Reducing Peak Power with a Table-Driven Adaptive Processor Core

Vasileios Kontorinis
University of California,
San Diego

Amirali Shayan
University of California,
San Diego

Rakesh Kumar
University of Illinois,
Urbana-Champaign

Dean M. Tullsen
University of California,
San Diego

ABSTRACT

The increasing power dissipation of current processors and processor cores constrains design options, increases packaging and cooling costs, increases power delivery costs, and decreases reliability. Much research has been focused on decreasing average power dissipation, which most directly addresses cooling costs and reliability. However, much less has been done to decrease peak power, which most directly impacts the processor design, packaging, and power delivery. This research proposes a new architecture which provides a significant decrease in peak power with limited performance loss. It does this through the use of a highly adaptive processor. Many components of the processor can be configured at different levels, but because they are centrally controlled, the architecture can guarantee that they are never all configured maximally at the same time. This paper describes this adaptive processor and explores mechanisms for transitioning between allowed configurations to maximize performance within a peak power constraint. Such an architecture can cut peak power by 25% with less than 5% performance loss; among other advantages, this frees 5.3% of total core area used for decoupling capacitors.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Adaptable architectures

General Terms

Design, Experimentation, Performance

Keywords

peak power, adaptive architectures, resource resizing, decoupling capacitance, voltage variation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...$10.00.

1. INTRODUCTION

The power dissipation of processors has become a first-class concern for both mobile devices as well as high-performance processors. This issue only becomes more challenging as we continue to pack more cores and other devices onto the processor die. Recent research has presented a number of techniques that enable a reduction in the average power of a processor or processor core and reduce the adverse effects of high power on cost and reliability [16, 39, 41, 7]. Much less attention has been devoted to peak power; however, the peak power dissipation of a processor is also a first-class concern because it drives the design of the processor, thermal budgeting for the processor and system, the packaging and cooling costs, and the power supply costs and efficiency.

Designing robust power delivery for high performance processors requires expensive power supply and packaging solutions, including additional layers and decoupling capacitors embedded in the packaging [25]. Placing on-chip decoupling capacitors on heavily congested silicon areas to prevent power overshoots is not always possible [38]. Due to the random and unpredictable nature of the operational power demand, power distribution networks are over-designed for a worst case scenario which rarely occurs in normal operation. Peak power reduction will reduce on chip voltage variation along with the associated timing uncertainty and potential signal failures, or it will allow designers to achieve the same voltage variation with less silicon real estate for decoupling capacitance.

Also, power supply efficiency is proportional to the peak load current demand of the processor. A power supply that is designed for the peak power requirement will be less efficient when supplying loads that are substantially under the peak power [42]. That is, reducing peak power will result in both cheaper power supply solutions, but also power supplies that consume less power (even if the delivered average power has not changed) due to the increase in power supply efficiency.

This paper presents a peak power management technique for current and future processors that attempts to guarantee that the peak power consumption of a processor is far lower than the sum of all core blocks. We do this via a highly adaptive processor and a table-driven approach to configuring the processor. In prior approaches [1, 6, 23, 10], adaptive elements such as caches, queues, functional units, etc. make local decisions to decrease size based on local activity. This reduces average power, but does nothing for peak power. The same worst-case scenario which could theoretically incur a power drain close to peak power would also cause all of those elements to be maximally configured at once.

In our table-driven approach, resources are centrally configured. This central control allows us to guarantee that not all resources are maximally configured at once. We can choose an arbitrary peak power design point, and only allow those configurations that do not exceed that point. Performance loss is minimal because we still allow any single resource to be maximally configured, just not all at once. We show that typical applications get close to full performance as long as their primary bottleneck resources are fully configured. We find that it is important to be able to adapt dynamically, because not all applications have the same bottleneck resources.

Thus, the discussed table-driven adaptive architecture requires configurable components, a table of possible configurations, and a mechanism for selecting the configuration to run in the next epoch. This research explores all of these design issues. We find that we are able to reduce peak power by 25% with only a 5% loss in performance.

This paper is organized as follows. Section 2 discusses related work. Motivation and mechanisms for reducing peak power dissipation are discussed in Section 3. In Section 4, the adaptive architecture for reducing peak power is presented. The methodology is provided in Section 5. Results are shown in Section 6 and Section 7 concludes.

2. RELATED WORK

Albonesi, et al.[2] present a comprehensive study on how to tune processor resources in order to achieve better energy efficiency with several hardware and software techniques. We tune similar resources and in some cases even use the same hardware mechanisms for hardware adaptation. However, the policies for triggering and evaluating the potential configurations are completely different when targeting average power.

Most related work in adaptive micro-architectures considers a small number of configurable parameters. However, Lee et al. [22] explore the trends and limits of adaptivity in micro-processor design. They conduct a thorough study of a 240 billion point design space employing statistical inference techniques and genetic algorithms and provide intuition regarding the benefits of dynamically adapting processor resources in terms of performance, power, and efficiency. However, they do not provide run time heuristics that could be employed in order to dynamically adapt the processor and they target average power.

Dynamic Thermal Management [4, 15] is a reactive technique that does not control peak power but does ensure that the power dissipation does not cause the temperature to exceed a threshold. It does this by reacting to thermal sensors and decreasing processor activity when temperature thresholds are exceeded. These techniques can reduce packaging costs related to heat dissipation and reduce temperature-related failures.

The research of Isci, et al.[16] and Meng, et al.[24] address the problem of meeting a global power budget while minimizing performance loss. The former employs dynamic voltage and frequency scaling (DVFS) and uses a trial-and-error method for system performance optimization. The latter uses DVFS and cache adaptation in combination with analytic models for performance and power prediction. However, both works treat peak power budget as a soft limitation that can temporarily be exceeded and rely on power sensor measurements for reactively tuning the power consumption. Our architecture is designed to provide peak power guarantees and in that context does not rely on power sensor measurements.

Sartori and Kumar [32, 33] target peak power on multicores. They employ DVFS to limit the peak power consumption of each core in a pro-active way that prevents global power overshoots. Our work is orthogonal and complementary to Sartori's work. Assuming different levels of peak power per core, decentralized peak power management can be used on top of our core design. This paper focuses on reducing the peak power of each core, providing a building block for global multi-core power budgeting solutions.

In this paper, we exploit a number of previously proposed, local adaptive techniques originally designed to reduce average power in particular components. These references are given in Section 4.2.

3. THE POWER OF ADAPTATION

Bounding peak power has several key advantages. It can reduce packaging and cooling costs. It will reduce the cost and increase the efficiency of the power supply. It may eliminate the need for thermal sensors in cost-sensitive designs (high performance designs would likely still need them, but perhaps fewer). It allows more aggressive design in other parts of the processor or system. It reduces the need for large decoupling capacitors on chip. Power delivery circuitry does not have to be over-designed.

Reactive dynamic thermal management techniques only provide some of these advantages. It should also be noted that these techniques are not mutually exclusive, but actually complementary. Our adaptive processor with peak power guarantees can be a highly effective part of the response to thermal emergencies, because it allows us to change the peak power reactively. Also, while the reactive mechanisms do nothing for average power in the absence of thermal events, our architecture also reduces average power, because it always has part of the core turned off.

A good general-purpose processor will typically be configured such that any program that runs on the processor will find it's most critical resources sufficiently supplied to enable high throughput — this is what makes it a good general-purpose processor. However, it is rare that a *single* application needs all of those resources. We find that most applications have only a few bottlenecks, and as long as those resources are sufficiently supplied, the program can get nearly full performance.

Figure 1 motivates this research. It shows the result of experiments that provide a fully configured processor (similar to the MAX_CONF design point described later in this paper), a minimally configured processor where most resources are dramatically reduced (cut in half), and a series of intermediate processors where a subset of the resources are at full capacity. The resources that change include instruction cache, data cache, integer and floating point instruction queues, reorder-buffer, load-store execution units, integer and floating point execution units, and renaming registers.

The stacked bar chart in the specific graph depicts (1) the average speedup over the minimally configured core when one resource is maximized for each benchmark (2) the average speedup when two resources are maximized, (3) the average speedup when three resources are maximized, and (4) the fully configured result. When results are shown for a subset of resources at maximum, we select those resources that give the highest performance for that particular benchmark. Two observations jump out of this graph. The first is that the total difference between the fully configured performance and the minimally configured (the baseline for the graph) is quite significant, since the former is approximately 50% faster than the latter.

Even more significantly, we see that we can cover 65% of the gap between the two by only providing two out of ten resources at full capacity, as long as we choose those two resources carefully. This number rises to more than 85% when we provide three resources at full capacity. This is a powerful result — we can heavily under-configure 70% of the processor's components (very roughly speaking) and give up little performance. The bottleneck resources vary between applications, so we cannot achieve this same result with a single static configuration. This is confirmed in Table 1, where the most important resources per application are presented. Lee and Brooks [22] note similar variance in bottlenecks, although

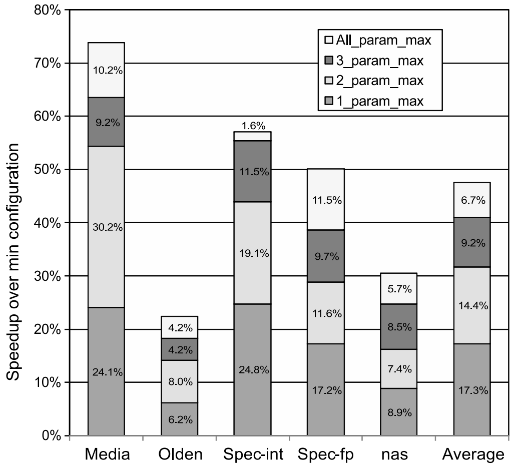

Figure 1: Shows the results of several experiments, where potential bottlenecks, or performance constraints, are selectively removed per experiment, for a variety of benchmarks. The stacked bar shows the performance gain when the most important constraint is removed, when two most important constraints are removed, when three most important constraints are removed and when all constraints are removed. Performance gain is relative to a minimally configured processor core.

the actual resources are different. These results indicate that we should be able to achieve close to full performance for any single application with a configuration whose cost (in peak power, in particular) is closer to the minimally configured core than it is to the maximally configured core.

There are two issues that might limit our ability to provide power guarantees, static leakage and process variation. Static leakage varies significantly with temperature. So, coupled with thermal sensors and a mechanism for dealing with thermal events (again, possibly using this architecture to deal with those events), our design can still provide a cap for both static and dynamic power. Process variation [17] is expected to result in cores that vary from the expected power behavior, even across a single chip. Again, this is an opportunity for this architecture, rather than a problem. Assuming we can detect this variation in testing/verification, we can either provide the same list of configurations to each core and thus provide a different guaranteed peak power for each core, or provide a different list of configurations to each core and thus provide the same peak power for each core (see Section 4.4). In each case, we still provide reduced peak power at the level of the processor.

4. AN ADAPTIVE ARCHITECTURE TO LIMIT PEAK POWER

Current high-performance processors rely on knowledge of expected, or average, application behavior to design packaging and thermal sensors to detect when behavior exceeds thermal bounds. For example, Intel's current power strategy can be seen in [15]. For each processor, a set of typical applications are run on the processor, and a power point is identified that is higher than the average power dissipation of most applications. This is called the Thermal Design Power (TDP). System designers are expected to create systems that can thermally handle this level of power. When power exceeds this threshold and temperature rises, thermal control circuitry is activated. Fig 13 in [15] demonstrates clearly that frequent activation of the thermal control circuit will result in significant performance loss for the end user. Such a design does not provide any peak power guarantees (only peak temperature), hence it is still susceptible to a power virus [26]. Moreover, TDP cannot be used for the power supply design, because unlike temperature, power demand can go to its maximum value in any given cycle. Further, the processor itself, including clock distribution, power delivery circuitry, decoupling capacitors, and other wiring, must be designed for the true peak power, not the TDP.

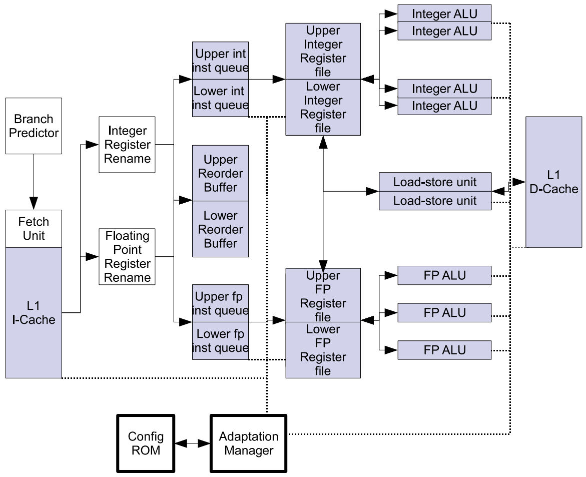

Figure 2: The adaptive architecture. The shaded parts are dynamically configured for power savings and max performance while peak power limits are guaranteed.

In contrast to reactive approaches, the architecture described here guarantees that power will never exceed certain thresholds, by ensuring that in any given cycle a certain portion of the processor core is not active. It is impossible for the processor to get into a state where too many resources are being used or in a high power state, even in the presence of a power virus. The same could be achieved by just designing a smaller core; however, some applications will always degrade because of the resources excluded. With an adaptive architecture, we give up little performance relative to the fully configured core, while achieving the peak power of a much smaller core.

Figure 2 shows our architecture. Most of the architecture is configurable. Control of all configurable pieces is managed centrally. Reconfigurations happen at very coarse intervals, so there is no latency issue related to central control. The Config ROM contains a set of configurations that are known not to exceed our peak power threshold. The Adaptation Manager collects performance counter information over an epoch, and based on that data possibly selects a new configuration from the Config ROM for the next epoch.

We assume our CPU is a core of a CMP in 65nm technology. We focus on a single core (including L1 caches) in this study, assuming each core limits peak power individually, resulting in a cumulative reduction in total processor power. The L2 cache could easily be incorporated, as well. For private L2 caches, it would be part of the core power management, and would only increase the gains shown in this paper. For a shared cache, it would need to be part of a global peak power solution.

Table 2 shows the explored architectural design space, including all components we consider configurable. The total number of configurations grows exponentially with the number of adaptable components, making prohibitive any hardware implementation that would need to evaluate a significant fraction of them dynamically.

Media	iq	fq	ialu	falu	ldst	ic	dc	ipr	fpr	rob
g721d			2			1	3			
mesa-texgen			2			1	3			
epic	3		1				2			
jpege			1			3	2			

Olden	iq	fq	ialu	falu	ldst	ic	dc	ipr	fpr	rob
perim_big	3		1				2			
mst			3			1		2		
treeadd	1							2		3
health	3					1		2		
bisort_med	3		2				1			
em3d_med						1		2		3

Spec-int	iq	fq	ialu	falu	ldst	ic	dc	ipr	fpr	rob
gzip-source			2			3	1			
vpr-route							1	2		3
gcc-scilab			3			1	2			
mcf	1		2					3		
crafty			3			1	2			
parser			3			1	2			
eon-cook			3			1	2			
perlbmk-makerand			3			1	2			
gap			2			1	3			
vortex-3			3			1	2			
bzip2-graphic			3				2	1		
twolf			3			1	2			

Spec-fp	iq	fq	ialu	falu	ldst	ic	dc	ipr	fpr	rob
wupwise		1	3			2				
swim		1							2	3
mgrid							1		2	3
applu		1			2				3	
mesa			3			1	2			
galgel							1	3	2	
art-110							3	1		2
equake		1						3	2	
facerec			2			1	3			
ammp		3		2				1		
lucas		2		1				3		
fma3d				3		1	2			
sixtrack		2		1		3				
apsi		3				1	2			

Nas	iq	fq	ialu	falu	ldst	ic	dc	ipr	fpr	rob
mg.big		3							1	2
ft.big		2		3			1			
sp.big		2							1	3
cg.big	3	1					2			
bt.big		1							2	3
ep.big		1		3		2				

iq: integer instruction queue	*fq:* floating point instruction queue	*ialu:* integer arithm. logic unit	*falu:* floating point arithm. logic unit	*ldst:* load/store unit
ic: instruction L1 cache	*dc:* data L1 cache	*ipr:* integer physical registers	*fpr:* floating point physical registers	*rob:* reorder buffer

Table 1: Presents the important parameters per benchmark. The resource that gives the best performance when maximized is marked with 1, the resource that gives the best performance when maximized in combination with the first one is marked with 2 and the resource that gives the best results with the first two is marked with 3.

INT instruction queue	16,32 entries
INT registers	64,128
FP instruction queue	16,32 entries
FP registers	64,128
INT alus	2,4
D cache	1,2,4,8 ways of 4k each
FP alus	1,2,3
I cache	1,2,4,8 ways of 4k each
Load/Store Units	1,2
Reorder Buffer	128,256 entries

Table 2: Design Space

Relative power threshold	Number of configurations
0-70 %	132
0-75 %	279
0-80 %	360
0-85 %	285
0-inf %	1

Table 3: Configuration distribution over different peak power groups

4.1 Filling the Config ROM

We assume that the Config ROM should contain all the reasonable resource configurations that the core can be in. The first challenge we encounter is what configurations to put in the ROM. This section describes our process.

We determine the peak power for every possible configuration in the design space of Table 2. The total number of configurations sums up to 6144, which corresponds to the strict product of the number of all resource configurations. After determining the peak power threshold, we then eliminate all configurations that exceed that threshold. In most cases (depending on the threshold), we are still left with a large number of configurations; however, many are redundant and unnecessary. We therefore also eliminate all redundant configurations – those that are a strict subset of another in our list. For example, if configuration B is identical to A, except that A has 8 cache ways and B has 4, B is eliminated. This step provides us with a reasonable set of configurations for all threshold levels. The remaining configurations then go in the Config ROM.

This heuristic makes the implicit assumption that a larger configuration always outperforms a smaller one. We found this to be *almost* always true. The most notable exception was *crafty* which runs up to 11% faster with configurations that are not the largest. With more resources, *crafty* executes further down the wrong path and as a result, wrong path loads and stores generate useless data traffic that occupy elements of the memory hierarchy once the program returns to the correct path. An architecture that wanted to also exploit this phenomenon (able to find an optimal architecture even when it is a strict subset of other available configurations) would need to have more configurations and likely a more complex algorithm for selecting the next configuration — and the additional gains would be small.

We consider several possible power thresholds in this research, although we assume the processor core is configured for a single threshold (Section 4.4 discusses other assumptions). Specifically we consider thresholds corresponding to 70%, 75%, 80%, 85% of the core's original peak power, as well as *no limit* which corresponds to the maximally configured core. It is difficult to go much lower than 70%, because much of the core power goes to things that we don't consider configurable. And even the configurable components are never disabled completely.

Table 3 shows how many configurations would appear in the table for each power threshold. The storage cost of the table is quite low – we require 13 bits per configuration (2 bits for FP ALUs, Icache and Dcache, 1 bit for the others), so the 75% list would require 454 bytes.

4.2 Configuration Mechanisms

Each epoch, a new configuration is potentially chosen. Section 6.2 describes the actual decision mechanisms used to select the next configuration. This section is concerned with the mechanics of switching configurations and adapting individual components once that selection is made.

Reconfiguration is done carefully to ensure no resources are ever added until other resources are reduced. For example, if in the next epoch the Adaptation Manager chooses a configuration with a larger Icache and a smaller integer queue, we must first wait until the instruction queue drains out sufficiently and we have successfully power gated it, and then we can power up the new Icache ways.

We attempt to model these delays accurately and conservatively, and include an additional power-gating delay in order to ensure that powering down and especially powering up is done in a phased way, so that induction noise, ground bounce, and rush current [12, 28, 8] will not be an issue. The adaptable components of the processor can be placed in two categories. Components such as integer and load store units maintain no state and can be partially power-gated immediately after they are signaled by the centralized adaptation mechanism. On the other hand, instruction queues, register files, and data caches must drain state before partial power-gating is allowed.

The literature contains several approaches for L1 cache power reduction. Way-prediction [30] will reduce dynamic power dissipation on successful predicts, while Drowsy caches [11] will reduce leakage power by periodically setting the bitcells in a drowsy, low-power mode and cache decay [18] will reduce leakage by selectively power-gating cache lines. Unfortunately, none of the above provide an upper bound on power. We adopt a method similar to the selective cache-ways work [1] with the important difference that the contents of a way are flushed and invalidated before the way is deactivated. Albonesi, et al. keep the contents of the memory cells valid and use or invalidate those on demand; we flush the dirty entries to the higher cache level and then invalidate the whole way. Consequently we can always cap the power activity of the specific cache since deactivated parts of the cache array can not dissipate any leakage or dynamic power. Our approach negatively impacts performance in the short term by increasing the stress on the memory subsystem. However, when the interval between adaptations is sufficiently large, the performance drop is insignificant.

For instruction queue partitioning, we assume a circuit mechanism similar to [6]. Integer and floating point instruction queues are both clustered in two 16-entry parts. Before disabling a cluster we ensure that all the instructions have issued by using a simple NOR of all active bits of the entries in the cluster. No compaction is assumed. Once the adaptation mechanism decides to shut down an instruction queue cluster, register renaming is throttled. Once the cluster issues all remaining instructions in that partition, we can begin power gating and resume renaming.

The register files and the reorder buffer of our architecture follows the resizing principles described in [10]. We segment the bitlines of the RAM components of the various buffers but at a much coarser granularity. Therefore we have a lower and upper cluster of the register file, and a lower and upper cluster of the reorder buffer. When we partially deactivate the structure we always disable the higher part and only after ensuring that the specific entries are not active. For that we again throttle register renaming during the adaptation period until the desired partition becomes idle. Unlike the queues, which always empty out eventually, register file parts may never empty without some intervention – in this case, by injecting some additional instructions to be renamed [10].

In [14] the authors provide details on ALU power gating together with several techniques to minimize performance loss once power saving techniques are applied. We assume similar mechanisms but instead use our centralized control.

Component	Power-up Delay (cyc)	Power-down Delay (cyc)
Dcache	651	163
Icache	335	84
Int inst queue	127	32
FP inst queue	127	32
Int Alus	198	50
FP Alus	375	94
Int reg file	277	69
FP reg file	277	69
rob	42	11

Table 4: Assumed delays to powergate components

Component	Min(cyc)	Average(cyc)	Max(cyc)
Dcache	163	384	1085
Int inst. queue	32	93	1725
FP inst queue	32	65	1741
Int reg file	69	101	3557
FP reg file	69	106	1108
ROB	11	114	2254

Table 5: Delays to disable components holding state

During the adaptation process handshake signals are exchanged between the Adaptation Manager that decides the next configuration of the processor from those stored in the Config ROM, and different adaptable components. Initially the Adaptation Manager will notify one or more components with a request for power down. The component will then take any necessary action (e.g., flush cache lines, etc.) to ensure safe partial deactivation as described above. After that completes, the unit signals the Adaptation Manager. At that point, the manager can initiate the power gating of the component, which will take some known number of cycles. Only at the end of that period can the Adaptation Manager initiate the power-up of those resources that will grow. Communication costs are minimal and happen very infrequently.

In Table 4 the delays to turn on and off each component are shown. According to [31], 200 ns suffice to power-up 1.2 million gates in 90nm technology. Ignoring the speed increase that should come with 65nm technology, and assuming linear scaling based on gate counts, our largest adaptive structure should be able to power up in a conservative 350 ns. Hence, powering-up the Dcache ways in the modeled core takes 350ns, and the delay for other components is faster, depending on each component's area. A 4:1 ratio was assumed between the delay to power up and power down a component, since powering down causes practically no inductive noise effect and should be much faster.

Table 5 summarizes the minimum, average, and maximum delays observed for powering down structures that hold state. This delay includes the assumed powergating delay and the time to free all occupied entries, but does not include communication costs with the Adaptation Manager.

While we have modeled all of these delays carefully, we show in Section 6.4 that performance is highly insensitive to these delays.

4.3 Implementation overhead

The configuration selection mechanisms we describe in Section 6.2 vary in their complexity – the computation is not complex, but in some cases the set of alternative configurations considered is large. Fortunately, the config ROM and the adaptation manager – the main additions of our technique – are accessed infrequently and are not part of the critical path.

Leveraging this fact, we can evaluate each configuration in the Config ROM consecutively, minimizing the hardware resources required for the Adaptation Manager. Assuming the most complex heuristic proposed in the results section, we need to estimate the "closeness vector" and the "weight vector". For the former we need to perform 10 2-bit subtractions (as many as the parameters we consider in each configuration) and for the latter we need to perform 10 23-bit comparisons between measured and threshold values (23 is log of the max interval used between adaptations – 8M cycles). Assuming that these subtractions and comparisons can be done consecutively as well, all we need is a small ALU capable of doing narrow width subtractions, a few additional state bits, and muxes. Our circuit analysis with standard cells in 65nm technology for this additional computational hardware indicates that it should add well under 1% to the total peak power and less than 0.5% of area. The Config ROM was conservatively modeled as RAM, using CACTI, and introduced less than 0.1% peak power overhead and 0.75% area overhead. In terms of average power the mechanism cost can be ignored since this small specialized engine is activated infrequently.

Given the tolerance of delays associated with this computation, and the frequency at which we adapt, we could even relegate the decision of selecting the best configuration to software. Even if the decision is made in software, we would propose still hardwiring the configurations in the hardware table to ensure that malicious software could never force an illegal configuration.

Design verification and testing will increase for the considered highly-adaptive core compared to a core with fixed-size resources. However, since we only permit a subset of all the possible core configurations testing will be simplified compared to an adaptive processor where configuration decisions are decoupled – such as the proposed architectures for average power reduction.

4.4 Static or Dynamic Filling of Config ROM

There is nothing preventing a single architecture from having multiple ROMs, or modifying the ROM contents over time (obviously in the latter case a reprogrammable ROM, or RAM, would be required), switching between them either at verification/test time, bootup time, or even runtime. This raises a number of new and promising potential uses of this technology.

(1) We could use it in conjunction with Dynamic Thermal Management – thermal events trigger the core to go into a lower peak power envelope. We could even have different power envelopes for other events – plugged in/not plugged in, low battery, etc.

(2) We could account for thermal heterogeneity in the processor (e.g., cores in the center tend to run hot, those on the edges cooler [9]) by applying a different peak power envelope to different cores.

(3) Similarly, we could counteract the effects of process variation by filling the ROM at verification time with a set of configurations which match the exact thermal and power characteristics of that individual core.

(4) Our technique can be coupled with fault tolerance. If a hard error is detected on an adaptable component, then the configurations that incorporate that specific partition can be disabled, as part of the core's overall fault isolation solution. In this way, we naturally redirect the power from faulty components to those that can use them.

(5) We could maintain an overall processor peak power envelope, but use it judiciously to maximize global throughput. We allow a core that can use the extra power for more performance to go to $P + \Delta$ while another is forced to $P - \Delta$. Our architecture, applied to a CMP, already provides a mechanism for transforming a homogeneous design into a heterogeneous architecture (each core configured separately to make the best use of its peak power limit). This optimization further enhances that ability, as we now have the ability to set the peak power limit of each core according to the needs of the individual application.

Several of these are the subject of ongoing or future research.

	Processor	Core
Peak Power Value(W)	36	28.57
Average Value(W)	20.87	15.90

Table 6: Absolute Power Values for the modeled processor

Cores	1	I cache	32k, 8 way
Fetch width	4	I cache miss penalty	8 cyc
INT instruction queue	32 entries	D cache	32k, 8 way
FP instruction queue	32 entries	D cache miss penalty	8 cyc
Reorder Buffer entries	256	shared L2 cache	2 MB, 4 way
FP registers	128	L2 miss penalty	40 cyc
INT registers	128	L3	4 MB, 4 way
Cache line size	64 bytes	L3 miss penalty	315 cyc
Frequency	1.83GHz	Vdd	1.2V

Table 7: Architectural Specification

5. METHODOLOGY

In order to evaluate different adaptation policies that optimize performance for given power budgets, we added support for dynamic adaptation to the SMTSIM simulator [37], integrated with the Wattch power models[5]. Wattch was modified so that the underlying CACTI [36] models have more updated circuit parameters and a reorder buffer is added to the power modeling.

5.1 Power Modeling

Wattch was developed for relative, activity-factor based power measurements and was originally validated against processors built in early 2000. Hence, it does not capture absolute power consumption for modern architectures but approximates well the relative power trends when the on-chip resources dynamically change. Wattch lacks a leakage power model and does not consider different circuit design styles or power reduction techniques. Our methodology is similar to the one in [19] and addresses the above issues. In that work, they apply linear scaling to the output of Wattch for a particular technology, so that it matches published values for a real processor at two points – peak power and average power. If we assume Wattch results reflect relative power changes accurately, this methodology should produce accurate absolute power results for the calibrated processor. Below we present the data used for the scaling. For this methodology, we need real processor peak power and average power, and the peak and average power from Wattch.

We get the Wattch peak power by maximizing all activity factors. We get Wattch typical power simply by running a number of benchmarks through it. For the target processor, we obtained the TDP value from the datasheets of Intel Core Solo at 65nm and assumed the latter as 75% of the peak power [15]. We also get the average power from [15] – in fact, this paper gives us a range of power values over a workload set which overlaps heavily with ours, allowing us to validate our model even more strongly.

This is only for dynamic power. We also need to add leakage power estimates. Specifically, we assume leakage power to account for 40% of overall typical power and approximately 28% of the peak power [13] for our 65 nm design. Since leakage scales with area, we break down the assumed value according to area ratios given by [21]. The L2 area is estimated as 70% of the core area from an Intel Core Solo die photo. The leakage values are additionally scaled linearly when a component is reconfigured dynamically, assuming that we always powergate deconfigured resources.

Table 6 gives the absolute peak and average power for the modeled processor as well as the corresponding core (processor excluding L2). Figure 3 presents the breakdown of peak and average power to different components. For the estimation of the average power we averaged the power consumption across our set of

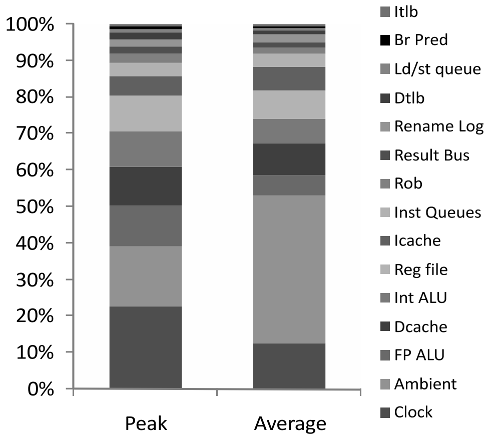

Figure 3: The breakdown to core components for the peak and average power. The power reported is comprised of dynamic and leakage power.

benchmarks when running on the maximally configured processor. We find that 50% of the total peak power and more than 40% of the average power is being used by our configurable resources, and the remaining power is used by non-configurable components. Because none of our resources are configurable to a size of zero, of course, only a portion of that power can be eliminated.

Table 7 gives the characteristics of our baseline architecture.

5.2 Benchmarks

In order to explore the benefits of adaptability we use the whole SPEC2000 benchmark suite and a selection (picked randomly) of Media, NAS and Olden benchmarks. Adaptive processors inherently are more effective in matching the varying needs of different programs. Hence, we use 42 benchmarks in total to expose all sorts of different execution behaviors. We use the Simpoint tool [35] to estimate the proper fast forward distance for up to 4 representative simpoints per benchmark and then we simulate 50M instructions at each simpoint. Multi-simpoint simulations were used to better capture intra-thread phase changes.

6. RESULTS

In this section, we first explore the potential for exploiting adaptivity for peak power reduction by modeling ideal static and dynamic configuration selection. We then examine real heuristics for navigating the configuration space dynamically.

6.1 Dynamic adaptation vs Static tuning

This section explores the potential of adaptivity to reduce peak power, while still limiting the performance hit introduced from the resource restrictions. Figures 4 through 6 present results for several oracle configuration policies (at each peak power threshold).

The policies simulated include the static worst per simpoint (WORST_PER_SIMPOINT), the static best per benchmark (BEST_PER_BENCH), and the static best per simpoint (BEST_PER_SIMPOINT). For those, the best (or worst) static configuration is identified for each application individually over the entire simulation interval (recall that our configuration trimming algorithm significantly restricts how bad "worst" can be). The fourth column (BEST_STATIC) corresponds to the best core configuration in the specific peak power chunk across all benchmarks. The fifth column (IDEAL_ADAPT) demonstrates the performance of a core than changes configurations dynamically and always chooses the best configuration for each interval of 1M instructions. Thus, the fourth bar represents potential performance from just designing the best possible non-configurable core within the peak power constraint. The second bar represents the potential from adaptivity to exploit inter-application diversity. The difference between the second and fourth bars represents the potential due to intra-application diversity. All the results were normalized to the BEST_STATIC, because this represents the best possible non-adaptive processor you could build within this peak power constraint, and constitutes our baseline. The last column (MAX_CONF) represents the fully configured core, which does not share the same peak power constraint as the rest of the core options in this graph.

Several conclusions come out of these figures. As expected the more limited the peak power budget the bigger the gains from adaptivity. For the lowest peak power budget an ideal adaptive core would perform on average 16% better than a core with fixed-size resources and tuned for best results (BEST_STATIC), while for the highest peak power budget the benefits are much smaller.

We also see that even with a tight peak power budget, we are able to get very close to the same performance as the full core. This confirms that most applications require just a few key resources.

But the bottleneck resources vary, and the best static core cannot give each of these applications what it needs most, while the adaptive core can.

Also, a larger selection of benchmarks, such as are used in real processor design, would almost certainly result in a larger gap between the compromise (best static) architecture and the adaptive architecture.

As we relax the peak power budget (for example, at 75%), we see a somewhat smaller gain from adaptivity over the static configuration, but we are able to nearly replicate the performance of the full core. At 80% of peak power the differences are smaller, but still noticeable. For an 85% threshold (results not shown), the best static core is pretty competitive with our adaptive core.

The use of multiple simpoints and the between-simpoint adaptation makes a significant difference in only gcc and bzip2. This indicates that overall we gain much more from inter-thread diversity than from intra-thread diversity.

Also notice in these results that we see the aforementioned behavior for *crafty*, and to a lesser extent for *vpr* — that they achieve higher results with a configuration smaller than the largest. Again, this is a result of being able to more aggressively pursue wrong branch paths with more resources, and is in general an anomalous result. Interestingly, this effect enables, at the higher peak thresholds (80% and 85%), the ideal adaptive technique to actually outperform the maximally configured core (by a small amount).

One measure of core design "goodness" is the ratio of average power to peak power. A low ratio implies an architecture that requires the system (including the processor design, the cooling, packaging, the power supply, etc.) be over-provisioned for the delivered performance. A high ratio, then, implies a more efficient design. Figure 7 presents the average power and peak power for the IDEAL_ADAPT architecture.

This architecture, then, shows two very positive trends, it reduces both average power and the ratio of peak to average power at the same time. It should be noted, in fact, that we have moved the ratio of peak to average power to a point that is more typical of an in-order processor core than the out-of-order core we are simulating [20]. Again, this enables the use of a more efficient power supply (better targeted at the average power), saving wall power.

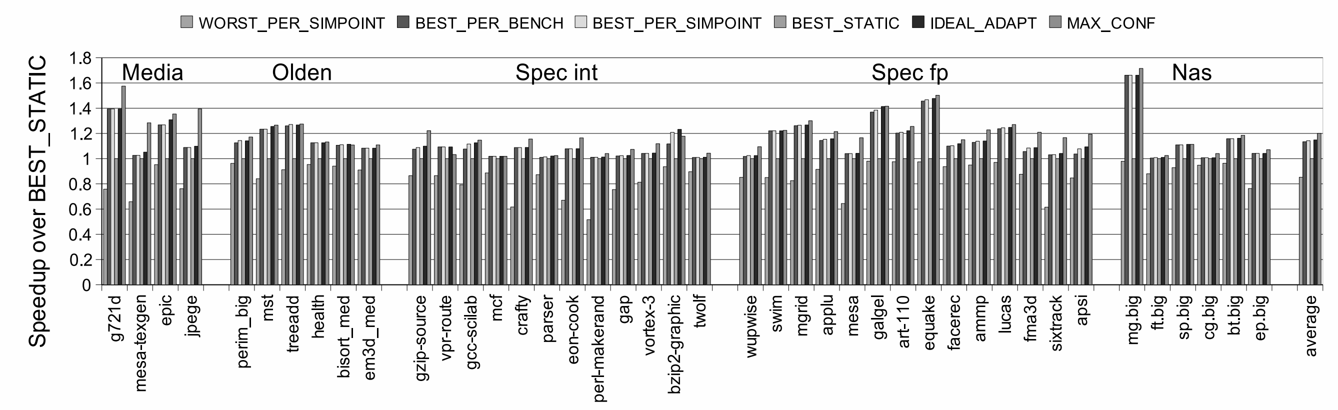

Figure 4: The performance of six configuration policies with a peak power constraint of 70% of the maximum configuration. The best static configuration is: iqs:32 fqs:32 ialu:2 falu:1 ldst:1 ics:16 dcs:16 ipr:64 fpr:64 rob:256

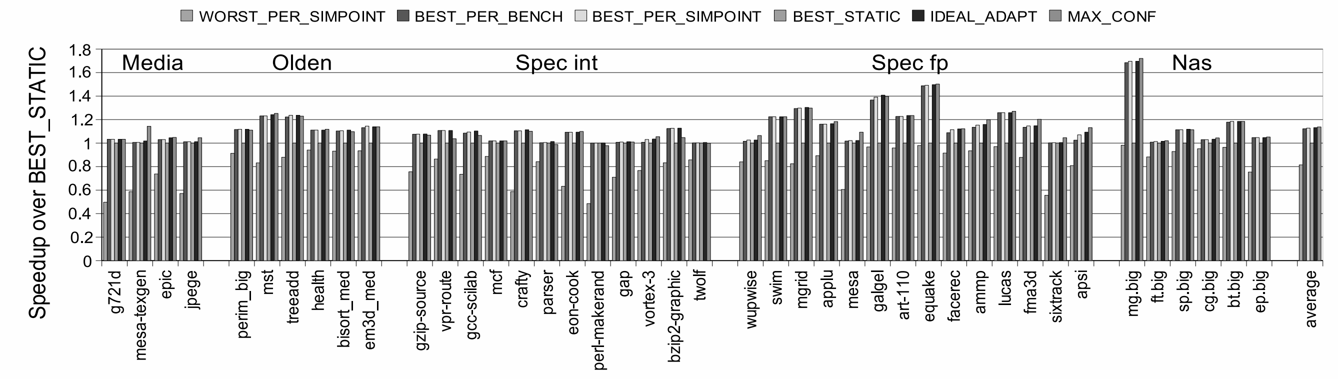

Figure 5: The performance of six configuration policies with a peak power constraint of 75% of the maximum configuration. The best static configuration is: iqs:32 fqs:32 ialu:4 falu:1 ldst:1 ics:16 dcs:16 ipr:64 fpr:64 rob:128

6.2 Realistic Adaptive Techniques

In order for the adaptation manager to approach the potential we have seen for our adaptive processor, it needs to constantly navigate the list of potential configurations in the config ROM, hopefully staying close to the optimal configuration at all times.

Specifically, the configuration manager has to address three issues: *When should a reconfiguration be triggered?* We should be attempting to reconfigure at least often enough to capture major program phase changes, as well as transitions between applications (context switches). *Which configuration from the ROM is the most appropriate to run next?* The ROM stores more than 100 configurations for each power chunk. Being able to distinguish the configurations that are more likely to perform better becomes critical. Finally, *how do we evaluate the configuration chosen?* For the latter we need a notion of whether performance has improved or declined as a result of a reconfiguration decision.

We examine several policies for effectively navigating the configuration space. Each is described as a 3-tuple, corresponding to the questions in the previous paragraph. Thus, the naming convention is < adaptation triggering mechanism >_< configuration selection method >_< evaluation method >. We show results for the following policies:

INTV_RANDOM_NONE: After a predetermined interval of cycles a different configuration is chosen randomly. There is no feedback mechanism or evaluation of the configurations. This is a simple policy and it is not expected to perform well. But it is a useful result in that it essentially gives us the "expected" behavior of the set of all the potentially good static architectures.

INTV_SCORE_NONE: After a predetermined interval of cycles, a different configuration is chosen according to a score-based technique. Statistics are kept for conflict events of each resource that is dynamically configured. A conflict occurs when a resource is being contended for. Specifically, we maintain instruction queue conflicts (an instruction cannot enter the instruction queue because it is full), floating point queue conflicts, integer register conflicts (an instruction cannot be renamed because there are no available renaming registers), floating point register conflicts, integer ALU conflicts, floating point ALU conflicts, reorder buffer conflicts, Icache misses, and Dcache misses. To have a notion of which resource is most required we maintain the ratio of conflicts per cycle (we also tried conflicts per instruction, with no significant performance difference observed). A vector is formed indicating the components that exceeded thresholds we set based on experimentation (weight_vector). Then, another vector is formed which describes how closely a configuration under consideration relates to the current configuration (closeness_vector) – this vector can have both positive and negative values. The closeness vector is simply the result of subtracting the two entries (the configuration under consideration from the current configuration) from the config ROM. For example if we consider 1,2,4,8 ways caches and the current configuration has 2 way-enabled cache while the configuration under consideration has 4, the element of the closeness vector for that

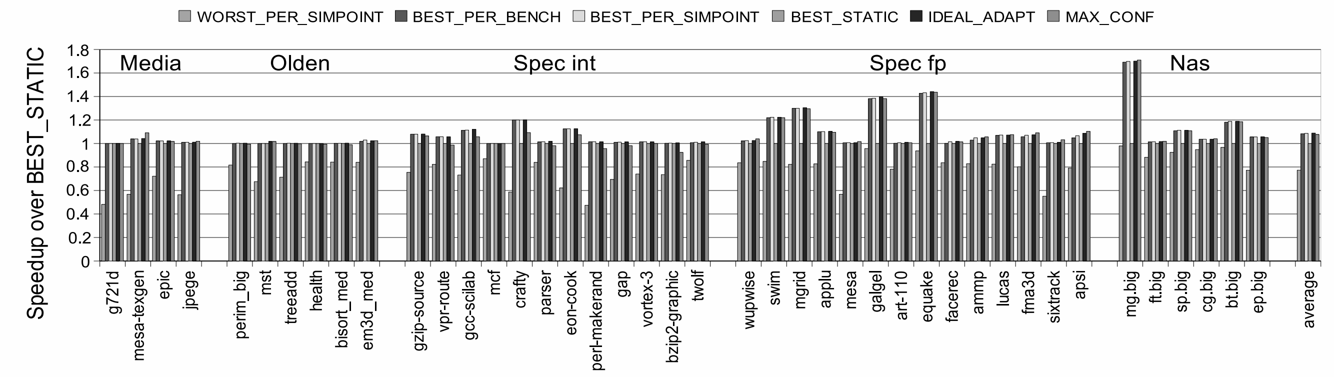

Figure 6: The performance of six configuration policies with a peak power constraint of 80% of the maximum configuration. The best static configuration is: iqs:32 fqs:32 ialu:4 falu:1 ldst:1 ics:16 dcs:16 ipr:128 fpr:64 rob:256

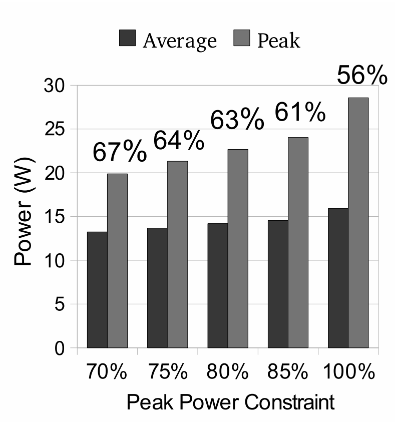

Figure 7: The average and peak power of our idealized adaptive core. Above the bars is shown the ratio of peak to average power.

cache would be 1. If instead of 4 there were 8 ways the closeness element would be 2. For a configuration with a 1-way cache, the closeness element would be -1. The total score for each configuration is found by multiplying the two vectors and summing the elements, as illustrated in Table 8.

Once all the scores are estimated, the configuration with the highest score is selected. There are frequent ties, and so the selection is then made randomly. This policy makes a more educated configuration selection by favoring configurations that increase the resources with conflicts. However, it still lacks a feedback mechanism to evaluate the configuration selected.

INTV_SCORE_SAMPLE: This is the same policy as the previous one with the addition of sampling as a feedback mechanism. Every interval of cycles a series of potential configurations are chosen based on the scoring system and the weights given by the execution of the previous configuration. The top n are chosen for sampling, again with random selection breaking ties. Experimentally, we found n=5 to work well. After all configurations are run for a sampling interval, the best IPC wins (the previous configuration's execution IPC is also considered) and that configuration runs until the next reconfiguration interval. The reconfiguration interval is set to keep a 1:10 ratio between the sampling time and the execution time. To prevent any transitional or cold start effects there is a warm up period before data is collected during a sampling interval.

EVDRIV_SCORE_SAMPLE: This policy does not consider adaptation periodically. Instead adaptations are considered only when there is evidence of a change in application behavior (event-driven). In particular, if we detect that over the previous interval, the measured IPC, or the cache behavior (misses per cycle), changed by a relative margin of 30%, then we initiate a reconfiguration evaluation. The frequency of adaptations is limited by an upper (8M cycles) and lower bound (2M cycles).

INTVAD_SCORE_SAMPLE: With this policy we adapt the interval between reconfiguration sampling based on how useful reconfiguration is. When we sample and find a better configuration than the previous, we cut the time to the next reconfiguration in half. When we fail to find a better configuration, we double the interval (adaptive interval). In this way, we search quickly when we are far from optimal, and minimize sampling overhead when we are near it. Again there is a minimum bound (2M cycles) and a maximum bound (8M cycles) for the interval.

In figure 8 we present the speedup over the best static configuration achieved with the different adaptive policies applied to the lowest peak power bound. We experimented with interval values ranging from 0.5M to 10M. The policies with fixed interval produced the best results with an interval of 2M cycles, while the policies with variable interval perform best with lower bound 2M and upper 8M cycles. We observe that randomly selecting a configuration from the ROM performs significantly worse than the best static configuration. This is the difference between the average static configuration and the optimal static configuration. With the addition of *score* to make more educated selections the performance is on average equivalent to the best static configuration. The evaluation that comes with sampling gives a major boost to performance since bad configurations are avoided. Finally, the dynamic interval techniques provide some additional gains – the adaptive interval doing better than the event-driven (phase detection) mechanism. The problem with the phase detection heuristic is that in its current form, when it encounters an abrupt phase change, it gets just one chance to find a good configuration before it settles in for a long interval, and adapts very slowly after that.

We see that with our best dynamic technique, we cover almost 10% out of the potential 15% speedup of the oracle adaptive technique over the best static. Overall with the best realistic adaptive approach we perform 10% worse than the maximally configured processor. Thus we have shown that we can reduce the peak power of a processor core by 30%, only sacrificing 10% in performance. The best static configuration with the same peak power would have a 20% performance hit. At 75% peak power, the performance loss is less than 5%.

	iq size	fq size	ialus	falus	ldst units	iregs	fregs	Icache ways	Dcache ways	rob
Current config	32	16	2	1	2	128	128	4	1	128
Considered config	16	16	4	3	1	64	128	1	2	128
Closeness vector	-1	0	1	2	-1	-1	0	-2	1	0
Weight vector	1	0	0	1	0	1	0	0	1	1
Results (cl_vec * wei_vec)	-1	0	0	2	0	-1	0	0	1	0
Total score (sum of results)	1									

Table 8: Example of configuration score estimation

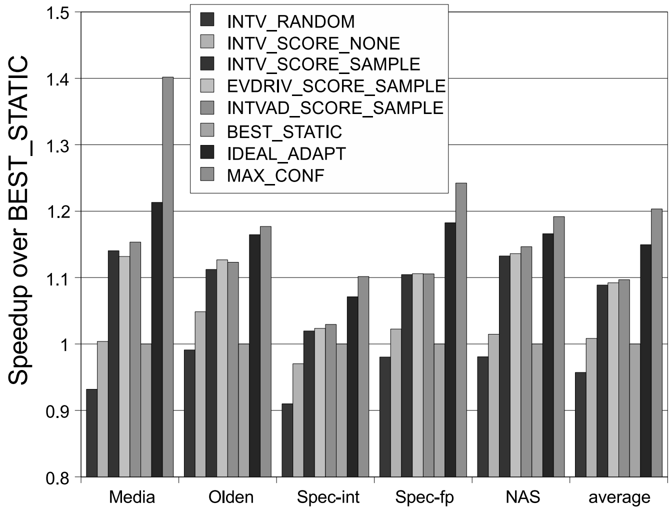

Figure 8: The performance of different adaptive policies with respect to the best static configuration

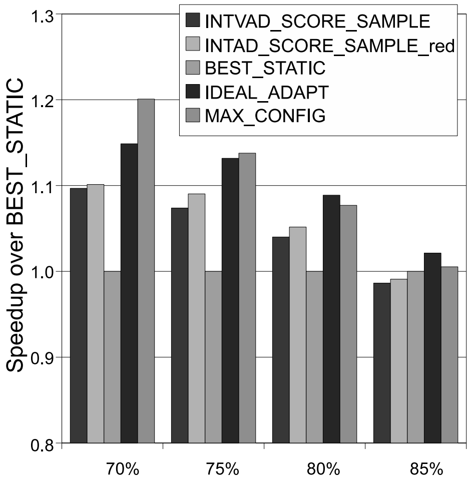

Figure 9: Comparison of full and reduced ROM for different peak power constraints

Relative power threshold	Number of configurations
0-70 %	71
0-75 %	97
0-80 %	119
0-85 %	116
0-inf %	1

Table 9: Configuration distribution of the reduced set over different peak power groups

6.3 Reducing ROM configurations

There are two costs to having a large number of configurations in the config ROM. One is storage, although that cost is relatively low. The second cost is the difficulty in finding the optimal configuration. We can further pare down the configurations experimentally, separating those often found to be worthwhile from those rarely selected. For this experiment, we do this using the oracle adaptive experiments presented in Section 6.1. We eliminate from the config ROM only those configurations *never* selected as the best configuration for even a single interval for any benchmark. This is a fairly conservative filter. Even still, we cut the number of configurations by more than half (see Table 9). Note that there is some danger to this optimization, if the actual workload is not well represented by the benchmark set used to pare down the configurations.

Figure 9 presents the best adaptive policy (INTVAD_SCORE_SAMPLE) with the full ROM and the reduced ROM for different peak power thresholds. As expected, reducing the number of configurations improves our performance, inching us even closer to the ideal result. At 75% with the inclusive set of configurations we perform 6.4% worse than the maximally configured core, while with the reduced set we do 4.8% worse. At 80%, our realistic adaptation policy with the reduced set of configurations is only 2.5% worse than MAX_CONF. In many scenarios, trading 5% performance for a 25% reduction in peak power, or 2.5% for a 20% reduction, and the cost and real power savings that go with it, represents a huge win.

It is worth noting that DVFS, proposed for peak power budgeting, sees more than double the performance hit when achieving similar peak power savings (Table 1 in [24]).

6.4 Delay sensitivity analysis

This section examines the impact of the assumed reconfiguration latency on our results. The adaptive policy used was INTVAD_SCORE_SAMPLE. To change the duration of adaptation we multiplied all the parameters of Table 4 with different factors as demonstrated in figure 10. The "No delay" bar corresponds to the unrealistic scenario where component powergating happens instantly – note that adaptation still is not instant since we wait for instruction queues to drain or dirty L1 entries to be flushed to the L2. For any reasonable assumption about these delays, the impact on performance is negligible. Undoubtedly, the adaptive interval optimization contributes to this, since on average the interval lengths were larger than the rest of the techniques.

6.5 Quantifying the benefits of Peak Power reduction

In this section, we analyze the impact of the peak power reduction on silicon area required for on-chip decoupling capacitors and the voltage variation. The voltage variation in sub-65nm technologies should be kept below 5% of the nominal supply voltage

	Experiment 1	Experiment 2
Relative power threshold	On-chip decap (% of Core Area)	Max. Voltage Variation (% V_{DD})
0-70 %	9%	4.48%
0-75 %	9.7%	4.80%
0-80 %	10.5%	5.12%
0-85 %	11.5%	5.44%
0-inf %	15%	6.48%

Table 10: Peak power reduction impact on the voltage variation and the on-chip decoupling capacitor area

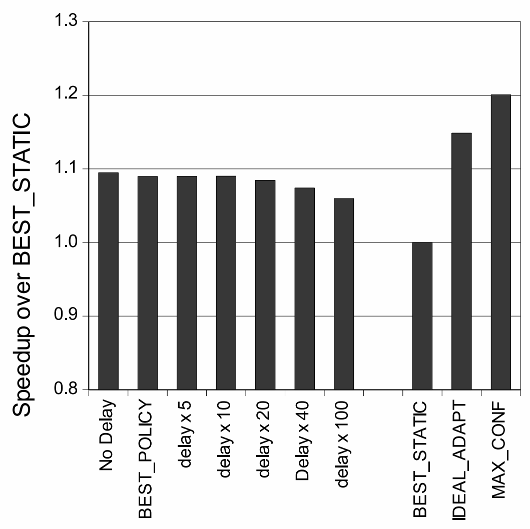

Figure 10: Sensitivity analysis of INTVAD_SCORE_SAMPLE to the delay of adaptation

V_{DD} [3]. We model a distributed power distribution network with power and ground network RLCG parasitics based on [34, 27] for the total processor die area of $26.25mm^2$ as shown in Figure 11. The core current demand is modeled based on the peak current demand – peak power divided by the supply voltage – and the clock frequency and is distributed as time-variant current sources over the power and ground mesh. The distributed current sources are modeled as triangular waveforms with the clock frequency period ($\frac{1}{f_{clk}=1.83GHz}$) and the rise and fall time of $10 \times f_{clk}$ [38, 40].

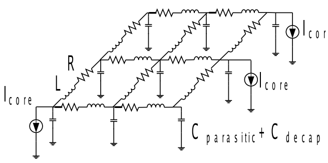

Figure 11: Distributed power distribution network model.

To calculate the amount of necessary decoupling capacitances required to maintain a voltage fluctuation within 5%, we use the following first order approximation: $P = C_T \cdot {V_{DD}}^2 \cdot f \cdot p_{0-1}$ where P is the total chip peak power consumption, V_{DD} denotes the supply voltage, f is the clock frequency, C_T is the on-chip decoupling capacitance, and p_{0-1} is the probability that a $0-1$ transition occurs [34]. C_T includes the intrinsic decoupling capacitance which is usually small and the intentional decoupling capacitance which we add and for which we report area numbers [29].

We find that peak power reduction saves significant die area for the intentional on-chip decoupling capacitors and minimizes the voltage drop at the same time. We perform the transient analysis based on [34] to find the minimum on-chip decoupling capacitors which are required based on 65-nm thin oxide technology from the initial estimation. Table 10 illustrates the experimental data for the on-chip decoupling capacitor estimation. We ran two different analyses. Experiment 1 illustrates that to maintain voltage variation below 5% of the nominal V_{DD}, the reduction of peak power to 75% of the original value reduces the amount of required on-chip decoupling capacitor area by 5.3% (of total core area). Experiment 2 demonstrates that if we instead maintain the same amount of on-chip decoupling capacitors (in this test case $150nF$) the voltage variation is suppressed significantly when we reduce the peak power. For the 75% threshold, the variation is reduced by 26%.

7. CONCLUSION

This paper describes an adaptive processor that uses table-driven reconfiguration to place a cap on peak core power dissipation. By only allowing configurations specified in this table, the core is ensured to always operate below a given power threshold. While it is impossible to configure all resources at full capacity, each application can find a configuration that provides the resources it most needs at full capacity. In this way, we minimize the performance loss while providing significant peak power savings. This technique is shown to enable a 25% reduction in peak power with less than a 5% performance cost. Additionally, it can outperform the best static design at the same peak power threshold by 9%. The Peak power reduction translates to 5.3% less total silicon for decoupling capacitance or a 26% reduction in voltage variation for the same decoupling capacitance. Furthermore, the design of the chip voltage supply becomes cheaper and more efficient.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonmymous reviewers for many useful suggestions, and Jeffery Brown for his help and advice with the simulator. This research was supported in part by NSF Grant CCF-0702349 and grants from Intel Corporation.

9. REFERENCES

[1] D. H. Albonesi. Selective cache-ways: On demand cache resource allocation. In *Proc. of MICRO*, 1999.

[2] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 2003.

[3] B. Amelifard and M. Pedram. Optimal selection of voltage regulator modules in a power delivery network. In *Proc. of DAC*, 2007.

[4] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proc. of HPCA*, 2001.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. of ISCA*, 2000.

[6] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proc. of GLSVLSI*, 2001.

[7] A. Buyuktosunoglu, T. Karkhanis, D. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *Proc. of ISCA*, 2003.

[8] Y.-S. Chang, S. K. Gupta, and M. A. Breuer. Analysis of ground bounce in deep sub-micron circuits. In *Proc. of VTS*, 1997.

[9] A. K. Coskun, R. Strong, D. M. Tullsen, and T. Simunic Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proc. of SIGMETRICS*, 2009.

[10] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. Technical report, Univ. of Rochester, 2002.

[11] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proc. of ISCA*, 2002.

[12] E. Grochowski, D. Ayers, and V. Tiwari. Microarchitectural di/dt control. In *Proc. of IEEE Design and Test*, 2003.

[13] A. Grove. IEDM 2002 Keynote Luncheon Speech.

[14] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proc. of ISLPED*, 2004.

[15] Intel Corp. *Intel Pentium 4 Processor in the 423-pin Package Thermal Design Guidelines*, Nov. 2000.

[16] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proc. of MICRO*, 2006.

[17] ITRS. *International Technology Roadmap for Semiconductors 2003, http://public.itrs.net*.

[18] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proc. of ISCA*, 2001.

[19] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *Proc. of MICRO*, 2003.

[20] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessing. *IEEE Computer*, 2005.

[21] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proc. of PACT*, 2006.

[22] B. C. Lee and D. Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *Proc. of ASPLOS*, 2008.

[23] R. Maro, Y. Bai, and R. I. Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Proc. of PACS*, 2001.

[24] K. Meng, R. Joseph, R. P. Dick, and L. Shang. Multi-optimization power management for chip multiprocessors. In *Proc. of PACT*, 2008.

[25] P. Muthana, A. Engin, M. Swaminathan, R. Tummala, V. Sundaram, B. Wiedenman, D. Amey, K. Dietz, and S. Banerji. Design, modeling, and characterization of embedded capacitor networks for core decoupling in the package. *Trans. on Advanced Packaging*, 2007.

[26] K. Najeeb, V. V. R. Konda, S. K. S. Hari, V. Kamakoti, and V. M. Vedula. Power virus generation using behavioral models of circuits. In *Proc. of VTS*, 2007.

[27] V. Pandit and W. H. Ryu. Multi-ghz modeling and characterization of on-chip power delivery network. In *Proc. of EPEP*, Oct. 2008.

[28] M. D. Pant, P. Pant, D. S. Wills, and V. Tiwari. Inductive noise reduction at the architectural level. In *Proc. of VLSID*, 2000.

[29] M. Popovich, A. V. Mezhiba, and E. G. Friedman. *Power Distribution Networks with On-Chip Decoupling Capacitors*. Springer, 2008.

[30] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proc. of MICRO*, 2001.

[31] P. Royannez, H. Mair, F. Dahan, M. Wagner, M. Streeter, L. Bouetel, J. Blasquez, H. Clasen, G. Semino, J. Dong, D. Scott, B. Pitts, C. Raibaut, and U. Ko. 90nm low leakage soc design techniques for wireless applications. In *Proc. of ISSCC*, 2005.

[32] J. Sartori and R. Kumar. Distributed peak power management for many-core architectures. In *Proc. of DATE*, Mar. 2009.

[33] J. Sartori and R. Kumar. Three scalable approaches to improving many-core throughput for a given peak power budget. In *Proc. of hiPC*, Dec. 2009.

[34] A. Shayan, X. Hu, H. Peng, W. Yu, W. Zhang, C.-K. Cheng, M. Popovich, X. Chen, L. Chua-Eaon, and X. Kong. Parallel flow to analyze the impact of the voltage regulator model in nanoscale power distribution network. In *Proc. of ISQED*, 2009.

[35] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. of ASPLOS*, Oct. 2002.

[36] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P.Jouppi. Tech report CACTI 5.1. Technical report, HPL, 2008.

[37] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proc. of CMG Conference*, 1996.

[38] G. Unikowsky. Allocating decoupling capacitors to reduce simultaneous switching noise on chips. *MIT PhD Thesis*, 2004.

[39] S. Yaldiz, A. Demir, S. Tasiran, Y. Leblebici, and P. Ienne. Characterizing and Exploiting Task-Load Variability and Correlation for Energy Management in Multi-Core Systems. In *Proc. of Workshop ESTIMedia*, 2005.

[40] H. Yu, C. Chu, and L. He. Off-chip decoupling capacitor allocation for chip package co-design. In *Proc. of DAC*, 2007.

[41] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proc. of DAC*, 2002.

[42] X. Zhou, P.-L. Wong, P. Xu, F. Lee, and A. Huang. Investigation of candidate VRM topologies for future microprocessors. *Trans. on Power Electronics*, Nov 2000.