stroke*, in SMT. Power density in high-performance microprocessors is the problem of high power dissipation in a small area causing local hot spots in the chip. In heat stroke, a malicious thread repeatedly accesses a resource to create a hot spot at the resource. If the resource is not shared then the hot spot affects only the malicious thread and it is easy to trace the hot spot to the malicious thread and to stop fetching from the thread. However, if the resource is shared then the hot spot affects all the threads and it is harder to pinpoint the source of the problem.

Today's systems effectively protect themselves against known DOS attacks. Heat stroke, however, is a new attack and current systems are unprotected against it. If unaddressed, heat stroke can be used by attackers to launch successful DOS attacks. Therefore, we must address the heat stroke threat.

Solving power density using only packaging is hard. Power density continues to increase with technology generations as scaling of current, clock speed, and device density outpaces downscaling of supply voltages and the thermal ability of packages to dissipate heat [10]. A localized hot spot can reach emergency temperatures regardless of average or peak external package temperature or chip power; therefore techniques designed to reduce those parameters are ineffective at alleviating hot spots. Exotic technologies such as liquid cooling and immersion can improve packages, but are expensive and do not scale with technology [16].

Previous architectural solutions to power density alleviate hot spots by slowing down activity until the temperature drops to an acceptable level. The schemes slow down the

clock and lower the supply voltage as done in [12], completely stop the processor as done in [1] and in commercial processors [8], or stop activity at the hot spot and migrate and restrict activity to cooler components [9]. The schemes are based on the premise that normal programs cause only transient and not prolonged hot spots, and that the package can keep the temperature at an acceptable level for most of the execution time of normal programs. Accordingly, the schemes temporarily slow down activity to allow the hot spot to cool down and then resume to full-speed operation. Consequently, the performance degradation incurred by the schemes is acceptable. Heat stroke, however, does not behave like normal programs and causes severe and prolonged hot spots. Because the hot spots occur at the shared resources of SMT, applying the schemes to alleviate the hot spots forces the *entire* processor into a repeated cycle of heating and cooling periods. Because heat stroke can create hot spots fairly quickly (e.g., within 5-10 million cycles at 4 GHz) and cooling takes much longer (e.g., 50 million cycles), the heat-cool cycle results in severe performance degradation in all the threads.

The degradation caused by heat stroke is *neither* by monopolizing shared resources in SMT, *nor* by exploiting SMT's ICOUNT fetch policy [14]. ICOUNT attempts to maximize throughput by choosing the thread that has the fewest instructions in flight assuming that fewer instructions in flight implies less stalls and higher utilization. Consequently, if an extremely high-IPC thread is run with normal threads, the high-IPC thread gets a larger share of the pipeline than the other threads under ICOUNT. We calibrate heat stroke to cause virtually no degradation with perfect packaging that can remove any amount of heat instantaneously, and severe degradation with realistic packaging, showing that heat stroke is a general and novel attack that does not monopolize SMT's shared resources nor exploit ICOUNT in any way. Moreover, SMT-aware OS schedulers [13] cannot alleviate heat-stroke. Such schedulers address *coincidental* incompatibilities among simultaneously executing threads leading to performance degradation. However, heat-stroke is a *deliberate* malicious behavior for which the scheduler does not look out.

To address heat stroke, we propose *selective sedation* based on two key observations: (1) There is a large difference in the rates of access of the heated resource by hot-spot-creating threads and normal threads. (2) Previous hot-spot solutions slow down the *entire* pipeline degrading all the threads whereas it is *only* the hot-spot-creating thread that needs to be slowed down. The first observation implies that it is easy to differentiate between the two types of threads on the basis of resource usage. Accordingly, we monitor the per-thread usage of potential-hot-spot resources. When a resource reaches a threshold just below the emergency temperature (similar to [1]), we identify the thread with the *highest* resource usage as the culprit thread and slow the thread down. When the resource's temperature drops to normal, the thread resumes full-speed operation. Because the heating and cooling times are long, our monitoring can be slow and designed to be power- and space-efficient. By such selective throttling we avoid slowing down the entire pipeline and prevent one thread from hindering the other threads, as per our second observation.

Two important features of our solution are: (1) We do not attempt to solve the general problem of power density. Instead, we only prevent a thread with a power-density problem from degrading the performance of other threads which do not have power-density problem. (2) Regardless of whether a thread is malicious or not, if the thread has a power-density problem, it must be prevented from causing hot-spots and degrading the performance of other threads. Therefore, it is *unnecessary* to determine if a thread is actually malicious or not. Accordingly, our solution avoids making this determination and still prevents hot-spot-creating threads from affecting other threads. Note that, we are not unfair to non-malicious threads because any power-density scheme *must* stall such a thread if it has power-density problem.

Using SMT simulations, we show that (1) running a SPEC2K program with a heat-stroke thread degrades the performance of SPEC2K programs by a factor of four in a processor with realistic packaging; (2) our solution restores performance virtually entirely even in the presence of a severely malicious heat-stroke thread; and (3) our solution does not affect the performance of normal threads in the absence of heat stroke.

The rest of the paper is organized as follows. In Section 2 we describe the related work and background material. In Section 3 we describe heat stroke and our solution in detail. In Section 4 we describe our experimental methodology and in Section 5 we present the results from our experimental evaluations. Finally in Section 6 we conclude.

2 Background and Related Work

We now provide some background details on the power density problem, and DOS attacks in the context of SMT.

2.1 Power Density

The conduction of heat can be modelled via equivalent *heat circuits*, in which the voltage indicates the temperature, the flow of current represents the flow of heat, and various components have their respective thermal resistances and capacitances. The more readily a component conducts heat the smaller is its thermal resistance, and the more heat a component can absorb, per unit change in temperature, the higher is its thermal capacitance. Analogous to electrical circuits, heat circuits also have RC time constants which determine how rapidly a component may be heated up or cooled down.

When a component in a processor is accessed at a high rate, the repeated switching of transistors generates a large amount of heat. Because the thermal capacitance of a typical component tends to be small, this heat raises the tem-

perature of the component considerably, creating a local hot spot. The heat may propagate from the hot spot in two different directions. It may travel laterally across the die to neighboring components raising their temperature, or it may travel vertically out of the die to the heat sink. Much like an electrical circuit, more of the heat will flow through the path of less thermal resistance. Unfortunately the thermal resistances are such that the flow of heat in the lateral direction is not appreciable. The motivation for thermal greases and fans is to reduce the thermal resistance of the path to the heat sink, drawing the heat through that path.

Nonetheless, the rate at which heat can flow out to the heat sink is limited by the thermal resistance of that path. When a component generates heat at a rate that is larger than the rate which the path to the sink can accommodate, the temperature of the component will steadily increase. One obvious solution is to slow down the generation while maintaining a steady extraction. To that end various solutions for the power density problem temporarily either slow down or completely suspend the activity at that particular component. [8] simply halts the processor's pipeline, while [12] scale down the clock cycle and voltage, to slow down the pipeline until the hot spot has cooled down. The RC time constant of the path to the heat sink determines how much time the component will require to cool down. For a typical heat sink the cooling time is in the order of 10 ms. Once this cooling time has elapsed, activity at the component can be resumed to full speed.

2.2 DOS attacks in Simultaneous Multithreading

A typical program thread executing on a superscalar generally underutilizes the available pipeline resources for most of the execution time. Simultaneous multithreading (SMT) can boost the net throughput of the pipeline by allowing other threads to use the resources which would otherwise go unutilized. Because of its high instruction throughput and low implementation cost, SMT is being adopted by the microprocessor industry. However, because multiple threads share pipeline resources in SMT there exists opportunity for malicious threads to launch DOS attacks by abusing the shared resources.

For example, [7] describes a form of DOS attack in SMT, in which a malicious thread may repeatedly execute self-modifying code, causing the trace cache to be flushed repeatedly. Because all threads in SMT share the same trace cache, this repeated flushing degrades the performance of all threads.

Because multiple threads share resources in SMT, a power-density hot-spot at a shared resource can affect all threads. In the next section we explain how a malicious thread can exploit power density to launch a DOS attack against other threads.

3 Heat Stroke and Selective Sedation

We have provided details on the general power density problem and briefly described some previous solutions. We now describe how a malicious thread in SMT can leverage power density to carry out a DOS attack. We then propose a solution that addresses such DOS attacks.

```
L$1:
        addl $1, $2, $3
        br L$1
```

FIGURE 1: Example code of a malicious thread that causes heat stroke

3.1 Inflicting Heat Stroke

Recall that when a resource is accessed at a high rate, it generates a large amount of heat. The rate of heat dissipation towards the heat sink is limited by the thermal resistance of that path. Because a resource may generate heat at a rate higher than the capacity of that path, the temperature of the resource may steadily increase until it reaches an unacceptable level.

A malicious thread can exploit this fact to repeatedly access a resource at a high-rate over a long period of time, causing such a hot spot. In Figure 1 we show the example code for such a malicious thread. The thread has a large number of independent instructions so that all of them may execute rapidly without stalls. Because each instruction in the code accesses the register file, the thread effectively issues accesses to the register file at a high-rate. Prolonged execution of such code will lead to a hot spot at the register file. A pipeline will generally have a number of temperature sensors, one at each potential-hot-spot location. Once the temperature sensor at the register file is triggered, the pipeline must invoke some mechanism to attend to this power-density problem. Known techniques for controlling power density either slow down or completely stall the entire SMT pipeline until the hot-spot cools down.

Because the time constants involved in cooling are of the order of 10 ms, a large number of cycles are lost in this cooling-down phase. During the cooling-down phase, the performance of all threads on the SMT suffers. If the malicious thread can causes hot spots repeatedly, then repeated time-outs for cooling will be required. We observe that it takes a mildly malicious thread about 1.2 ms to heat up the register file to the emergency temperature, and each time that happens the pipeline needs 12.5 ms to cool down. Thus, with hot-spots generated back-to-back, the duty cycle of the entire pipeline degrades to $1.2/(1.2+12) = 0.09$. We show in results that such a small duty cycle can degrade the IPC of normal threads by 88%. We call this severe degradation via repeated hot-spots a case of heat stroke.

Because an SMT pipeline may fetch instructions from multiple threads at any cycle, SMT needs some arbitration mechanism to divide the fetch bandwidth per cycle. ICOUNT, the commonly used fetch policy for SMT, attempts to maximize the net IPC throughput of an SMT pipeline. ICOUNT fetches from that thread which has the fewest number of instructions in flight, assuming that fewer

```
L$1:
        addl $1, $2, $3
        br L$1

L$2:
        ldq $4, addr1
        ldq $4, addr2
        .....
        .....
        ldq $4, addr9
        br L$2
```

FIGURE 2: A moderately malicious thread

instructions means fewer stalls and higher utilization. While ICOUNT inherently prevents starvation of threads, a high-IPC thread can monopolize the fetch bandwidth and degrade the performance of other threads. Observing the code shown in Figure 1, we may suspect that the malicious thread degrades other threads by monopolizing the fetch bandwidth via its high IPC, and not via any power-density problems. To isolate the role of power-density in a DOS attack from any fetch-policy side-effects, we use the moderately malicious code shown in Figure 2. The code consists of two phases, the first phase is similar to the code shown in Figure 1, and attempts to generate a hot-spot via high rate accesses of the register-file. The second phase consists of a number of L2 cache misses (by choosing addr1 through addr9 such that they map to the same set in an 8-way cache). By adjusting the duration of each phase we can fine tune the IPC of this malicious thread to an acceptable level. Thus we ensure that any degradation of the other threads is caused by power-density problems and not by fetch-bandwidth monopolization.

3.2 Selective Sedation

Having explained how and why heat stroke occurs, we now propose a solution for heat stroke. Before proposing the actual solution we make a few key observations that will help us understand the required structure of the solution.

We make the first key observation that the access-rate behavior of threads which cause hot-spots is distinctly different from that of normal threads. This observation implies that the access-rate behavior of various threads at each resource clearly divides the threads into potential hot-spot creators and normal threads. We can monitor the access-rate behavior of the threads at each resource and use that information to identify the threads causing any power density problems.

The underlying reason for heat stroke is that a thread with a power density problem causes the entire pipeline to be slowed down. Our second key observation is that it is not necessary to slow down entire pipeline for cooling, rather it is only the problematic thread that really needs to be slowed down. We identify that the solution needs to implement a per-thread slowing down instead of a global slowing down, preventing a malicious thread from degrading the performance of other threads. However, underneath our solution, we still retain a global stop-and-go as a safety-net. By this safety-net we ensure that if, under any circumstance, the pipeline does reach an emergency temperature, we can shut it down to avoid permanent damage. We now describe selective sedation, our solution to heat stroke.

3.2.1 Identifying problematic threads

If a non-malicious thread causes hot-spots then it too will cause the pipeline to slow down, degrading the performance all other threads. Thus we do not need to distinguish between malicious and non-malicious threads. Instead, if any thread causes power density problems, we must identify it and prevent it from adversely affecting other threads. Note that we do not try to solve the general power-density problem. By addressing heat stroke, we are simply preventing a thread with a power density problem from degrading the performance of other threads which do not have a power density problem.

We maintain per-thread counters that track the access-rates of different resources. We need to track the access behavior of threads over a long period of time (e.g., 1 ms). Simply counting the total number of accesses over a long period of time does not work. A non-malicious thread may make accesses at a small but steady rate over the entire period, achieving a total count higher than that of a malicious thread which carries out a relatively short burst of aggressive-rate accesses. We found that the malicious thread from Figure 1 may critically heat up the register-file in just about 5 million cycles (at 4GHz). We must continuously monitor the threads over long periods of time in order to detect suspicious behavior. Because the time constants involved in hot-spot generation are large this tracking need not be done at a fine granularity, instead we may sample the access-rates infrequently, say by counting the number of accesses in every 1000 cycles. Obviously, it would be both space-inefficient and cumbersome for analysis if we were to store all the access-rate values periodically measured over a long duration of time. Instead, we compute a running weighted average on all the access-rate values by weighting each sample inversely proportionally to its age. At every sampling instant the average is computed as:

$$Wt.\ Avg = (1\text{-}x) * Wt.\ Avg + x * access\text{-}rate$$

Every time the access-rate is sampled, a new weighted average is computed, and the weight of each of the previous access-rate values gets diminished by a factor of (1-*x*). The parameter *x* can be tuned to adjust the memory of the weighted average. We empirically know that the time to generate a hot-spot is in the order of a million cycles (at 4 Ghz). Given that we sample the access-rate after every 1000 cycles, we need to retain memory for effectively 1000 sample points. We find that x = 1/128 suffices for such purposes.

Because the computation of the weighted average involves two multiplication operations, one may think that this computation is expensive. However, if we choose x to be a power of 2, then the multiplication operations are reduced to shift operations. We can assign x to a value of 1/128, replacing the multiplication by an 7-bit shift operation. Because the remainder of the operations are simple additions and subtractions, the computation of the weighted average becomes inexpensive. The infrastructure required to monitor access-rate behavior consists of one counter, one register and some peripheral arithmetic logic, per resource per thread. Each counter records the access-rate for one thread at a particular resource, and it is incremented every time the thread accesses that resource. The register holds the weighted average of access-rate for that thread at that particular resource, and we recompute the weighted average at every sampling interval using its current value and the value in the access-rate counter. After we read the value in the access-rate counter, we reset the counter to zero in order to begin measuring the next sample.

Because the weighted average tracks the access-rate behavior over a reasonable period of time, it is an effective metric for identifying the culprit thread when a hot-spot occurs. We see that the weighted average for threads with a power-density problem tends to be distinctly higher than that of other threads. However we also observe that typical programs, such as the SPEC2K suite, occasionally exhibit short bursts of a high weighted-average without causing any power-density problems. Hence, policing the threads via an absolute weighted-average threshold would degrade performance significantly due to false positives (i.e., threads with no power-density problems are penalized). Furthermore, raising the weighted-average threshold in order to reduce the performance degradation would enable a malicious thread to inflict heat stroke without being detected. Instead, we use a temperature-based threshold to detect suspicious behavior. When the temperature of a resource rises to near the emergency temperature we can expect that the weighted average for the culprit thread will be distinctly higher than that of the other threads. In order to avoid emergencies, we borrow from [1] and adjust the temperature sensors to trigger at a temperature slightly below the emergency temperature (e.g, 356K when the actual emergency temperature is 358.5K). We call this temperature threshold the upper-threshold. In the event of an upper-threshold trigger at any resource, we identify the culprit thread as the one with the *highest* weighted average at that resource. A temperature-based threshold rarely causes false positives because the upper-threshold is set close to the emergency temperature.

Upper-threshold triggering is similar to the idea of emergency-temperature triggers first proposed in [1]. This paper uses emergency temperature triggers to prevent the chip-wide temperature from reaching a damaging level, and not to address local hot-spot problems. We could imagine adapting the paper to address the problem of local hot-spots. However, this adaptation in itself would not solve the problem because, upon temperature emergencies, the adaptation would stall the *entire* pipeline, essentially degenerating to stop-and-go.

3.2.2 Sedating Problematic Threads

Once we have identified the culprit thread as described above, we *sedate* its execution by ceasing to fetch instructions from that thread. We observe, from the behavior of average programs such as the SPEC2K suite, that a non-malicious thread may also cause an occasional upper-threshold trigger. Recall that we do not attempt to distinguish between malicious and non-malicious threads. While the sedation of such a non-malicious thread is needed (because any power-density scheme must slow down at least that thread if not the entire pipeline), it would be detrimental to the performance of the thread if its execution were sedated indefinitely. To that end, we sedate an offending thread only for a period long enough to allow the resource to cool down. Once the culprit thread is sedated, we expect that the resource will not be accessed aggressively and will begin to cool down. We detect sufficient completion of cooling by another threshold which is set to a temperature just above that of normal operation for that resource (e.g., 355K for the integer register file, where normal operating temperature is 354K). We call this temperature threshold the lower-threshold. Thus, for the purposes of selective sedation, we associate two temperature triggers with each resource, one for the upper-threshold and one for the lower-threshold. When the sensor triggers at the lower-threshold we restore the sedated thread to normal execution. Note that during sedation, the access-rate and the weighted average of the culprit thread are not computed at all. Thus, we ensure that the period of inactivity will not artificially lower the weighted average for that thread.

So far we have assumed that there is only one thread with a power-density problem, which may not necessarily be true. If there are multiple threads with power density problems, then sedating the first culprit thread does not guarantee that the heated-up resource will actually cool down. Therefore, after the upper-threshold triggers, we wait for a duration that is twice the expected cooling time of the resource, and then reexamine the temperature of the resource. We choose twice the duration because the cooling time inherently assumes no heat generation, whereas for our purposes a thread may still be running, generating some heat. After this duration if the temperature is still above the lower-threshold, we conclude that there is another thread with a power-density problem which is still operative. As before, we identify and sedate the thread with the highest weighted-average. When the resource cools down to the lower-threshold, we resume normal execution of all threads that were sedated for that resource. As long as the resource does not cool down to the lower-threshold and there are un-sedated threads still operative, we must continue periodic reexamination. The only exception is that when there is only one un-sedated thread left, that thread cannot degrade the performance of any other thread. We

allow the last unsedated thread to continue to operate even above the upper-threshold. In the event that the thread heats up the resource to the emergency temperature, the safety-net stop-and-go mechanism intervenes. Stop-and-go stalls the entire pipeline until the resource cools down to normal operating temperature, restoring all sedated threads to normal execution.

In addition to alleviating heat-stroke in hardware, we also report the offending threads to the operating system. This reporting facilitates the identification of offensive threads and their users.

We show in our experimental evaluation that selective sedation successfully alleviates heat-stroke without causing performance loss due to false-positives.

3.3 Generality of Heat Stroke

A number of previous proposals address the issue of fairness in the context of SMT execution. [7] addresses DOS attacks based on trace-cache flushing, and [13] proposes an OS scheduler that ensures fair, priority-based CPU utilization across all the threads in an SMT machine. We argue that heat stroke is a general DOS attack which may not be alleviated by the techniques proposed in [7] and [13].

[7] addresses a specific DOS attack which is characterized by repeated execution of self-modifying code, resulting in repeated flushing of the trace-cache. A malicious thread may degrade the performance of other threads in an SMT by flushing the trace-cache repeatedly. [7] proposes a scheme that detects such behavior and notifies the OS about the offending thread. [7] detects culprit threads based on the observation that, the cache-flush pattern of normal threads is noticeably distinct from that of self-modifying code. While our scheme is similar to [7] in terms of behavior-based detection, we exploit power density while [7] exploits self-modifying code. Furthermore, we actually prevent heat-stroke attacks via selective sedation, whereas [7] only detects and reports culprit threads to the OS.

In SMT machines, multiple threads may occupy the CPU during a single quantum, yet they may make different amounts of progress during that quantum. Thus, simply keeping an account of the number of quanta that each thread runs for, does not guarantee fair, priority-based CPU usage in SMT. [13] proposes an OS scheduler that monitors the individual progress across threads within one quantum to provide this guarantee. The OS scheduler uses this information to guarantee progress proportional to priority by allocating quantums to groups of threads (to be run simultaneously) and to individual threads (to be run alone). Because the monitoring poses an overhead in system throughput, [13] first runs a monitoring phase and then allocates CPU quantums for a longer non-monitored period. [13] does not consider the possibility of malicious programs, and is designed under the premise that all programs behave in a non-malicious manner.

While it may seem that such an OS scheduler can prevent a DOS attack by guaranteeing fair CPU usage to all threads even in the event of a heat stroke, a malicious thread can defeat the scheduler by exploiting the internal details of the scheduler. (1) If a malicious thread deliberately degrades the performance of other threads, such an OS scheduler assumes that the degradation is due to coincidental incompatibility for SMT execution. The scheduler continues to execute the malicious thread, and continues to team it up with other threads in search of non-existent SMT compatibility. In contrast, selective sedation actually identifies such malicious threads and notifies the OS, so that the scheduler may mark such threads ineligible for execution. (2) While the OS scheduler guarantees fair, priority-based CPU sharing, it may do so at the cost of low CPU utilization. By launching repeated heat strokes during the evaluation phase, a malicious thread may force the scheduler to schedule other threads for solo execution (i.e., only one thread in the pipeline) on the CPU for long periods of time. Alternately, a malicious thread may assume a high priority and exhibit an artificially low IPC during specific parts of the evaluation phase, forcing the scheduler to schedule the malicious thread for solo execution over long durations. Long periods of solo execution degenerate the SMT machine to a non-SMT machine, degrading overall system utilization. (3) If the duration of the monitored and non-monitored periods are fixed then a malicious thread may easily behave as a normal thread during the monitoring periods and launch repeated heat-stroke attacks during the non-monitored periods. Whereas, if the duration of the monitored and non-monitored periods are randomized, then a malicious thread can alternate between normal and malicious behavior to achieve probabilistic DOS. Using more than one malicious thread, the probabilistic DOS attack can be turned into an effective DOS attack. Such attacks not only degrade system utilization but also defeat the primary task of the scheduler, preventing fair priority-based CPU sharing.

4 Experimental Methodology

In this section, we describe the simulation methodology, hardware parameters and benchmarks that we use in our experiments. We demonstrate heat stroke and selective sedation using execution driven simulations of an SMT. We build our SMT simulator on SimpleScalar 3.0b [3]. Our simulator uses the ICOUNT fetch policy and can fetch from two threads every cycle. The architectural configuration parameters are shown in Table 1. Our SMT simulator implements common optimizations techniques such as squashing a thread on an L2 miss to avoid filling up the issue queue. All the architectural techniques which we use in our SMT simulator are commonly used in commercially available SMT processors.

To model the power consumption, we integrated Wattch [2] with our base SMT simulator. We extend the Wattch model to include HotSpot [12] in order to model the power density in the SMT simulator. The processor runs at 4GHz

Table 1: System parameters.

Architectural Parameters	
Instruction issue	6, out-of-order
L1	64KB 4-way i & d, 2-cycle
L2	2M 8way shared 12-cycle
RUU/LSQ	128/32 entries
Memory ports	2
Off-chip memory latency	300 cycles
SMT	2 contexts
Power Density Parameters	
Vdd	1.1 V
Base Frequency	4 Ghz
Convection resistance	0.8 K/W
Heat-sink thickness	6.9 mm
Thermal RC cooling time	10 ms

frequency and senses the temperature every 20,000 cycles (this sensing frequency is well under the thermal RC time-constant of any resource). The circuit and packaging parameters are shown in Table 1. For the core of the processor we use the floorplan provided in [12]. We use a chip-wide Vdd of 1.1 V. The parameters correspond to next-generation of high-performance processors according to [11]. Our thermal packaging corresponds to an air-cooled, high performance system.

While there are a number of proposed techniques for addressing hots-spots [12], in our power-density SMT simulator, we use *stop-and-go* as the base-case technique for preventing hot-spots. From Figure 6 in [12], we see that for realistic configurations *stop-and-go* (called global clock gating in [12]) has nearly the same throughput as DVS. Further, DVS is not expected to scale with technology. DVS reduces Vdd to reduce the power consumption which may not be possible in future for scaled, low-voltage technologies. Transistor threshold voltage scales more slowly than the supply voltage [11] and as the gap between the supply voltage (e.g., 1.1V) and the threshold voltage (e.g., 0.25 V) closes, there is a substantially less flexibility for DVS. Finally, supply voltage reduction may cause soft errors or prevent transistors from switching even at reduced clock frequencies. Because stop-and-go performs comparably to other schemes for our purposes, and is already implemented in commercially available processors today, we use stop-and-go as the base-case method for hot-spot prevention.

We do not compare against a number of other techniques which are either not generally applicable or create implementation difficulties. Temperature-Tracking Frequency Scaling (TTDFS), as proposed in [12], allows the processor to heat above its "maximum" temperature by slowing the clock and relaxing timing constraints. As stated in [12] 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.

We run each simulation for 500 million cycles. This duration corresponds to the quantum of a typical operating system. To show the effect of Heat-Stroke in a 2-way SMT, we run each individual SPEC2K benchmark with a malicious thread similar to the one shown in Figure 2.

In the implementation of selective sedation we maintain per-thread access-rate counters corresponding to each resource. We sample the access-rate every 1000 cycles. Because it takes in the order of a million cycles to create a hot-spot, we maintain a weighted average that captures a window of 0.5 million cycles by choosing the value of x (in Section 3.2.1) to be 1/128.

5 Results

To demonstrate heat stroke and selective sedation, we present a number of experimental results. In all experiments, except where stated otherwise, we run one program from the SPEC2K suite and one malicious program, on a 2-way SMT. Except for Section 5.5, we use a convection resistance of 0.8 K/W [12] for the realistic heat. In all experiments we assume stop-and-go as the base-case technique for addressing power-density problems. We used 358 K as the highest allowable operating temperature [12]. Except for Section 5.6, we use 356 K as the upper-threshold and 355 K as the lower-threshold for selective sedation.

In our experimental evaluations we demonstrate the following results: (1) We show the number of times that the SMT pipeline heats up to the emergency temperature, with and without selective sedation. (2) We contrast the average access-rate behavior of SPEC programs against the behavior of malicious programs. (3) We demonstrate the actual performance degradation of SPEC programs due to heat stroke, and the effectiveness of selective sedation in restoring their performance. (4) We show the breakdown of execution times for benchmark programs to illustrate how selective sedation prevents heat-stroke. (5) In order to establish the robustness of temperature-based thresholds, we vary the thresholds and show that the effectiveness of selective sedation is not critically sensitive to the thresholds we choose. (6) We show that both the damage from heat-stroke and the effectiveness of selective sedation remain unchanged qualitatively with improvements in heat-sinks and packaging technologies. (7) To show that selective sedation does not adversely affect the execution of non-malicious program, we execute pairs of only SPEC programs without any malicious threads.

In order to isolate the effects of the ICOUNT policy and to establish the effectiveness of the weighted-average resource-usage metric (Section 3.2.1), we use three variations of the malicious code. *Variant1* is an aggressive program which accesses the register-file at a high rate and also has a high IPC (same as Figure 1). *Variant2* is also an aggressive program with a high register-file access-rate but has a relatively lower IPC (same as Figure 2). *Variant3* is a

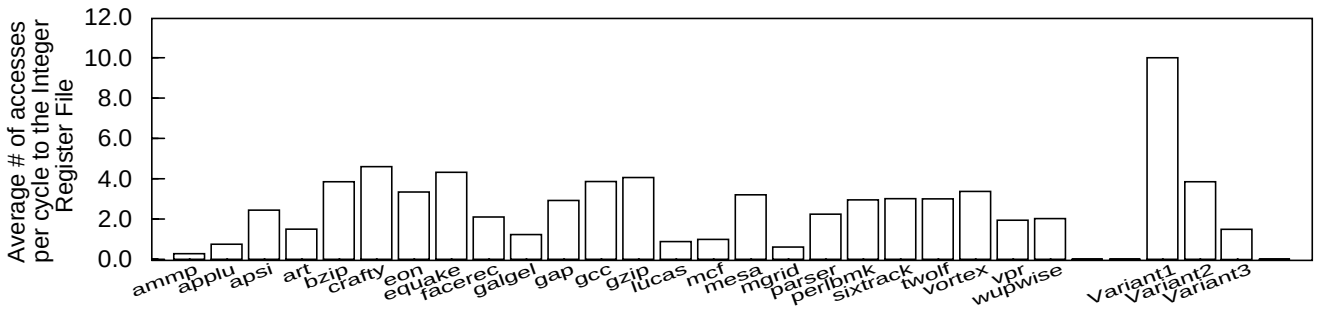

FIGURE 3: Average access-rates of integer register-file for SPEC programs and the three variants

moderately malicious program that accesses the register file at a rate chosen to evade selective sedation (variation of Figure 2).

5.1 Average Access-Rates

In Section 3.2.1 we introduced the weighted average metric, and argued that the flat access-rate averaged over a period of time is not a viable metric for identifying problematic threads. We now show this flat average access-rate of the integer register-file for SPEC benchmarks against those for the three malicious variants.

We execute each program alone, periodically sampling the access-rate and averaging the samples over a duration of one OS quantum, to effectively obtain the accesses per cycle for that program. In Figure 3 we show the average access-rate of the integer register-file for all SPEC benchmarks and for the three malicious variants. We observe that the average access-rate stays below 6 for all SPEC benchmarks. For *variant1* the average access-rate is 10, which is widely separated from the access-rates of SPEC programs. However, the average access-rates for *variant2* and *variant3* are 4 and 1.5 respectively, which are not distinguishable from that of SPEC programs. We show in Section 5.3, that *variant2* successfully inflicts heat-stroke, whereas the low access-rate of *variant3* limits its ability to inflict heat-stroke. We further show in Section 5.3 that the weighted average metric successfully detects and contains attacks by *variant2*. We also show in Section 5.3 that *variant1* monopolizes fetch and significantly affects SPEC programs due both to ICOUNT and power-density problems. Because *variant2* does not monopolize fetch, we consider *variant2* to be representative of malicious programs meant for heat-stroke attacks. In all those experiments where, for lack of space, we can show only one variant, we choose to show results for *variant2*.

5.2 Number of Temperature Emergencies

For a reasonable heat-sink, we expect that non-malicious programs will rarely heat an SMT processor up to the emergency temperature. However, in the presence of malicious programs we expect the number of temperature emergencies to be significantly high. Because selective sedation alleviates heat stroke, we expect selective sedation to successfully reduce the number of temperature emergencies back to a normal level.

In Figure 4 we show the number of times the processor heats up to the emergency temperature, within one OS quantum, for various benchmarks. For each benchmark we show three bars corresponding to, from left to right: (1) The SPEC benchmark executes alone. (2) The SPEC benchmark executes along with *variant2* while supervised by stop-and-go. (3) The SPEC benchmark executes along with *variant2* while supervised by selective sedation. We see that with the exception of bzip, crafty, equake, sixtrack and vortex, all benchmarks cause none or a few temperature emergencies when executing alone (1st bar). In the presence of *variant2* (2nd bar), the number of temperature emergencies increases to at least 8 for all benchmarks, amounting to more than a four fold increase in temperature emergencies averaged across all benchmarks. After deploying selective sedation (3rd bar), we see that, with the exception of bzip, crafty, gzip, mcf and wupwise, the number of

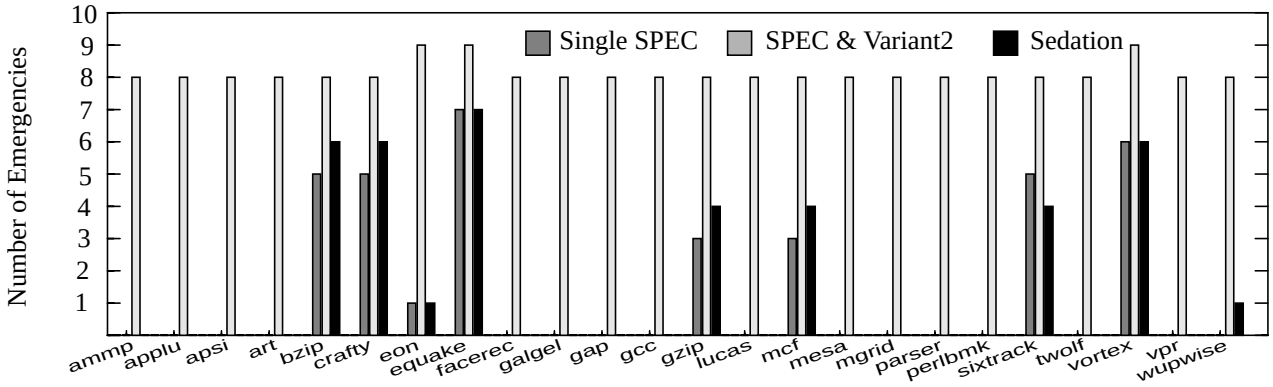

FIGURE 4: Number of temperature emergencies in one OS Quantum

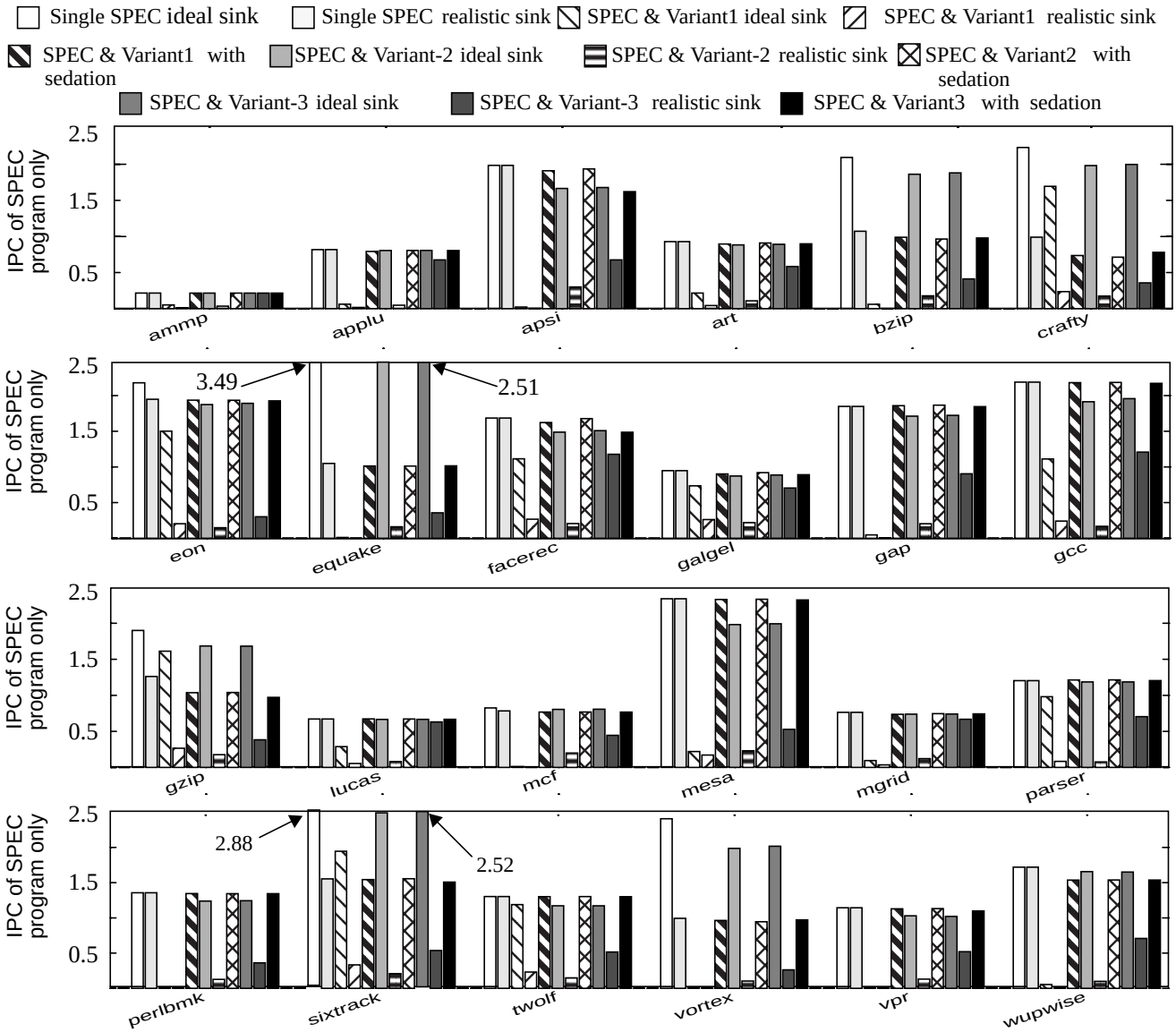

FIGURE 5: IPC Performance with Heat-Stroke and Selective Sedation

temperature emergencies for all benchmarks is restored to exactly the same number as for solo execution. bzip, crafty, gzip and mcf are benchmarks which already have power-density problems. The sedated execution of a malicious thread slightly increments their power-density problem converting a few near-emergency instances into actual temperature emergencies.

5.3 Effects of Heat Stroke and Selective Sedation

We now show the effect of heat stroke on the performance of targeted programs, and show that selective sedation effectively counters such DOS attacks. To establish that heat stroke is a real problem in SMT, we must isolate any effects of the ICOUNT fetch-policy, and of heat-sink limitations. For the first two bars in Figure 5, we run each SPEC benchmark *alone*, once with an ideal heat-sink (i.e., one that has infinite heat removal rate), and once with a realistic heat-sink (i.e., one that has a finite and reasonable heat removal rate). If the realistic heat-sink is effective, we expect that most benchmarks will not exhibit performance degradation compared to the case of an ideal heat-sink. We then run each SPEC benchmark in simultaneous execution with each of the three malicious invariants (Section 5), one by one. In Figure 5, for every benchmark-variant pair we show three bars, one for each of the following configurations: (1) An ideal heat-sink. (2) A realistic heat-sink, supervised by stop-and-go. (3) A realistic heat sink, supervised by selective-sedation. In all configurations we measure the IPC performance of the benchmark program only (the y-axis is the IPC of only the SPEC program). If a malicious monopolizes fetch, we expect the first configuration (3^{rd},6^{th}, and 9^{th} bar) to show a noticeable performance degradation compared to the case of ideal-heat-sink, solo benchmark execution (1^{st} bar). If the malicious variant is capable of inflicting heat stroke we expect the second con-

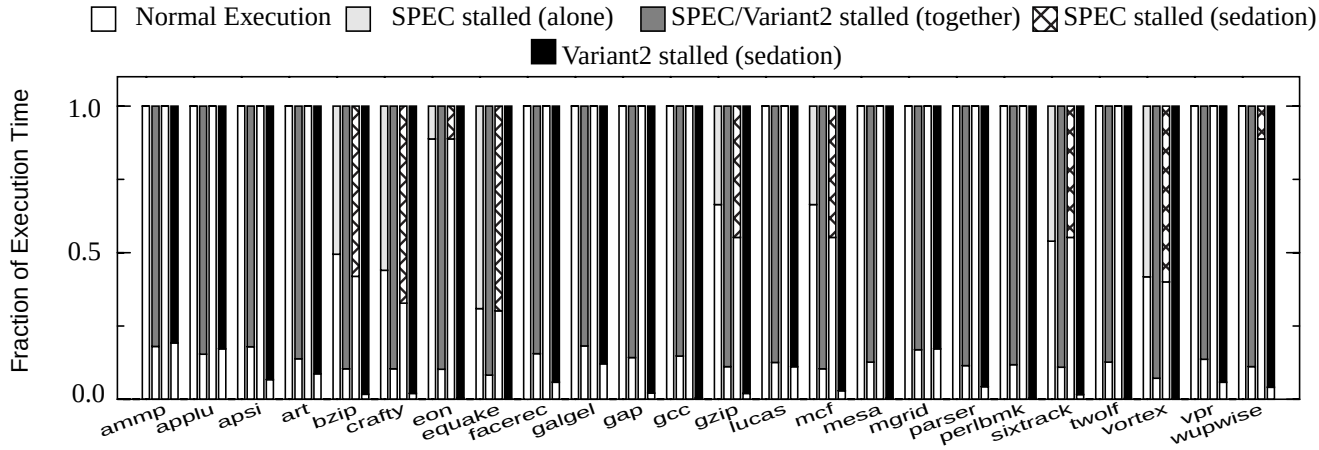

FIGURE 6: Breakup of execution times

figuration (4th,7th, and 10th bar) to show substantial performance degradation compared to the first configuration. Because selective sedation successfully counters heat stroke attacks, we expect the third configuration (5th,8th, and 11th bar) to perform significantly better than the second configuration, achieving a performance comparable to that of solo-execution with realistic a heat-sink (2nd bar).

With the exception of bzip, crafty, equake, sixtrack and vortex, we see that the solo-execution performance for all benchmarks is nearly the same regardless of ideal or realistic heat-sinks. We conclude that the realistic heat-sink is effective in heat-removal rate for typical programs, and that heat-stroke does not exploit an ineffective heat-sink.

Recall that the first of the simultaneous-execution configurations (3rd,6th, and 9th bar) attempts to isolate any ICOUNT-policy side-effects. From Figure 5 we see that *variant1* (3rd bar) exhibits noticeable performance degradation across many benchmarks for this first configuration. Whereas, *variant2* (6th bar) and *variant3* (9th bar) perform comparably to the ideal-heat-sink, solo-execution case across most benchmarks (1st bar). We conclude that *variant2* and *variant3* are free from any ICOUNT-policy side-effects, whereas *variant1* involves such side-effects. For the remainder of our experiments we do not consider *variant1* in the context of heat stroke.

The second of the simultaneous-execution configurations (4rd,7th, and 10th bar) shows the extent of heat stroke that the malicious variants inflict. Recall that *variant3* moderates its access rate in attempt to avoid being detected and contained by selective sedation. From Figure 5 we see that *variant3* (10th bar) causes a performance degradation that is much less pronounced compared to that of *variant2* (7th bar). The performance degradation averaged across all benchmarks is 50.8% for *variant3*, whereas for *variant2* it is as severe as 88.2%. We see that heat stroke is a real problem, which can severely degrade the performance of SMT systems.

We illustrate the effectiveness of selective sedation using the third configuration of simultaneous execution (5th,8th, and 11th bar). From Figure 5 we see that, for all three variants, selective sedation successfully restores the performance of the benchmark programs to a level comparable to that of their solo-execution with a realistic heat-sink (2nd bar). The IPC averaged over all benchmarks, for solo execution with a realistic heat sink is 1.28, whereas with *variant2* supervised by selective sedation (8nd bar) the IPC is 1.24. We conclude that selective sedation successfully prevents heat-stroke attacks from a variety of malicious programs.

5.4 Breakdown of Execution Times

When executing alone, we expect typical non-malicious programs to spend most of their execution time in normal operation. However, when executing along with *variant2*, we expect that SPEC benchmarks will spend a major fraction of their execution time in stalls due to stop-and-go cooling periods. Because selective sedation prevents heat-stroke, we expect that even in the presence of *variant2* SPEC benchmarks will spend only a small fraction of their execution time, if at all, in cooling-period stalls. At the same time, selective sedation should force *variant2* to spend a major fraction of its execution time in sedation stalls.

In Figure 6 we show the breakup of execution times for SPEC benchmarks under three scenarios: (1) Executing alone (1st bar). (2) Executing along with *variant2*, supervised by stop-and-go (2nd bar) (3) Executing along with *variant2*, supervised by selective sedation (3rd bar). We also show the breakup of the execution time of *variant2*, supervised by selective sedation (4th bar). We see that when running solo (1st bar), averaged across all benchmarks, SPEC programs spend 85% of total execution time in normal operation and only 15% in stalls for cooling. Most programs spend their entire time in normal execution, except for bzip, crafty, equake, gzip, mcf, sixtrack and vortex, which have slight power-density problems. Under heat-stroke conditions (2nd bar), averaged across all benchmarks, SPEC programs spend as much as 87% in cooling-period stalls, resulting in severe performance degradation. Selective sedation counters heat-stroke attacks (3rd bar), enabling SPEC programs to spend as much as 83% of their

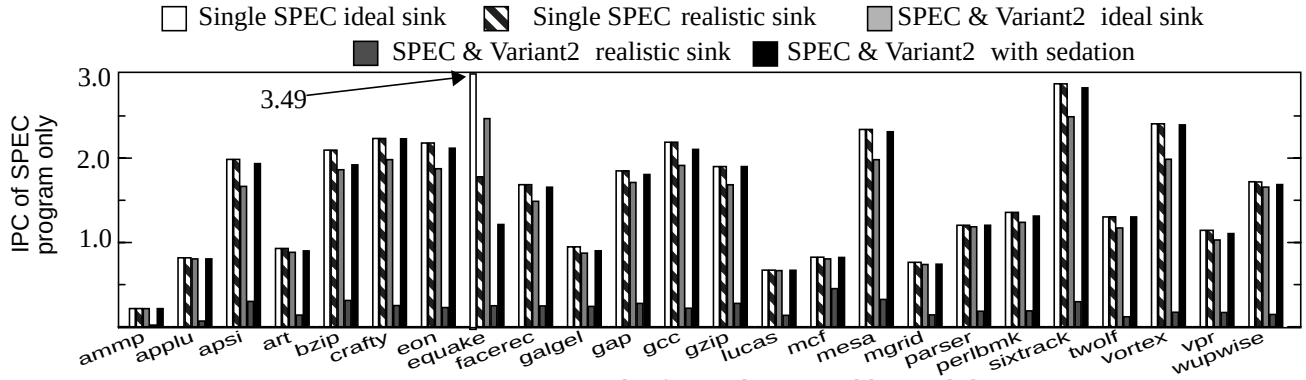

FIGURE 7: Results for an improved heat-sink

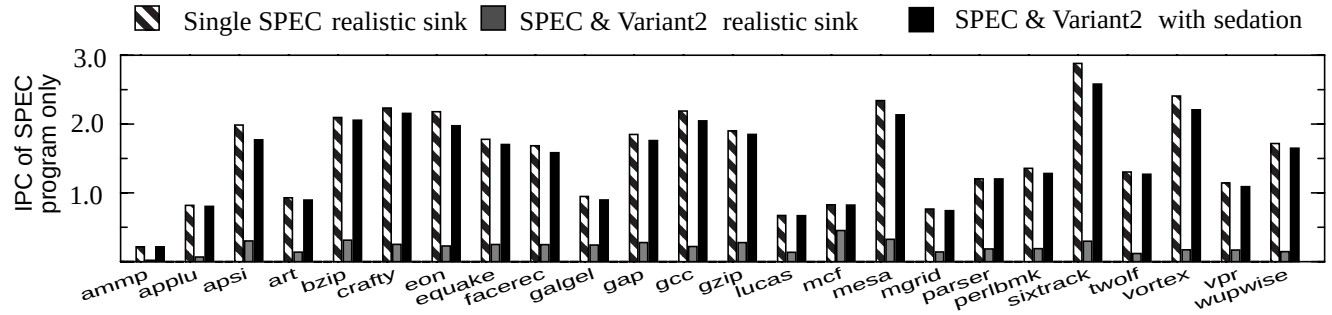

FIGURE 8: Results for varied thresholds with improved heat sink

execution time in normal operation, averaged across all benchmarks. Under selective sedation, all benchmarks spend nearly as much time in normal operation as they do in solo execution. We also see that, selective sedation successfully identifies *variant2* as the culprit program and sedates it for as much as 99.9% of its execution time (4th bar), averaged across all benchmarks.

5.5 Evaluations with an Improved Heat-Sink

In all experiments so far, we assumed a convection resistance of 0.8 K/W for a realistic heat-sink. We now repeat the evaluations shown in Section 5.3 but with a heat-sink which has a convection resistance of 0.7 K/W. Due to lack of space we present results only corresponding to *variant2*. In Figure 7 we see that when executing alone, except for equake, all other benchmarks' performance with realistic sink (2nd bar) is similar to that of the ideal heat-sink case (1st bar). The 1st and 2nd bars are closer in this figure than in Figure 5 due to the better heat-sink shown here. When executing along with *variant2*, and supervised by stop-and-go (4th bar), all benchmarks suffer substantial performance loss, showing that heat-stroke is just as effective even with an improved heat-sink. The IPC performance of all benchmarks, when executing with *variant2*, and supervised by selective sedation (5th bar), is comparable to their solo-execution performance with a realistic heat-sink (2nd bar). We conclude that heat-stroke is a threat to be reckoned with even with improved packaging and cooling technologies, and that selective sedation remains effective.

5.6 Threshold Sensitivity

We now vary the temperature thresholds to show that the effectiveness of selective sedation is not critically sensitive to the thresholds we choose. We use the improved heat sink of Section 5.5 for this experiment because it allows a larger threshold variation. We repeat the evaluations of Section 5.3, but with the upper- and lower-thresholds set to 353 K and 352 K respectively. We also evaluated threshold sensitivity for the less-aggressive heat-sink and obtained similar results. Due to lack of space we present evaluations only for the improved heat-sink and only for *variant2*.

The results for the modified thresholds, shown in Figure 8 differ from the results presented in Section 5.5 only in the 3rd bar which corresponds to the benchmarks executing along with *variant2*, while supervised by selective sedation. We see that even for the new set of thresholds, selective sedation successfully prevents heat-stroke. The IPC averaged over all benchmarks, for solo execution with a realistic heat sink is 1.56 (1st bar), whereas with *variant2* supervised by selective sedation the IPC is 1.47 (3rd bar). We conclude that the effectiveness of selective sedation is not critically sensitive to the choice of temperature thresholds.

5.7 Effect on Non-malicious Programs

We now investigate whether selective sedation adversely affects the performance of non-malicious programs. We observe that equake exhibits a greater degree of power-density problem that all other SPEC benchmarks. Therefore, we run equake paired with other SPEC programs on the SMT and observe their combined IPC performance with

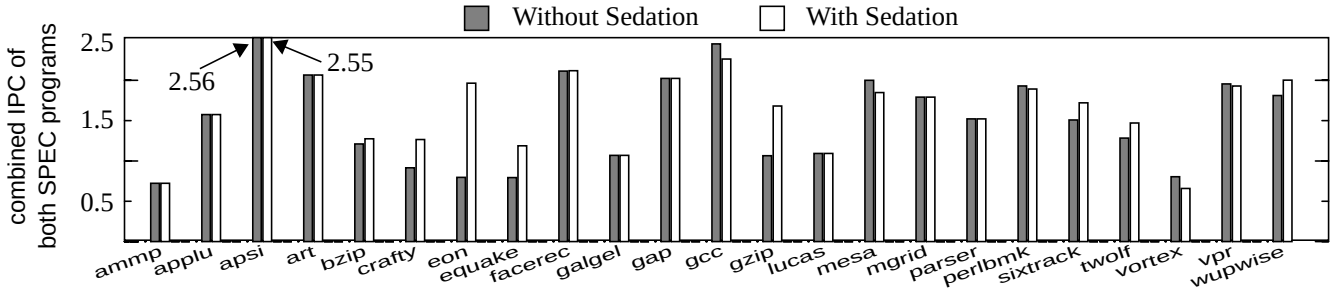

FIGURE 9: Total IPC of two SPEC threads (equake paired with other SPEC benchmarks)

and without selective sedation.

In Figure 9 we show the total IPC of the SMT for pairs of SPEC benchmarks for (1) with a realistic heat-sink using stop-and-go (1st bar), and (2) with a realistic heat-sink using selective sedation (2nd bar). We see that the IPC for selective sedation is comparable to the IPC for stop-and-go. In fact for a number of benchmarks (e.g., eon, gzip) selective sedation improves the performance in comparison to stop-and-go by preempting emergencies. We conclude that selective sedation has no adverse effect on the performance of non-malicious programs.

6 Conclusions

In this paper we describe how power-density can be exploited to launch DOS attacks in SMT. Currently known techniques that address the power-density problem slow down the entire SMT pipeline degrading the performance of all threads. A malicious thread can inflict heat-stroke, a novel DOS attack, by repeatedly creating hot-spots to adversely affect the performance of other threads in the system. We made two key observations to address heat-stroke: (1) The average resource access behavior of malicious threads is distinctly different from that of non-malicious threads. (2) When a hot-spot occurs it is not necessary to stall all the threads in the SMT, rather we need to stall only that thread which is responsible for the hot-spot. We proposed to detect power-density problems using temperature-based thresholds, and to identify culprit threads using a weighted average of their resource access-rates. We proposed selective sedation, a scheme that selectively penalizes only the culprit thread in the event of a hot-spot, while allowing other threads to make normal progress. Through experimental evaluations we showed that heat-stoke is a real problem which can severely degrade the performance of the threads in an SMT. We showed that selective sedation successfully prevents heat-stroke.

Because SMT is being widely adopted by the microprocessor industry and because power-density is becoming increasingly problematic, it is important to understand and to propose solutions to attacks such as heat-stroke.

7 References

[1] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Seventh International Symposium on High Performance Computer Architecture*, 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] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, June 1997.

[4] CERT Coordination Center. Denial of Service Attacks. http://www.cert.org/tech_tips/denial_of_service.html.

[5] CERT Coordination Center. Denial-of-Service Incidents. http://www.cert.org/research/JHThesis/Chapter11.html.

[6] CERT Coordination Center. TCP SYN Flooding and IP Spoofing Attacks. http://www.cert.org/advisories/CA-1996-21.html.

[7] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 409–418, 2002.

[8] 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.

[9] 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.

[10] F. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies. In *Keynote speech: 32nd International Symposium on Microarchitecture*, Dec. 1999.

[11] SIA. *International Technology Roadmap for Semiconductors (ITRS)*. http://public.itrs.net/Files/2002Update/2002Update.htm, 2002.

[12] 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*, pages 2–13, June 2003.

[13] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 66–76, 2002.

[14] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 191–202, June 1996.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[16] R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur. Thermal performance challenges from silicon to systems. In *Intel Technology Journal 3Q 2000*, Q3 2000.