Power

Kevin Bowers & Jordan Wales

While it's a little hard to quantify the effect of cosmic ray strikes causing transient faults in computer systems, power is something most of us care about. Whether it's your phone, PDA, laptop or embedded microcontroller, we want our devices to last longer, which means either a bigger battery, or less power waste. There are two basic approaches to saving power. The first involves reducing the amount of waste in the cache, while the second deals with other techniques, like slowing down the processor or turning components off.

Part I: Cache

A lot of power is wasted in the cache. 33% of all power is in the cache on a Pentium Pro, and even in the StrongARM chip, the cache consumes 16% of the power used. There have been two ways to reduce this power usage. The first is to try and stop the power leakage from the transistors in the cache, while the second involves the power needed to get information out of the cache.

Transistors don't hold charge for extended periods of time. This leads to the need for refreshes in order to maintain state. At any one time, no more than 10% of the cache is actually being used. The rest is sitting there waiting to be used, but not currently being accessed. This leads to two approaches for decreasing power. First, shut off the section that isn't used, and then read the data back in from the L2 cache when and if it is needed. This works to some extent, and with smart prefecting is able to achieve a reduction in energy-delay of 62% at a 4% cost in execution time. The second approach is to put the caches to sleep, a state in which they can't be read, but data is not lost and they only consume 25% of the power. To recover data, instead of pulling from the L2 cache, the cache just needs to be activated before any data is read out. Using this technique can lead to a reduction in cache power of 54% with a 1% cost in execution time.

The second approach to saving power in the cache is to reduce the amount of power used to get information out of the cache. In a 4-way associative cache, getting data out requires reading 4 lines and discarding the 3 that you don't want. This is a waste of 75% of the power. If something could be done to eliminate this, we could save a lot. There are three basic ways to accomplish this. First, predict which way you are going to access, and only read it out. This works very well for instruction caches. Secondly, data that is used often can be moved to special direct mapped cache space, allowing for faster lookup without the wasted power as it is the only data at that location. This works well for data caches. Combining the two leads to a 65% reduction in cache power, 8% overall reduction, at a cost of only 3% performance.

The other option is to use the compiler to check for often used data/instruction values, and hint that they should be stored in direct-access cache. Doing the modification at compile time allows for more information to be taken into account when making the decision, which usually leads to better decisions. A valid bit on the cache ensures the

contents weren't modified by another process, and the memory can be accessed without all the waste in power. This technique can save 8% overall power, the same as way-prediction above, but without the performance penalty, though you have to recompile.

Part II: Chip and Architecture

This section reviews three papers. We shall briefly describe each of them in turn, in short sentences taking almost the form of bulleted points.

TEMPERATURE AWARE MICROARCHITECTURES

- K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranaranyanan, D. Tarjan, " Temperature-Aware Microarchitecture", in ISCA, June 2003.

Motivations

Heat damage makes cooling a necessity. *Rising power density* and *rising cooling costs* make temperature-regulation by architectural devices an attractive substitute for active cooling, for active cooling just uses more power. Therefore, low-power and portable devices must be cooled by other means.

The authors choose an *"architecture-level" solution*, meaning that their solution deals differently with different aspects of the architecture; they customize various methods to work for the regions to which they are applied. A "chip-level" solution responds more globally to temperature increases. Rather than compensating by altering functioning, many chip-level solutions (Pentium 4) halt execution until cool enough to resume. An "architecture-level" solution will run the architecture in such a way as to compensate for heat without having to always stop. Moreover, an architecture-level simulation will better reveal heating gradients, which are not manifest in a global heating model.

Simulation

Simulations will enable the authors to *customize their solutions* to the specific heating problems of individual architecture. However, simulations can be expensive. Therefore, the authors develop a quick-and-dirty (though not too dirty) simulation to analyze heat transfer and hot-spots in a computer chip. The goal is to have a simulation that is *accurate, efficient, and portable, and flexible* enough to accommodate strange new architectures for rapid simulated prototyping.

The simulation they produce utilizes R-C (resistance-capacitance) simulation to model the transfer of heat from one heat-absorbing area (capacitance), through a transmissive connection (resistance) to another (capacitance). The simulator was verified in several test-cases with a comparison to a rigorous Finite Element Analysis (FEA) simulation (< 5.8% error).

Were the test cases that the authors compared to FEA sufficient to inspire confidence? Is the margin of error (less than 5.8%) something that we can tolerate when attempting high-performance design? We might say yes to both of these because any design that is rapidly sketched in the HotSpot simulation can easily be ported to FloWorks (the more

intense sim) for verification. Note that the "fitting factor" calibration for thermal capacitances in silicon was determined for HotSpot by FloWorks.

We might further ask: Is the "fitting factor" something that must be re-determined every time a chip is designed? The answer is, probably, no – as long as materials and basic design remain the same.
The simulation does have several limitations. The simulation *does not account for certain contributions* to heating, such as the clock grid and other interconnections. Future simulations may separate the self-heating of the wire from the heat transfer to the silicon. This requires further investigation to determine an accurate modeling method. Simulation *assumes even heating of element bottom surfaces*. This is not realistic. The simulation should eventually account for heating effects of wires that pass through inactive areas.

The simulation was intended to find the hottest parts of the chip (and the portions where temperature gradients are most severe) to identify those requiring remediation. Modeling was done for a layout similar to that of the Alpha 21364. However, *certain elements, such as multi-processor interface logic, were not modeled* because they are not involved in the study. Others, such as I/O pads, cannot be adequately modeled and are therefore treated as L2 Cache.

Results

The L2 Cache, despite heavy use, was substantially cool, owing to its wider area from which heat could be more easily dispersed. The *ALU* and the *Integer Register File* were the principal hot-spots.

Remediation Methods Employed

(GLOBAL) *Dynamic Frequency Scaling* – As temperature rises, conduction changes. Frequency scaling must take this into account. Therefore, Temperature-Sensitive Frequency Scaling is employed whereby the Frequency is decreased as temperature increases. Given the way conduction properties change at high temperature, this frequency reduction enables the chip to run, without incurring timing errors, slightly hotter than would otherwise be permissible. Every frequency change requires a stall of 10-50 micro-seconds, but it is believed that future architectures will be able to continue execution with a gradual change in frequency. The critical factor here is that frequent changing of clock-speed will result in much wasted time (and energy) in stalling.

(GLOBAL) *Dynamic Voltage Scaling* – Unlike frequency scaling, this technique reduces power usage (and therefore temperature) throughout the architecture. As with frequency scaling, the increments are set such that stall-time will be minimized (by minimizing the number of scale operations). Again, it is anticipated that future architectures will be able to continue execution through gradual changes. A PI controller is used to respond dynamically to temperature changes. A low-pass filter is used to avoid "fluttering" near the boundary of set-points.

(LOCAL) *Localized Toggling* – Stall unused components of superscalar architecture, which has been divided into "engines" for this purpose: fetch, integer, floating point, and load/store. This method, in allowing execution to continue, is superior to "chip-level" solutions that halt execution entirely to cool off. When stalling unused components is no longer sufficient, dynamic voltage scaling will give the system a rest in which to cool off.

(LOCAL) *Computation Migration* – When the integer register file becomes too hot, its contents are copied to a slower file further from the chip. This permits a cool-off for the primary file while maintaining execution.

(GLOBAL) *Global Clock Gating* – PI-controlled stalling of the entire chip (as referenced above) is found to under-perform when compared to Localized Toggling. The unnecessary waste of a global stall is not a factor in local gating.

-

Fabrication Process Adjustment (Chip Layout)'

Control and Sensing Issues

PROBLEM: If sensors improperly placed, they may fail to activate compensation.

PROBLEM: If thresholds set incorrectly, migration may occur too often, a sort of thrashing between register files.

Sensor positioning can be difficult – we assume that we are properly reading temperature, but we may be reading an assumed hot-spot that is substantially cooler than the actual hot spot. Therefore, we must balance this error by *setting lower thresholds* without, however, interfering too much in the operation of the chip. We might recall that the dynamic frequency scaling, which permits some amount of "hot" operation, could be of assistance in this case, allowing us some room for comfort even if we are potentially permitting a hot-spot to go unremediated.

LOCAL AND GLOBAL HARDWARE ADAPTATIONS

- R. Sasanka, C. J. Hughes, S. V. Adve "*Joint Local and Global Hardware Adaptations for Energy*", in ASPLOS, October 2002.

Motivations

Here we confront a different issue – now we wish specifically to save power rather than reduce temperature. Power saving has obvious application from large-scale parallel computing to tiny battery-operated devices. Neither do you wish to construct a power substation for your new sever farm nor do you wish your battery to lose its charge after entering an appointment into your Palm Pilot.

Our goal is to save energy rather than averting physical damage. Problem sites include unused superscalar elements, which lose energy by transistor "leakage." Moreover, multimedia applications have "dead" time; the ALU inactive while the calculated frame

is being delivered to RAM, system, etc.

The authors ask two questions:

When (temporal granularity)?

Where (spatial granularity)?

As with temperature compensation, we will always be looking at this "local/global" question while determining remediation methods.

Solutions

(GLOBAL) DVS-Mediated Frequency switching…
A Global adaptation for general use, this method can slow the execution of a real-time app while still meeting the deadline for "just-in-time" execution.

(LOCAL) *Local adaptations for multimedia use.*

Using intra-frame execution variability, the architecture dynamically adapts the **instruction window size** to *maximize throughput* by loading as many instructions as can be executed quickly, while *minimizing stalling* and *minimizing wasted power* by loading no more than can be executed.

Also, the architecture is designed to turn off *unused* **superscalar elements** so that no power will be lost due to leakage. This is similar to the "temperature-based toggling" of the temperature-control paper.

Simulations determined that there was a significant power savings using local adaptations *only* when performing multimedia operations. Everything else benefited best from the global frequency switching mediated by voltage scaling.

Note that this paper does not use computation migration, for that technique only relieves hot areas; it does not save power but actually drains a little more power in the transfer of information from one part of the chip to another.

VISA FOR REAL-TIME SYSTEMS

- A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, F. Mueller, "*Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems*", in ISCA, June 2003.

Motivations

Time-critical applications require guarantee that CPU will complete operations on time. However, high-performance CPUs are too complex to permit calculation of Worst Case Execution Time (WCET).

If we can't use advanced CPUs, we will waste time (slower CPUs) and waste energy (less efficient CPUs).

If we can use advanced CPUs, we will save time (finish time-critical apps faster) and we may use the extra time for lower-priority applications or save power (use advanced efficiency improvements). There is a trade-off between using slack time for other apps and using slack time for power-saving.

If we run on an advanced chip with *guaranteed* execution time, we can run time-critical apps on a non-dedicated system. With an advanced CPU, we can run these applications and still have other threads in the background. Otherwise we might have to dedicate a tractable CPU just to the critical thread.

Solution

Encode a VISA "front-end" such that the architecture can emulate an architecture with guaranteed WCET. Run on the advanced architecture for as long as we are "ahead of time." To calculate progress, the authors recommend that the programmer divide each task into sub-tasks.

Set check-point dead-lines for sub-task completion. If a check-point is missed, drop back to VISA emulation for the remainder of execution.

- (Check-point for *completion* of instruction OP) = (time at which OP must *start* in the simple pipeline to guarantee completion by app dead-line)
- Therefore…
- If a check-point is missed we will still be slightly ahead of the game if we have completed even one stage of OP.

ProblemsFirst, keep in mind that this system is always better than using the simple pipe from the beginning. But…Setting check-points for completion at the *start* time in the simple pipeline is our guarantee… *Therefore*…If we miss one checkpoint we cannot necessarily ever return to advanced execution. To get around this apparent problem, we might watch to see if we have "gained" on our time, and step-up to the advanced architecture if we are again ahead-of-schedule.

Advantages of VISA

The unsafe advanced architecture need not be analyzed for us to use it. Using the VISA checkpoint system we can guarantee on-time execution. We achieve *higher throughput* – low-priority threads can exploit the slack time. We achieve *lower power* – dyanmic frequency switching can provide "just-in-time" execution on the fast architecture. We are afforded the opportunity to endorse such a product with *ridiculous advertisements* – people see pretty women hawking wares all the time but it still seems to work. Simple architectures indeed!

CautionAll of this is based on simulation…

- But we have to start somewhere…
- We can guarantee this architecture will work but we can't guarantee the precise power savings.

A BRIEF SUMMARY OF CHIP AND ARCHITECTURE-LEVEL SOLUTIONS

You may refer to the slides to see a summary of all the techniques employed. What that slides does not reveal, however, is that each of these attacked a slightly-different facet of the general problem of *efficiency* in execution.

We might, in fact, combine them all: We can incorporate *temperature regulation* using the basic simulation and scaling principles of the first paper. We can also utilize the *power-saving* devices of the second paper. Both of these contributions are compatible but will render the chip somewhat…complex. Therefore, we can slap a VISA front-end onto it so that we can guarantee time-critical execution reliability for the chip. With a blatant and crass advertising campaign, we can guarantee our dominance in the realiable low-power high-performance chip market.

As always, the tradeoffs are local/global compensation/computation (the more you calculate for compensation, the more computation you are ironically wasting), and simulation/implementation (simulations can be used to get a design started but only a prototype will demonstrate the true worth of the product). Another tradeoff not really mentioned is that of complexity: if these systems become so complex, they will be much less fault tolerant. Any system incorporating power-saving devices and temperature sensors will, as its design evolves, become entangled in those improvements to the point that it begins to rely on them. At the moment, the temperature-controlling design could fail and the chip would not break; it would just be less reliable at high temperature. However, if we combine all of these into one architecture, there is a great likelihood that we will lose some of the independence of these methods – shortcuts will be taken and, then, the failure of one sub-system or component could be fatal to the chip whereas, under the *same* program load, a different chip without all those fancy features might achieve equal performance without failing. This is conjectural, but something to think about.