A Hardware Architecture for Dynamic Performance and Energy Adaptation

Phillip Stanley-Marbell
Dept. of ECE
Carnegie Mellon University
Pittsburgh, PA 51213

Michael S. Hsiao
Dept. of ECE
Virginia Tech
Blacksburg, VA 24061

Ulrich Kremer
Dept. of Computer Science
Rutgers University
Piscataway, NJ 08854

ABSTRACT

Energy consumption of any component in a system may sometimes constitute just a small percentage of that of the overall system, making it necessary to address the issue of energy efficiency across the entire range of system components, from memory, to the CPU, to peripherals. Presented is a hardware architecture for detecting regions of application execution at runtime, for which there is opportunity to run a device at a slightly lower performance level, by reducing the operating frequency and voltage, to save energy. The proposed architecture, the Power Adaptation Unit (PAU) may be used to control the operating voltage of various system components, ranging from the CPU core, to memory and peripherals.

An evaluation of the tradeoffs in performance versus energy savings and hardware cost of the PAU is conducted, along with results on its efficacy for a set of benchmarks. It is shown that on the average, a single entry PAU provides energy savings of 27%, with a corresponding performance degradation of 0.75% for the SPEC CPU 2000 integer and floating point benchmarks investigated.

Keywords

Computer Architecture, Low Power, Dynamic Voltage Scaling, Memory-bound

1. INTRODUCTION

Reduction of the overall energy usage and per-cycle power consumption in microprocessors is becoming increasingly important as device integration increases. This leads to increases in the energy density of generated heat, which creates problems in reliability and packaging. Increased energy usage is likewise undesirable in applications with limited energy resources, such as mobile battery powered applications.

Reduction in microprocessor energy consumption can be achieved through many means, from altering the transistor level design and manufacturing process to consume less power per device, to modification of the processor microarchitecture to reduce energy consumption. It is necessary to address the issue of energy efficiency across the entire range of system components, from memory, to the CPU, to peripherals, since the CPU energy consumption may sometimes constitute a small percentage of that of the complete system. It is no longer sufficient for systems to be *low power*, but they must also be *energy aware*, adapting to application behavior and user requirements.

In applications in which there is an imbalance between the amount of computation and the time spent waiting for memory, it is possible to reduce the operating frequency and voltage of the CPU, memory, or peripherals, to reduce energy consumption at the cost of a tolerable performance penalty. Previous studies have shown that there is significant opportunity for slowing down system components such as the CPU without incurring significant overall performance penalties [8, 7].

Compiler approaches rely on static analyses to predict program behavior. In many cases, static information may not be accurate enough to take full advantage of program optimization opportunities. However, static analyses often have a more global view of overall program structure, allowing coarse-grain program transformations, in order to enable further optimizations at the fine-grain levels by the hardware. Hardware approaches are often complementary to compiler-based approaches such as [8]. The window of instructions that is seen by hardware may not always be large enough to make voltage and frequency scaling feasible. However, even though only the compiler can potentially have a complete view of the entire program structure, only the hardware has knowledge of runtime program behavior.

Presented in this paper is a hardware architecture for detecting regions of application execution at runtime, for which there is the possibility to run a device (e.g. CPU core) at a bounded decrease in performance while obtaining a significant decrease in per-cycle power and overall energy consumption. The proposed architecture, the Power Adaptation Unit (PAU), appropriately sets the operating voltage and frequency of the device, to reduce the power dissipation and to conserve energy, while not incurring more than a prescribed performance penalty. The PAU attempts to effectively identify such dynamic program regions, and to determine when it would be beneficial to perform voltage and frequency scaling, given the inherent overheads.

Because of the type of behavior the PAU captures, even a small single entry PAU is effective in reducing power consumption under bounded performance penalty, for the benchmark applications investigated. The additional hardware overhead due to the PAU is minimized due to the fact that a majority of the facilities it relies on (e.g. performance counters) are already integrated into contemporary processor designs. The overhead due to maintaining PAU state is shown to be small, and proposals are provided for using existing hardware to implement other functionality required

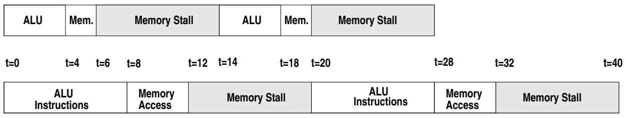

Figure 1: Opportunities for energy/performance tradeoff in a single-issue architecture

by the PAU.

The remainder of this paper is structured as follows. The next section describes opportunities for implementing dynamic resource scaling. Section 3 details the structure of the PAU architecture, and describes how entries are created and managed in the PAU. Section 4 illustrates the action of the PAU with an example. Section 5 discusses the analytical limits of utility of the PAU. Section 6 discusses the hardware overheads of the PAU. Sections 7 presents simulation results for a number of the SPEC CPU 2000 benchmarks. Section 8 discusses related work and Section 9 concludes the paper.

2. OPPORTUNITIES FOR SCALING

In statically scheduled single-issue processors, any decrease in the operating voltage or frequency will lead to longer execution times. However, if the application being executed is memory-bound in nature (i.e. it has a significant number of memory accesses, which cause cache misses), the processor may spend most of its time waiting for memory.

In such memory-bound applications, if the processor is run at a reduced operating voltage and memory remains at-speed, the portions of the runtime of a program performing computation (few) will take more time, while the portions of the runtime performing memory stalls (many) will remain the same, as illustrated in Figure 1. As illustrated in the figure, halving the operating voltage and frequency of the CPU while keeping that of memory constant can result in ideal case energy savings of 87.5% , with a 43% degradation in performance for the example scenario. In practice, this can only be approximately achieved, as there exist dependencies between the operating frequency of the CPU core and that of memory (one usually runs at a multiple of the frequency of the other).

Dynamically scheduled multiple issue (superscalar) architectures will permit the overlapping of computation and memory stalls, and would witness a smaller slowdown if the CPU core were run at a lower voltage while the operating voltage of memory were kept constant. This initial work focuses on single-issue in-order processors, such as those typically employed in low power embedded systems. The benefits to superscalar architectures will be pursued in future work.

3. POWER ADAPTATION UNIT

The power adaptation unit (PAU) is a hardware structure which detects regions of program execution with imbalance in memory and CPU activity, such as code execution that leads to frequent repeated memory stalls (e.g. memory-bound loop computations), or regions of execution that lead to significant CPU activity with little memory activity (e.g. CPU-bound loop computations). In both cases, the PAU outputs control signals suitable for adjusting the operating frequency and voltage of the unit it monitors, to

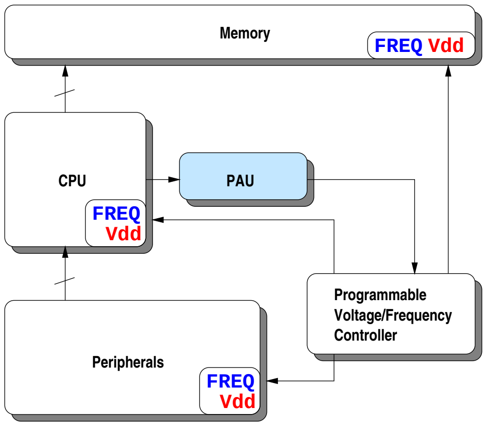

Figure 2: Typical implementation of PAU

values such that less power is consumed per cycle, while still maintaining similar performance. The PAU must determine an appropriate voltage and frequency at which to run the device it controls, for which a specified performance penalty (say, 1%) will not be exceeded.

The PAU in a typical system architecture is shown in Figure 2. The next two subsection focus on controlling the CPU for memory-bound applications and extending these ideas to controlling the cache and memory in CPU-bound applications respectively.

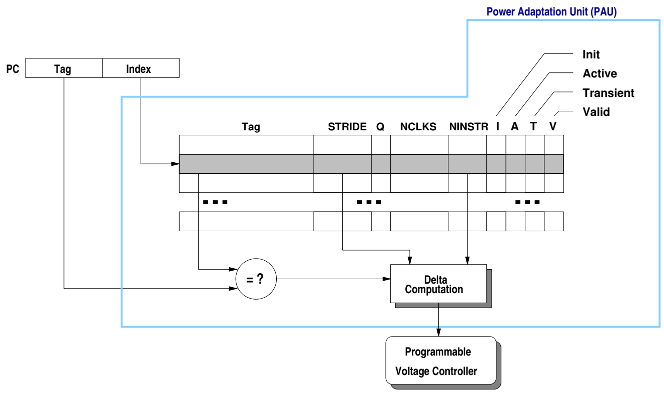

Figure 3: The Power Adaptation Unit

3.1 PAU Table Entry Management

The primary component of the PAU is the PAU table. Figure 3 illustrates the construction of the PAU table for a direct mapped configuration. The least significant bits of the program counter, the *index*, are used to select a PAU table entry. A hit in the PAU occurs when there is a match between the *Tag* from the PC (the most significant bits before the *index*) and the Tag in the PAU entry, as illustrated in Figure 3.

The PAU operates on *windows*, which are ranges of program counter values in the dynamic instruction stream. Windows are defined by a starting PC value, a *stride* in clock cycles, `STRIDE`, and a *count in overall instructions executed*, `NINSTRS`. An entry corresponding to a window is created on an event such as a cache miss. The `STRIDE` field is a distance

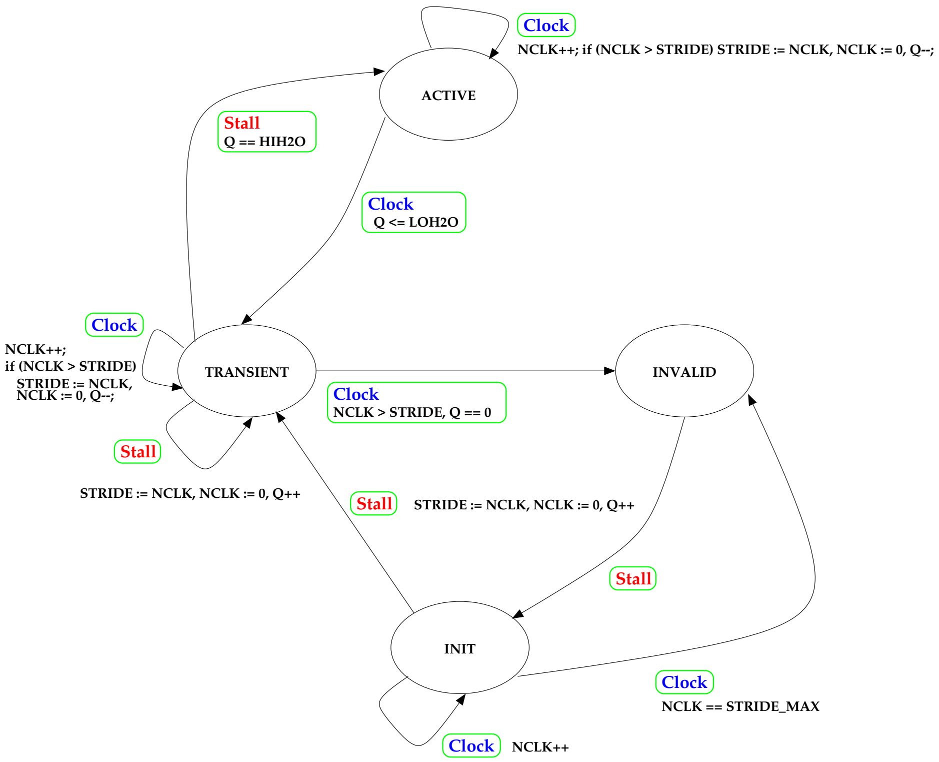

Figure 4: PAU entry state transition diagram

vector [3, 13], specifying the distance between the occurrence of events in the iteration space. The `NCLKS` field maintains the age of the entry symbolizing a window. Four state bits, `INIT`, `TRANSIENT`, `ACTIVE` and `VALID`, are used by the PAU to manage entries. The `Q` field is a saturating counter that provides an indicator of the degree of confidence to be attached to the particular PAU entry.

Figure 4 shows the state transition diagram for a PAU entry. There are four states in which an entry may exist, `INIT`, `TRANSIENT`, `ACTIVE` and `VALID` corresponding to the state bits in the PAU entry described previously. Transitions between states occur either when there is a pipeline stall due to a cache miss, or may be induced by the passage of a clock cycle. The two extremes of the state machine are the `INVALID` and `ACTIVE` states, with the `INIT` and `TRANSIENT` states providing hysteresis between these two states. In the figure, the transitions between states are labeled with both the event causing the transition (circled) and the action performed in the transition. For example, the transition between the `TRANSIENT` and `INVALID` states occurs on the passage of a clock cycle if NCLK is greater than STRIDE, and Q is zero.

Entries created in the PAU table are initially in the `INIT` state, and move to the `TRANSIENT` state when there is a stall caused by the instruction which maps to the entry in question. On every clock cycle, the `NCLKS` field of all valid entries are incremented, on the faith that the entries will be used in the future.

For all valid entries, the `NCLKS` field is reset to 0 on a stall, after copying it to the `STRIDE` field, and incrementing `Q`. The number of instructions that completed in that period is then recorded in the `NINSTRS` field for that PAU entry.

The goal of a PAU entry is to track PC values for which repeated stalls occur. The PAU will effectively track such cases even if the time between stalls is not constant. Whenever the `NCLKS` field for a `TRANSIENT` or `ACTIVE` entry reaches the value of the `STRIDE` field, it is reset to zero and the Q field decremented, therefore if the distance between stalls decreases monotonically, the `STRIDE` will be correctly updated with the new iteration distance.

If on the other hand the time between stalls increases, then one of two scenarios arises. If the number of iterations for which the distance between stalls is increased is large, then the entry will eventually be invalidated, and then recreated in the table with the new `STRIDE`. This purpose is served by the high and low water marks (`HIH2O` and `LOH2O`).

High and low water marks, determine how many repeated stalls occur before an entry is considered `ACTIVE`, and how many clock cycles must elapse before the determination is made to degrade an entry from `ACTIVE` status, respectively. The values of `HIH2O` and `LOH2O` may either be hard-coded in the architecture, or may be modified by software such as an operating system, or by applications, as a result of code appropriately inserted by a compiler. It is also possible to have the values of `HIH2O` and `LOH2O` adapt to application performance, their values being controlled using the information that is already summarized in the PAU, with a minimal amount of additional logic. Such techniques are beyond the scope of this paper and are left for future research.

Once `Q` reaches the high water mark, the entry goes into the `ACTIVE` state. If a PAU hit occurs on an `ACTIVE` entry, `VDD` and `FREQ` are altered as will be described in Section 5. If `Q` falls below a low water mark, `LOH2O`, this indicates that the repeated stalls with equal stride have stopped happening, and have not occurred for `STRIDE*(HIH2O-LOH2O)` cycles.

The VDD and FREQ are then set back to their default values.

In a multiprogramming environment where several different processes, with different performance characteristics, are multiplexed onto one or more processing units, HIH2O and LOH2O must permit the PAU to respond quickly enough, and STRIDE*(HIH2O-LOH2O) must be significantly smaller than the length of a process quantum. Alternatively, an operating system could invalidate all entries in the PAU on a context switch.

Addresses that only cause one stall could potentially tie up a PAU entry forever. To avoid this, PAU entries in the INIT state time out after PAU_STRIDE_MAX cycles.

3.2 Handling Cache and Memory

For real benefit across the board, both memory- and CPU-bound applications must be handled simultaneously – either the CPU is stalled for memory, or memory is idle while the CPU is busy, or both may be busy. It is desirable to use the same structure, if possible, to detect CPU-bound code regions, as for memory-bound regions, to amortize the on-chip real estate used in implementing the PAU. The control signals generated by a PAU entry for a given PC value can also be applied to shutting down memory banks or shutting down sets in a set-associative cache, along the lines of [2] and [16].

Periods of memory inactivity are detected by identifying memory load/store instructions at PC values for which the corresponding PAU entries NINSTR and STRIDE fields indicate a large ratio of computation to stalls for memory. For example, if NINSTR is very close to the ratio of STRIDE to the average machine CPI for the given architecture, then for repeated memory accesses to the corresponding address, there are very few cache misses. In such a situation, since most activity is occurring in the cache as opposed to main memory, memory can be run at a lower voltage. Similar techniques have previously been applied to RAMBUS DRAMS [1] in [14]. The PAU ensures that such adjustments only occur when they will be of long enough duration to be beneficial.

4. EXAMPLE

```
for (x = 100;;)
{
        if (x-- > 0)
                a = i;
        b = *n;
        c = *p++;
}
```

Figure 5: Example

At any given time, there may be more than one PAU entry in the ACTIVE state, i.e., during the execution of a loop, there may be several program counter values that lead to repeated stalls. In the example illustrated in Figure 5, let us assume that the assignments to variables a, b and c all cause repeated cache misses (e.g. the variables reside at memory addresses that map to the same cache line in a direct mapped cache). After 1 iteration of the loop, there will be 3 PAU entries corresponding to the PC values of the memory access instructions for the three assignments, and these will be placed in the INIT state, with their Q fields set to 0. The NCLK fields of all entries are incremented once each clock cycle hereafter. On the second iteration, after all three memory references cause cache misses once more, the three PAU entries will move from the INIT state to the TRANSIENT state, the Q fields of the entries will be incremented, the value of the NCLK field copied to the STRIDE field. The value of the NCLK fields at this point denote the number of clock cycles that have elapsed since the last hit for each entry. Likewise, the NINSTR field denotes the number of instructions that have been executed, since the last hit to the entry.

If the architecture is configured with LOH2O of 1 and a HIH2O of 3, then following a process similar to that described above, the entries will graduate to the ACTIVE state in the third iteration of the loop. On the fourth iteration, with all 3 entries in the ACTIVE state, the first PAU hit occurs due to the memory reference associated with the assignment to variable a. On a hit in an ACTIVE PAU entry, the values of the STRIDE and NINSTR fields are used to calculate the factor by which to slow down the device being controlled, which in these discussions, is the CPU. Intuitively, the ratio of NINSTR to STRIDE provides a measure of the ratio of computations to time spent stalling for memory. A detailed analysis of this calculation is described in the next section.

After 100 iterations of the loop, the variable x in the program decrements to zero, and the PAU entry corresponding to the memory access to variable a will degrade from ACTIVE to TRANSIENT and eventually to INVALID. In the organization of the PAU described here, the other ACTIVE entries would only be able to influence the operating voltage after this degradation from ACTIVE has occurred, (HIH2O-LOH2O)*STRIDE cycles after the 100th iteration of the loop.

Energy is saved when there is a PAU hit on an ACTIVE entry, and the operating voltage is lowered. When the operating voltage is lowered however, increased gate delays make it necessary to reduce the operating frequency as well, to maintain correct circuit design behavior.

At the new operating voltage, instructions will take longer to execute, but memory accesses will incur the same penalty in terms of absolute time, though the number of memory stall cycles will be smaller. There is an overhead (both time and energy) involved in lowering the operating voltage, as well as bringing it back up. This makes it useful only to lower the voltage if it can be determined that the processor will run at the low voltage for a long enough time. A more formal analysis of the opportunities for lowering operating voltage/frequency and the overheads involved therein, are presented in the next section.

5. LIMITS ON ENERGY SAVINGS

It is possible to incur no performance degradation if computation and memory accesses can be perfectly overlapped, the program being executed is memory-bound, and the CPU is run at a slower than default execution rate.

For an ACTIVE PAU entry, we can determine the effective instruction execution rate as:

$$\frac{instructions}{time} = \frac{NINSTR}{\frac{STRIDE}{FREQ}} = \frac{FREQ}{\frac{STRIDE}{NINSTRS}}$$

In the above, STRIDE/NINSTRS is the *effective CPI*, and is similar to the inverse of the *average-rate requirement* defined in [21]. It is desired to find an appropriate frequency at which we can run while keeping the ratio *instructions/time* constant. The following analysis is performed in terms of the frequency, and the interdependence between operating voltage and frequency is not explicitly shown.

The maximum value of the *instructions/time* ratio will be:

$$\frac{\frac{1}{AVGCPI}}{CYCLETIME} = \frac{RATED\_FREQ}{AVGCPI},$$

where AVGCPI is the theoretical average number of cycles necessary to execute an instruction on the architecture of interest, and RATED_FREQ is the processor's rated operating frequency. For an architecture in which memory operations can be perfectly overlapped with execution, it will be possible to lower the clock frequency until

$$\frac{RATED\_FREQ}{AVGCPI} = \frac{F_{new}}{\frac{NINSTRS}{STRIDE}}$$

Therefore

$$F_{new} = \left(\frac{RATED\_FREQ}{AVGCPI}\right) \cdot \left(\frac{NINSTRS}{STRIDE}\right)$$

The *slowdown factor*, δ, is a number greater than 1 by which the original operating frequency is divided to obtain the scaled frequency. The slowdown factor for cases of possible ideal overlap of memory operations and computation is:

$$\begin{aligned}\delta_{ideal+overlap} &= \frac{RATED\_FREQ}{F_{new}} \\ &= \left(\frac{NINSTR}{STRIDE}\right) \cdot AVGCPI\end{aligned}$$

In the general case, it will not be possible to perfectly overlap computation and memory accesses, thus slowdown of the processor will not be hidden by memory latency, since memory accesses will be sequential with computation.

In architectures that cannot overlap memory access and computation, the performance penalty can still be relatively small compared to the savings in energy, and per-cycle power will almost certainly be reduced. In such situations, since there will always be a performance degradation, it is necessary to define a limit to the acceptable degradation in performance. For the purposes of evaluation, a maximum degradation in performance of 1% will be used throughout the remainder of this paper.

$$\begin{aligned}T_{old} &= T_{mem} + T_{cpu} \\ T_{new} &= T_{mem} + \delta_{no-overlap} \cdot T_{cpu}\end{aligned}$$

For a $< 1\%$ slowdown:

$$\frac{T_{new} - T_{old}}{T_{old}} \leq 0.01$$

Therefore

$$\frac{\delta_{no-overlap} \cdot T_{cpu} - T_{cpu}}{T_{mem} + T_{cpu}} \leq 0.01$$

$$\delta_{no-overlap} \leq \frac{(0.01) \cdot (T_{mem} + T_{cpu}) + T_{cpu}}{T_{cpu}}$$

Let

$$\begin{aligned}mem\_frac &= \frac{T_{mem}}{T_{mem} + T_{cpu}} \\ cpu\_frac &= \frac{T_{cpu}}{T_{mem} + T_{cpu}}\end{aligned}$$

then

$$\delta_{no-overlap} \leq \frac{0.01(mem\_frac + cpu\_frac) + cpu\_frac}{cpu\_frac}$$

The slowdown factor can also be expressed in terms of the entries in the PAU structure:

$$\begin{aligned}mem\_frac &= \frac{STRIDE - NINSTR}{STRIDE} \\ cpu\_frac &= \frac{NINSTR}{STRIDE}\end{aligned}$$

Then

$$\delta_{no-overlap} \leq \frac{0.01(STRIDE - NINSTR) + NINSTR}{NINSTR}$$

6. PAU OVERHEAD AND TRADEOFFS

In this section, the overheads involved in adjusting the operating voltage and frequency are discussed, as well as the area cost of implementing the PAU table. The energy cost incurred by the PAU structure will be addressed in our future research. As will be shown in Section 7, a PAU size of even a single entry is effective in reducing energy consumption, while incurring a minimal degradation in performance.

Besides the PAU table, most of the information needed for each PAU entry is already available in current state-of-the-art architectures such as the Intel XScale architecture [10], and the Berkeley lpARM processor [15]. For example, the Intel XScale microarchitecture maintains event counters to monitor instruction and data cache hit rates, instruction and data Translation Look-aside Buffer (TLB) hit rates, pipeline stalls, Branch Target Buffer (BTB) prediction hit rates, and instruction execution count. Furthermore, eight additional events may be monitored when using the Intel XScale microarchitecture as the basis for an application specific standard product [10].

The largest real-estate overhead of the PAU is incurred by the PAU table and δ calculation. It should be possible to use unused functional units for δ computation, as the computation and the attendant voltage scaling can be postponed if resources are unavailable.

For an m-entry direct mapped PAU, in a b-bit architecture, with i-byte instructions, the number of bits, PAU_{bits} needed to implement one PAU entry, is given by:

$$\begin{aligned}PAU_{bits} = m \cdot ((b - log_2(m) - log_2(i)) &+ \\ 3 \cdot log_2(\texttt{PAU\_STRIDE\_MAX}) &+ \\ log_2(\texttt{HIH2O}) &+ \\ &+2)\end{aligned}$$

The terms on the right hand side of the above equation correspond to the (1) `Tag`, (2) `NCLK`, `STRIDE` and `NINSTR` (3) `Q`, (4) `FREQ` and (5) Entry state bits, respectively. Thus, a single entry PAU table can be implemented with just 106 bits on an architecture with a 32-bit PC and a chosen `PAU_STRIDE_MAX` of 2^{24}, `HIH2O` of 4 and 32-bit instructions.

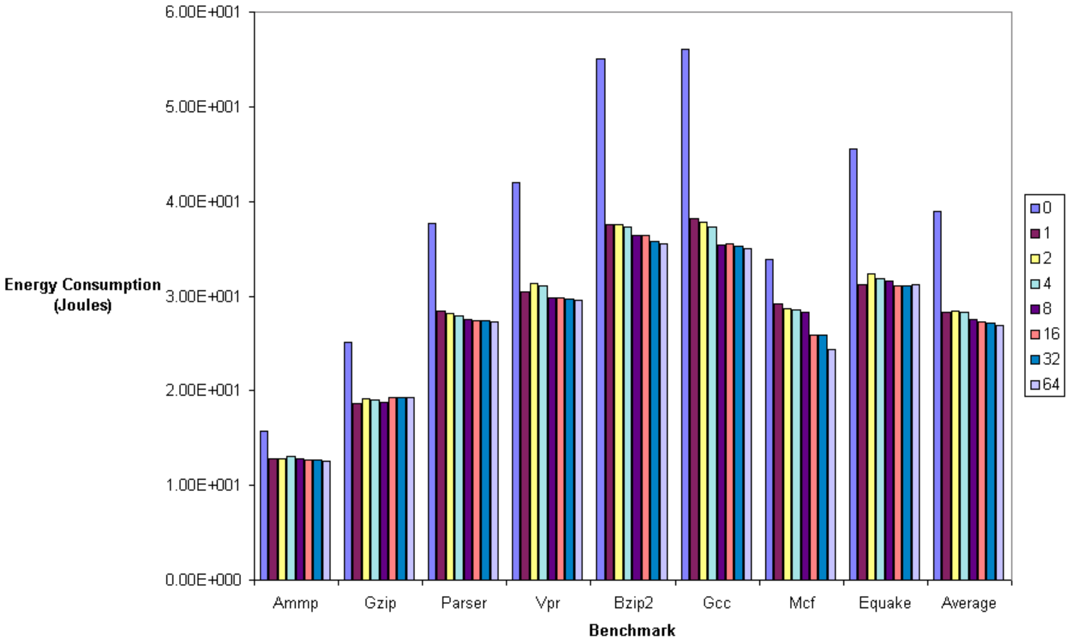

Figure 6: Effect of PAU Table size on energy consumption

Altering the operating voltage by the DC-DC converter is neither instantaneous nor energy-cost-free. In general, the time, t_{RFG} taken to reconfigure from a voltage V_1 to V_2, with a maximum current I_{MAX} at the output of the converter, converter efficiency η, and a supply smoothing capacitor with capacitance C, is given, from [6], by:

$$t_{RFG} \approx \frac{2 \cdot C}{I_{MAX}} \cdot |V_2 - V_1|$$

Likewise, the energy cost of reconfiguration, E_{RFG}, is given as:

$$E_{RFG} = (1 - \eta) \cdot C \cdot |V_2^2 - V_1^2|$$

With a DC-DC converter smoothing capacitance value of 10μF, which is twice the minimum suggested in [6], a transition from 3.3V to 1.65V, I_{MAX} of 1A and η of 90%, t_{RFG} equals 33μs. Similarly, the energy cost of reconfiguration, E_{RFG}, is 8.167500μJ. In the simulations a reconfiguration penalty of 1024 clock cycles, and 14μJ was used. This penalty may be pessimistic as it has been shown in [6] that it is possible to perform voltage scaling without halting computation for designs with a small die area, as is the case for embedded processors.

7. EFFICACY OF THE PAU

Beyond the overall architecture of the PAU, there exist implementation parameters that will determine the efficacy of the PAU in a system.

This section investigates the effect of the size of the PAU table on the energy savings and performance degradation, and ultimately, the effect on the energy-delay product. In a practical implementation, it is unlikely that a fully associative PAU structure will be utilized, due to hardware overhead involved, and a real PAU implementation is more likely to employ a small, set-associative or direct-mapped structure.

Benchmark	SPEC Suite	# of Instructions Simulated
164.gzip	Integer	200,000,000
175.vpr	Integer	200,000,000
197.parser	Integer	200,000,000
256.bzip2	Integer	200,000,000
176.gcc	Integer	200,000,000
181.mcf	Integer	122,076,300
183.equake	Float Point	200,000,000
188.ammp	Float Point	200,000,000

Table 1: Summary of benchmarks used in experimental analysis

Eight different direct-mapped PAU sizes of 0, 1, 2, 4, 8, 16, 32 and 64 entries were investigated. In all of these configurations, a VDD reconfiguration penalty of 14μJ and 1024 clock cycles was used, based on [6]. The overhead involved in performing voltage scaling was discussed further in Section 6.

7.1 Simulation Environment

The investigation was performed using the Myrmigki simulator, a power estimating execution driven simulator which models a single issue embedded processor [19]. The modeled architecture has a 5 stage in-order pipeline, unified 8K 4-way set-associative L1 cache with 16 byte blocks, and a miss penalty to main memory of 100 cycles. The power estimation framework has been shown to provide accuracy within 6.5% of the hardware it models.

The benchmarks were taken from the SPEC2000 benchmark suite, and compiled with GCC [18] version 2.95.3 for the Hitachi SH architecture. The optimization flags during compilation were the default flags specified for compiling each benchmark from the SPEC suite. Table 1 provides a summary of the benchmarks used, and number of dynamic

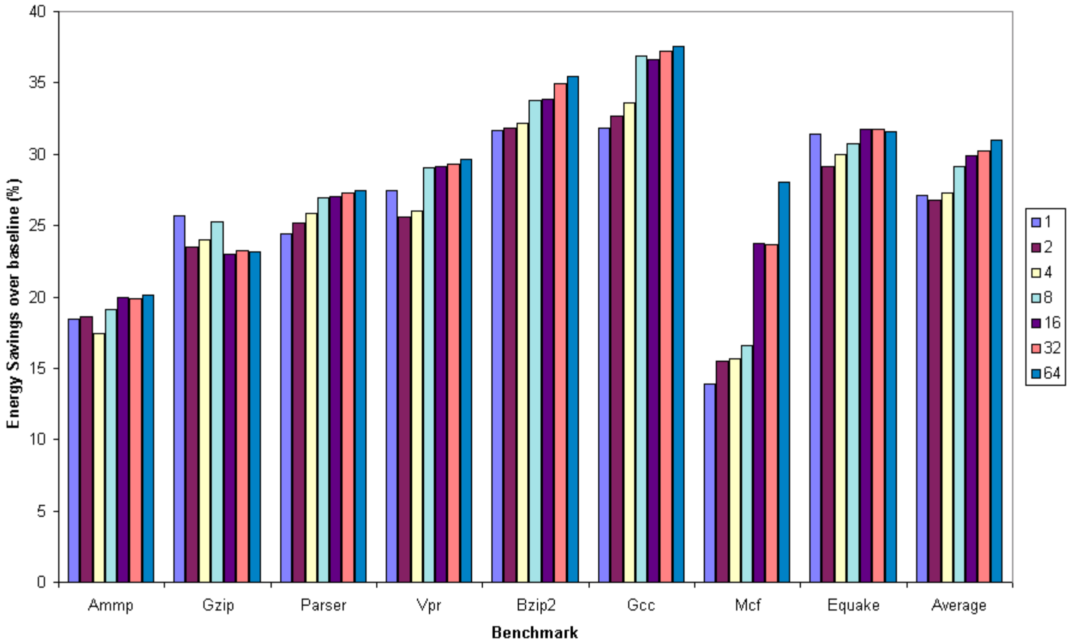

Figure 7: Effect of PAU Table size on energy savings

instructions for which they were simulated. Each of the benchmarks was simulated for the first 200 million dynamic instructions, unless the execution time was smaller, as was the case for 181.mcf. The inputs to the benchmarks were taken from the SPEC reduced simulation inputs [12], except for 176.gcc, where the reference input 166.i was used.

7.2 Effect of PAU Size on Energy Savings

Figure 6 illustrates the effect of the number of PAU entries in a direct-mapped PAU organization, on the energy consumption, for a targeted 1% performance degradation [1]. The zero-sized PAU table is the baseline case, and illustrates the energy consumption without the use of the PAU.

The percentage reduction in energy consumption with increasing PAU table size is illustrated in Figure 7. The general trend is that the energy savings for the largest PAU configuration (64 entries), is only slightly better than that of a single entry PAU. In the case of Gzip, the energy savings with a 64-entry PAU are actually less than those for a single entry PAU. This non-monotonic increase in savings with increasing PAU size can also be witnessed for Ammp, Vpr and Equake.

The reason for this behavior is that as the number of entries in the PAU table is increased, there is a greater possibility that regions of recurrent cache misses which have smaller duration with be allocated entries in the PAU. With an increase in the number of potentially less beneficial occupants of the PAU table, there is a greater occurrence of the voltage and frequency being lowered due to short runs of repeated stalls. Since there is an overhead involved in changing the operating voltage, such short runs lead to a smaller benefit in energy savings. Adding more entries to the PAU increases the opportunity for voltage scaling to occur, but does not increase the chances that a more beneficial execution region (longer, larger proportion of memory stalls) would be captured.

The trend in energy consumption in Figure 6 tracks that of energy savings in Figure 7, and the benchmarks with a larger energy consumption witness a greater savings with the use of the PAU. The effect of increased number of PAU table entries on the energy savings does not follow the same trend, with some benchmarks (e.g., Mcf) benefitting more from the use of larger PAU sizes than others (e.g. Equake).

For the average over the 8 benchmarks, there is a steady increase in the energy savings with increased number of PAU entries, except for the case of a 2-entry PAU table where there is a slight degradation over the single entry PAU. The additional energy savings from a 64-entry PAU are however not significant, with the 64-entry PAU having an energy saving of 31% versus 27% for the single entry PAU.

7.3 Effect of PAU Size on Performance Degradation

Figure 8 shows the trend in performance degradation with increasing number of PAU entries. As the number of PAU entries is increased, the number of times an entry takes control over the operating voltage increases, since there is a general increase in the number of `ACTIVE` entries. Due to the overhead involved in switching the operating voltage, there is a general increase in performance degradation which eventually plateaus, as the number of stall-inducing memory references approaches the number of PAU entries.

The increase in performance degradation is not monotonic, and for example, in going from a 4-entry PAU table to an 8-entry PAU table for Ammp, there is a decrease in the performance degradation. The reasons are the same as were previously discussed for the trend in the energy savings.

It is important to note that with increasing PAU size, even though there is an increase in the energy savings, there is a increase in performance degradation. Using larger PAU

[1] The actual performance degradation observed is not exactly 1%, as discussed in the next section

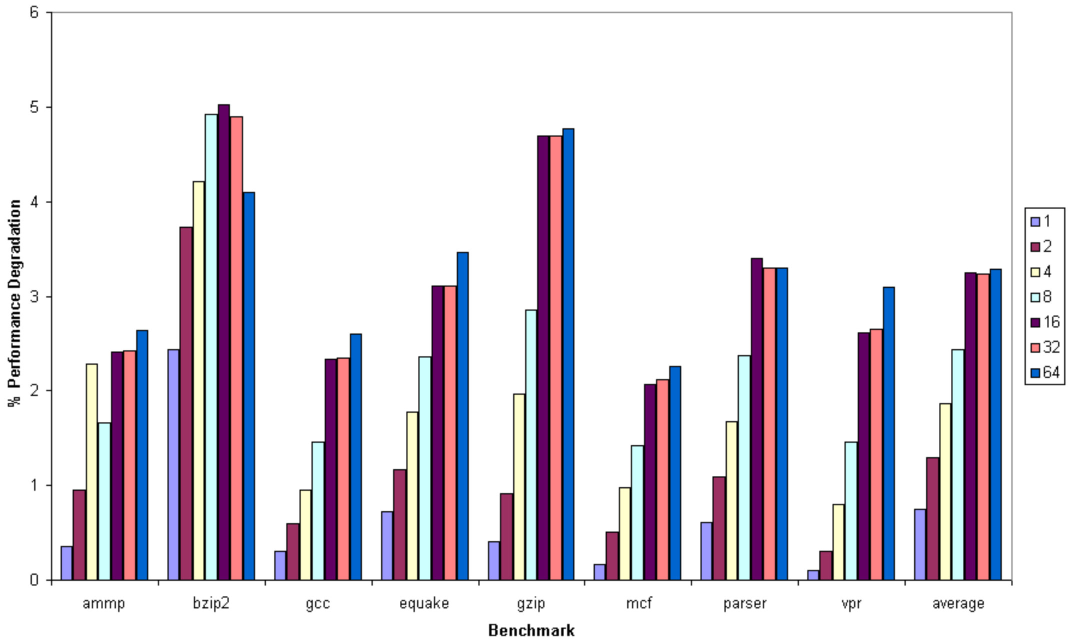

Figure 8: Effect of PAU Table size on performance degradation

tables does not provide greater accuracy, but rather only provides greater opportunity to perform resource scaling. In choosing an appropriate PAU size, it is therefore necessary to tradeoff the energy saved for the hit in performance. This makes it desirable to use the energy-delay product as a metric rather than just either performance or energy consumption.

7.4 Effect of PAU Size on Energy-Delay Product

To evaluate the efficacy of each configuration, both the energy savings and performance degradation must be considered together. An appropriate metric is the energy-delay product, with smaller values being better.

Figure 9 shows the trend in energy-delay product with increasing PAU size. In the figure, the baseline case (no PAU) is listed as a PAU table size of zero. On the average over all the benchmarks, there is little additional decrease in the energy-delay product after the addition of a single PAU entry.

Even though there is an apparent significant increase in performance degradation with increasing PAU table size (Figure 8), the contributions to energy savings far outweigh the performance penalty.

One factor that is not accounted for in Figure 9 is the additional hardware cost of larger PAU sizes. If this cost is high, it would preclude using larger PAU table sizes, as it might lead to an increase in the energy delay-product.

8. RELATED WORK

Although hardware architectures aimed at improving application performance have been around for decades, hardware targeted at reducing per-cycle power and overall energy consumption have only recently begun to be proposed.

In [20], it is observed that hardware techniques for shutting down unused hardware modules will provide the possibility for significant (upward of 20%) energy savings, over software techniques, which in themselves involve the execution of instructions which consume energy.

Hardware architectures which adapt to application needs, by reconfiguring to match applications and save energy have been proposed in [2, 11]. In [11], the authors detail a scheme for adjusting the number of hardware units, in this case resource reservation units in use, in a model of the SimpleScalar architecture, in order to reduce power consumption and overall energy consumption. The authors further propose applying the architecture to performing dynamic voltage scaling. In [2], sets in a set-associative cache are disabled to save energy, with a small degradation in performance.

Structures such as those described in [4, 17, 9] perform dynamic thermal management, reducing power consumption and saving energy, while incurring only limited application performance degradation.

In a similar manner to hardware architectures for performance, and similar also to the solutions proposed in [11], the PAU uses application history to determine opportunities for hardware reconfiguration. Furthermore, while [11] alters the hardware configuration on superscalar processors, to save energy, the PAU performs dynamic voltage scaling and clock speed setting, and addresses the spectrum of hardware architectures ranging from single-issue processors to multiple issue VLIW and superscalar architectures.

The calculation of the CPU slowdown factor in Section 5, is based on previous efforts described in [8]. In [8], the slowdown factor determination was for processors in which it is possible to overlap computation and memory accesses, such as multiple issue superscalar processors. The analysis presented in Section 5, builds upon and extends that of [8], to the general case of processors with and without the ability to overlap computations with memory accesses.

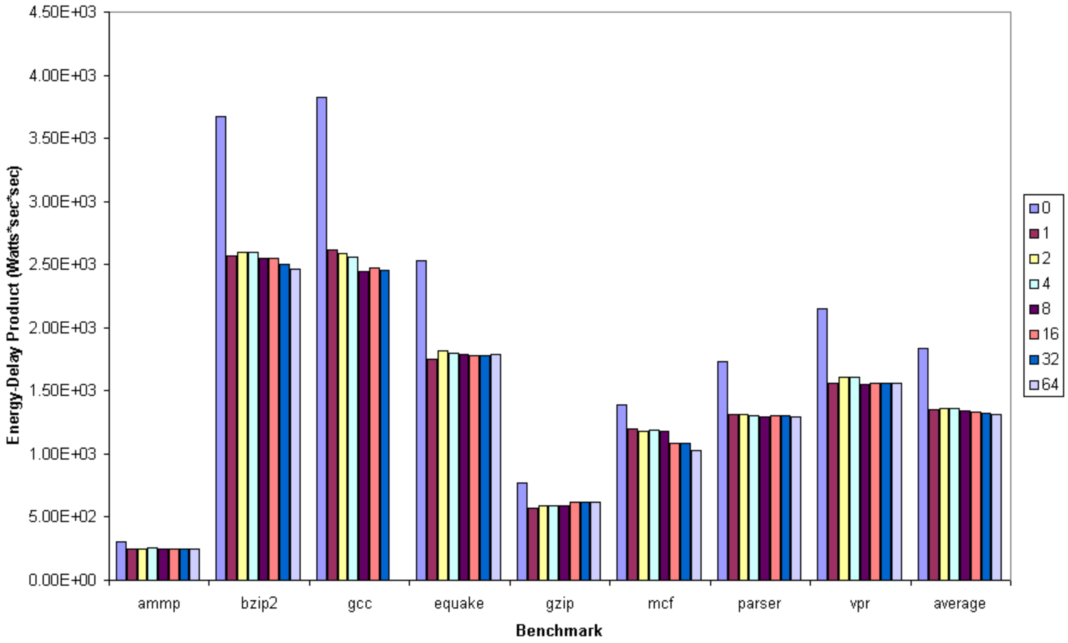

Figure 9: Effect of PAU Table size on Energy-Delay product

9. SUMMARY AND FUTURE WORK

Presented was a hardware structure, the PAU, that detects dynamic execution regions of a program for which there is a mismatch between the number of computations occurring, and the number of memory stalls, and if feasible, lowers the operating voltage and frequency of the processor to obtain a savings in energy with a slight degradation in performance.

A direct mapped configuration of the PAU was investigated for 8 PAU sizes ranging from a baseline configuration with no PAU to a 64-entry PAU. It was observed that PAU sizes of even a single entry provide an average of of 27% savings in energy with performance degradation of 0.75%. In general, it was observed that there was increased energy savings accompanied by increased performance degradation with increasing PAU size, as more penalties of voltage scaling were incurred. The overall effect of using larger PAUs was however positive, with an overall decrease in the energy-delay product as the PAU size was increased.

Lacking in the current analysis is an accurate estimate of the hardware cost of the PAU. This is the subject of our current research, and we are investigating various hardware implementations.

The usefulness of a PAU in a superscalar architecture is also being investigated, with the implementation of the PAU in the Wattch [5] simulator. This will also permit preliminary analysis of the hardware cost of the PAU table, as it will be possible to model the PAU table in Wattch as an array structure.

Lastly, the proposed hardware structure only addresses the dynamic power dissipation, though the use of voltage scaling. With decreasing feature sizes, leakage power is becoming increasingly important, and it is therefore necessary to investigate the possible impact of any proposal on the leakage power consumption.

It should be straightforward to incorporate a PAU into current state-of-the-art low power architectures, given that most of the hardware required by the PAU is currently beginning to appear in some commercial and research microprocessor designs. In the short term however, it should be possible to implement the PAU in a programmable logic device and use it as an additional board level device in a system design.

10. REFERENCES

[1] RDRAM. `http://www.rambus.com`, 1999.

[2] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. *Journal of Instruction Level Parallelism*, 2(2000):1–6, May 2000.

[3] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[4] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[6] T. D. Burd and R. W. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design, ISLPED'00*, pages 9–14, July 2000.

[7] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-Directed Dynamic Frequency and Voltage Scaling. In *Workshop on Power-Aware Computer Systems, ASPLOS-IX*, November 2000.

[8] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-Directed Dynamic Frequency/Voltage Scheduling for Energy Reduction in Microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design, ISLPED'01*, August 2001.

[9] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 202–213, 2000.

[10] Intel Corporation. Intel XScale Microarchitecture Technical Summary. Technical report, 2001.

[11] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of 2000 Design Automation and Test in Europe*, pages 190–196, 2001.

[12] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC2000 Benchmark Suite for Simulation-Based Computer Architecture Research. In *Proceedings of the Workshop on Workload Characterization, International Conference on Computer Design*, September 2000.

[13] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Jan. 1981.

[14] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power Aware Page Allocation. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–116, November 2000.

[15] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lparm microprocessor system. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design, ISLPED'00*, pages 96–101, July 2000.

[16] M. D. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. An Energy-Efficient High-Performance Deep-Submicron Instruction Cache. *IEEE Transactions on VLSI, special issue on Low-Power Electronics and Design*, February 2001.

[17] H. Sanchez, B. Kuttanna, T. Olson, M. Alexander, G. Gerosa, R. Philip, and J. Alvarez. Thermal Management System for High Performance PowerPC Microprocessors. In *Proceedings IEEE Compcon*, page 325, February 1997.

[18] R. M. Stallman. Using and Porting GNU CC, 1995.

[19] P. Stanley-Marbell and M. Hsiao. Fast, flexible, cycle-accurate energy estimation. In *ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED'01.*, August 2001.

[20] V. Tiwari and M. Lee. Power analysis of a 32-bit embedded microcontroller. In *Proceedings, Asia and south Pacific DAC*, pages (CD–ROM), August 1995.

[21] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings IEEE Symposium on Foundations of Computer Science*, pages 374–382, October 1995.