Exploring the Potential of Architecture-Level Power Optimizations

John S. Seng
Dept. of Computer Science
Cal Poly State University
San Luis Obispo, CA 93407
jseng@calpoly.edu

Dean M. Tullsen
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
tullsen@cs.ucsd.edu

Abstract

This paper examines the limits of microprocessor energy reduction available via certain classes of architecture-level optimization. It focuses on three sources of waste that consume energy. The first is the execution of instructions that are unnecessary for correct program execution. The second source of wasted power is speculation waste – waste due to speculative execution of instructions that do not commit their results. The third source is architectural waste. This comes from suboptimal sizing of processor structures. This study shows that when these sources of waste are eliminated, processor energy has the potential to be reduced by 55% and 52% for the integer and floating point benchmarks respectively.

1. Introduction

Much research has been aimed at reducing the power consumption of processors, including high-performance general purpose microprocessors. Power consumption can be reduced at many levels, from the circuit level to the application level. Our research explores the limits of power reduction available due to optimizations at the architecture level. We identify three broad categories of wasted energy that either have been, or could potentially be, targeted for reduction. Those categories are program waste, speculation waste, and architectural waste. The focus of this research is the reduction of power and energy dissipation in high-performance, high instruction level parallelism processors. and that is the context in which these limits are studied.

Program waste arises when program execution contains instructions that are not necessary for correct execution. This includes instructions that produce either dead or redundant values, or any instruction whose only consumers produce dead or redundant values. We consider an instruction *unnecessary* if it produces a dead value or does not affect any change on processor state (e.g., the contents of registers and memory). Examples of power and performance optimizations that target this type of waste include elimination of silent stores [38, 41] and dynamically dead instructions [10].

Speculation waste results from speculative execution following mispredicted branches. These instructions are fetched into the processor and consume pipeline resources and energy, but are never committed and do not change permanent processor state. Examples of energy and performance optimizations that target this type of waste include pipeline gating [23] and aggressive branch predictors [32].

Architectural waste results from the static sizing of architectural structures, such as caches, instruction queues, branch predictors, etc. Memory array structures are often designed to hold a large amount of data in order to obtain good overall performance, but a given program usually cannot exploit this at all times. Instruction queues are designed with a large number of entries when often there is little parallelism in the running code. For performance reasons, these structures are made as large as possible. Making them too small can also waste power (causing more miss traffic, increasing misspeculation, etc.). This study examines the power dissipation of caches and instruction queues that are always exactly the right size. Examples of power and performance optimizations that target this type of waste include selective cache ways [1] and dynamically resized issue queues [14].

The analysis in this paper assumes an aggressive, dynamically scheduled wide superscalar processor, but many of the results would apply to the same relative degree to nearly any processor. While a limit study, this research does not encompass the power reduction available from profound architectural changes (e.g., the difference between a dynamically scheduled processor and a statically scheduled processor), but rather focuses on a single architecture that adapts in idealized ways to avoid waste.

Because this is a limit study, we make no assumptions about how or to what extent these sources of waste are avoided. Some of the techniques to avoid the waste are obvious, or have been investigated before, others are more difficult to imagine. In this paper, we strive to not let our imagination constrain the evaluation of the limits of architectural power reduction.

Among the findings of this research are the following:

- To fully exploit the opportunities to reduce program waste (unnecessary computation) requires identifying not only the redundant or useless instructions, but also the instructions which only produce values for those instructions.

- Revolutionary advances (e.g., more than a factor of two or three) will not come without significant changes to the underlying architecture (the type of changes that lie outside the underlying assumptions of this study).

This paper is organized as follows. Section 2 describes related work. Section 3 describes the simulation and experimental methodology used in our experiments. Section 4 describes program waste and its associated energy costs. Section 5 describes speculation waste and quantifies the cost of speculative execution. Section 6 describes architectural waste. Section 7 demonstrates the effect on energy consumption when the three sources of waste are removed. The paper concludes in Section 8.

2. Related Work

Previous architecture research has identified many of the sources of waste identified in this paper; those papers have targeted both performance optimizations and power optimizations. In this section we list both types of previous research.

Program waste occurs when instructions that are either dead or redundant are executed by the processor. Prior works have studied particular types of unnecessary instructions. Specific prior work has targeted dynamically dead instructions [10, 24], silent stores [20, 38, 41], silent loads [21, 25, 38], silent register writes [16, 24, 37], and predictably redundant instructions [9, 33]. Each of these works provides a technique for dynamically identifying and predicting unnecessary instructions. Many of them only study the performance gains from eliminating execution of the these instructions.

A further analysis of unnecessary instruction chains is provided in [30]. In [30], Rotenberg similarly identifies sequences of instructions that are executed, but do not contribute to program correctness. That work analyzes sequence lengths of unnecessary instructions and proposes predicting such sequences. The paper proposes eliminating the instruction sequences from the primary thread of execution and verified by another execution context.

Program redundancy also exists in the form of instructions which operate on repetitive operands. The work by Sodani et al. [34] is a comprehensive study into the repetition of inputs and outputs of dynamic instruction instances. Our work presents the energy impact of dynamic instructions with common inputs and static instructions with a small set of input values. In [33] Sodani et al. propose instruction reuse, a performance optimization which eliminates execution of instructions with repetitive inputs.

Speculation waste occurs when instructions are fetched and executed after a mispredicted branch. These instructions will eventually be flushed from the pipeline without performing any useful computation. In benchmarks where there are many difficult to predict branches the number of instructions that are not committed is significant.

One approach to solving this problem is to improve the accuracy of branch predictors. This has been extensively studied. Work that attempts to minimize the energy effects of misspeculated instructions is much more limited. Pipeline gating [23] addresses the problem of misspeculation by keeping instructions from continuing down the processor pipeline once a low-confidence branch has been fetched. By limiting the number of instructions fetched after an often mispredicted branch, the amount of energy wasted can be minimized. A similar concept is presented by Bahar et al. [3]. Seng et al. [31] demonstrate that multithreaded execution [36] conserves a significant amount of energy by relying less on speculation to achieve performance.

Suboptimal sizing of processor structures exists when hardware structures are inappropriately sized, both statically and dynamically, for program behavior and the different phases of program behavior. An approach to solving this problem for caches has been dynamic reconfiguration of cache structures. This has been studied in [5, 19, 29, 39, 40]. Another general technique to reduce the power consumption of caches is to power down parts of the cache that are unused [1, 27].

Several papers have proposed techniques for reducing the energy consumption of the instruction issue logic. A common approach to reducing the energy of the issue queue has been to dynamically reconfigure the issue window size [11, 12, 14, 15]. The authors note that most instructions are issued from the head of the issue queue. They propose particular implementations of dynamically resized issue queues in order to gain power savings. Other work, in contrast, resizes the issue queue on the basis of available parallelism [4, 17].

Another technique to lessen the cost of suboptimal sizing is hierarchical designs. This model includes deeper cache hierarchies [2], as well as hierarchical instruction queues [28].

Some areas that we do not study include the branch predictor, the functional units, and the register file. Energy optimizations for branch predictors are studied in [26]. Some proposed architecture-level optimizations for functional units are described in [4, 7]. Register file optimizations include hierarchical register files [6].

3. Methodology

Simulations for this research were performed with the SMTSIM simulator [35], used exclusively in single-thread mode. In that mode it provides an accurate model of an out-of-order processor executing the Alpha instruction set architecture. The simulator was modified to include the Wattch 1.02 architecture level power model [8].

The SPEC2000 benchmarks were used to evaluate the designs, compiled using the Digital `cc` compiler with the `-O3` level of optimization. All simulations execute for 300 million committed instructions. The benchmarks are fast forwarded (emulated but not simulated) a sufficient distance to bypass initialization and startup code before measured simulation begins. Additionally, the processor caches and branch predictor are run through a warmup period of 10 million cycles before data collection. Table 1 shows the benchmarks used, their inputs, and the number of instructions fast forwarded. In all cases, the inputs were taken from among the reference inputs for those benchmarks.

Benchmark	Input	Fast forward (millions)
crafty	crafty.in	1000
eon	kajiya	100
gcc	200.i	10
gzip	input.program	50
parser	ref.in	300
perlbmk	perfect.pl	2000
twolf	ref	2500
vortex	lendian1.raw	2000
vpr	route	1000
art	c756hel.in	2000
equake	inp.in	3000
galgel	galgel.in	2600
gap	ref.in	1000
mesa	mesa.in	1000
mgrid	mgrid.in	2000

Table 1. The benchmarks (integer and floating point) used in this study, including inputs and fast-forward distances used to bypass initialization.

Details of the simulated processor model are given in Table 2. The processor model simulated is that of an 8-fetch 8-stage out-of-order superscalar microprocessor with 6 integer functional units. The instruction and floating-point queues contain 64 entries each, except when specified otherwise. The simulations model a processor with instruction and data caches, along with an on-chip secondary cache.

The potential energy savings from eliminating program waste will be largely independent of the processor model, because it scales primarily with instruction count. Waste due to speculation and architectural waste will, to some extent, be particular to the processor modeled here; however, these results should scale in a qualitative way to a variety of architectures.

For the experiments involving program waste, a trace of 300 million dynamic committed instructions is captured. Dynamic instruction instances are then marked in the trace if they are determined to be unnecessarily executed. Producers for these instructions are also considered unnecessary and marked if all their dependents are either unnecessary or producers for unnecessary instructions. In order to capture the dependence chain of instructions, the instruction trace is processed in reverse order.

In order to model the effect of removing the energy usage of dynamic instruction instances, the trace is used as input during a second simulation run. Care was taken to ensure that the same instructions were executed for each run of the simulator - this guaranteed that only the effects of the dead and redundant instructions were being measured. When an instruction is marked as unnecessary, we emulate it in the simulator but do not charge the processor for the energy cost of any aspect of its execution.

Because this is a limit study, we do not make assumptions about, or charge the processor for, the techniques used to exploit the particular phenomenon we study. For example, one technique to avoid predictably redundant execution is instruction reuse. Instruction reuse, as proposed in [33], would require energy for access to a reuse buffer, and would primarily save the power used by the execution units. In our study, we do not account for the energy of the additional processor structures required; we only remove the energy cost of instructions. This, then, does not limit the results to a particular architectural solution.

Parameter	Value
Fetch bandwidth	8 instructions per cycle
Functional Units Instruction Queues	3 FP, 6 Int (4 load/store) 64-entry FP, 64-entry Int
Inst Cache	64KB, 2-way, 64-byte lines
Data Cache	64KB, 2-way, 64-byte lines
L2 Cache (on-chip)	1 MB, 4-way, 64-byte lines
Latency (to CPU)	L2 18 cycles, Memory 300 cycles
Pipeline depth	8 stages
Min branch penalty	6 cycles
Branch predictor	21264 predictor
Instruction Latency	Based on Alpha 21164

Table 2. The processor configuration modeled.

4. Program Waste

Many instructions that are executed do not contribute to the useful work performed by the program. These instructions may include ones which produce values which will never be used, those instructions which do not change processor state in any way, or which predictably change program state in the same way. In this work, we quantify the impact of these redundant instructions on the energy usage of a processor. We start by examining those instructions that do not do useful work (either produce dead values or do not change program state), and study predictably redundant instructions in Section 4.2.

4.1. Unnecessary Instructions

Instructions which do not do useful work for program execution are considered *unnecessary*. We classify unnecessary instructions into a number of categories.

Dead instructions (*dead* in Figures 1 and 2) Instructions which produce dead values are instructions where the destination register will not be read before being overwritten by another instruction. In addition, store instructions are also considered dead if the memory location written is overwritten before being read by a load instruction. A dead instruction can produce a value which is dead for every execution instance (statically dead, because of poor instruction scheduling by a compiler) or for select instances (because of a particular path taken through the code). In both cases, the instruction producing a dead value has no effect on program correctness and we assume it can be eliminated.

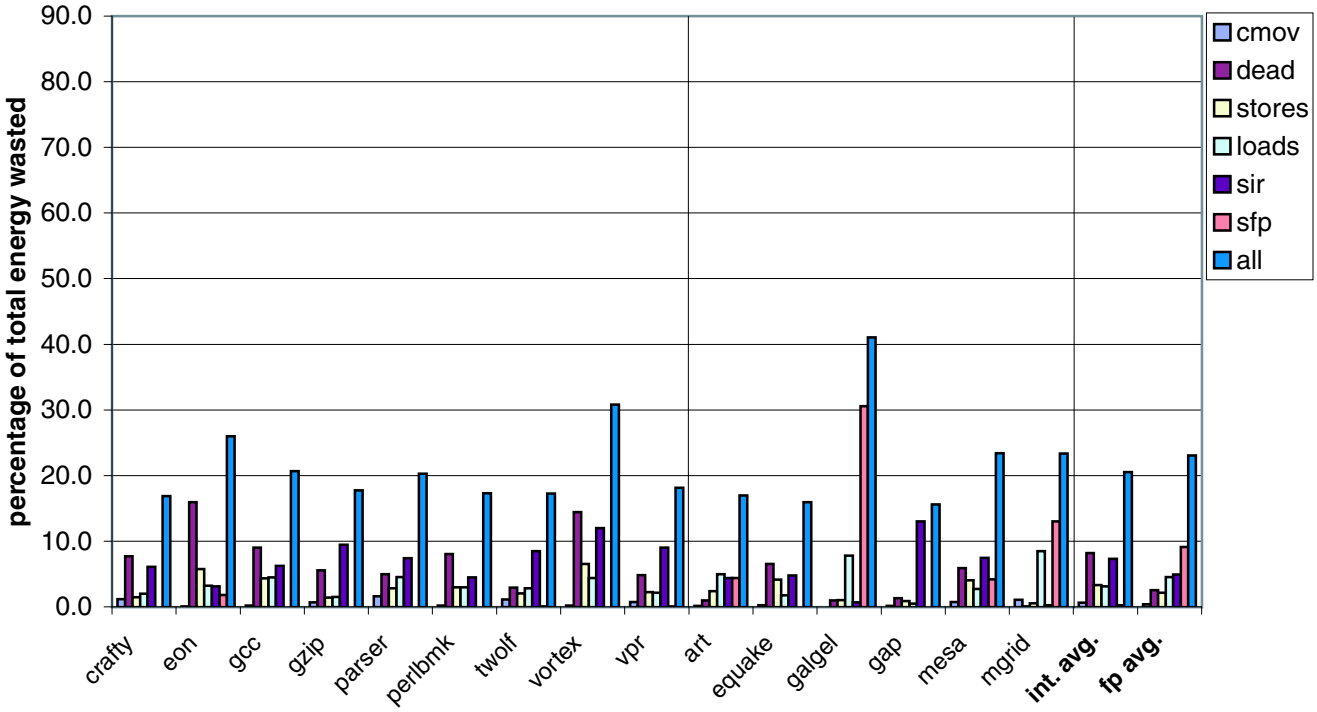

Figure 1. Processor energy waste due to unnecessary instructions.

Silent stores (*stores*) These are store operations where the value stored is the same as the value already at the address being stored to. Prior research has shown techniques which identify these instructions and remove them from the instruction stream [38, 41]. Removing silent stores from the dynamic instruction stream improves performance by reducing the memory traffic generated by these instructions. In this paper, we quantify the energy effects of these unnecessary instructions.

Silent loads (*loads*) These are similar in nature to silent stores. Silent loads are load instructions where the value being loaded is the same as the value already in the destination register. Again, these instructions have no effect on program state and correctness, and therefore can be removed without incorrect program execution. These are a special case of the silent register instructions, but are interesting as a sub-category because of the high performance and power cost of load operations.

Silent register operations Silent integer register operations (*sir*) are those integer instructions whose result is the same as the value already in the destination register. Similarly, silent floating point operations (*sfp*) are those instructions where the resulting value is the same as the destination floating point register.

Silent conditional moves (*cmov*) These are conditional moves where the condition is not met and the move does not occur. Because the move is not performed, the instruction is essentially a null operation and energy is wasted in the fetching and executing of these instructions. This is the only support for predication in the Alpha ISA. An architecture with richer support for predication, such as the Intel IA64, would have more opportunities for removing waste.

The truly redundant operations are the silent register operations, silent loads, and silent stores. The dead instructions and silent conditional moves are not redundant, but special cases where their execution has no effect on architectural state.

In addition to the dynamic instruction instances determined to be unnecessary, in many cases the *producers* of values consumed by those instructions can be marked unnecessary as well, as long as the value only gets used by instructions already deemed to be unnecessary. Since we search the dynamic trace in reverse order, this can be determined with a single pass through the trace. In following chains of unnecessary instructions, we only follow register dependences, assuming that values passed through memory are useful.

4.1.1 Program Waste Results

Figure 1 shows the percentage of total processor energy used for the execution of each type of program waste. The results shown represents the energy waste only due to those instructions which are marked as unnecessary and does not include the producers for those instructions. The data is shown for each of the integer benchmarks, the average of the integer benchmarks, each of the floating point benchmarks, and the

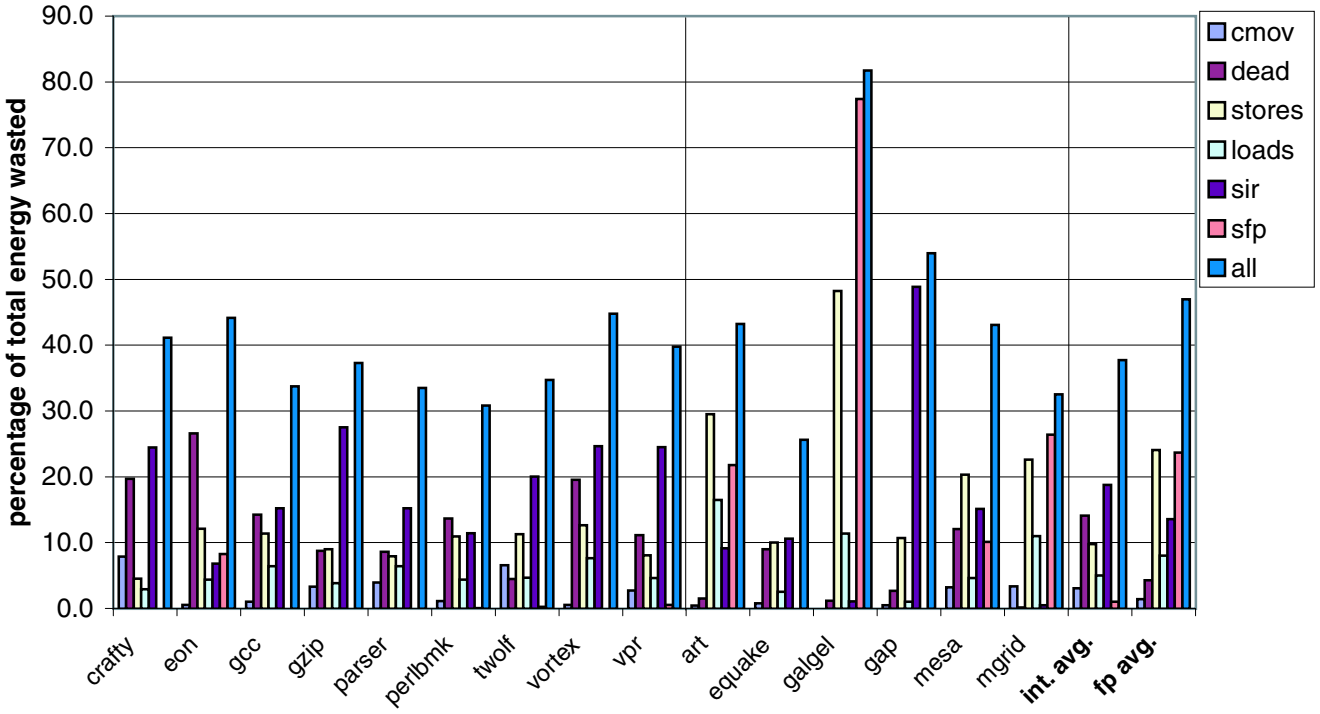

Figure 2. Processor energy waste due to unnecessary instructions and their producers.

average of the floating point benchmarks. The categories of unnecessary instructions are mutually exclusive except for the *dead* category where instructions are marked regardless of their type. The *all* category represents accounting for all types of program waste.

For the integer benchmarks, the most significant sources of program energy waste are due to dead (8.2%) and silent integer operations (7.4%). The `eon` benchmark wastes more energy with dead instructions (16.0%) than any other integer benchmark. Removing the energy costs of silent conditional operations provides little change (0.7% for the average). For the average of the integer benchmarks the total waste is 20.6%.

For the average of the floating point benchmarks, most waste is due to silent loads, silent integer, and silent floating point operations. Some of the floating point benchmarks exhibit very little waste for silent floating point operations (`art` and `gap`); for the `galgel` benchmark, this waste is significant at 30.6%. As with the integer benchmarks, little energy is wasted on silent conditional moves. The average total program waste for the floating point benchmarks is 23.1%.

When including the instructions whose only contribution is to produce values for unnecessary instructions (Figure 2), the potential energy savings increases significantly. For the integer benchmarks, the chains of instructions leading to dead and silent integer operations are the leading sources of program waste. The waste due to dead instruction chains is 14.1% and the waste due to silent integer instruction chains is 18.8%.

For the floating point benchmarks, silent floating point and silent store instruction sequences are the leading sources of energy waste. The `galgel` benchmark has an exceptionally large number of unnecessary silent floating point instruction chains (77.4% energy wasted). For these benchmarks, when counting the producer instructions for the silent stores, the total energy waste is significant (24.1%). This demonstrates that in the floating point benchmarks there are long sequences of instructions to compute a store value, and often that store is silent. In fact, for both integer and floating point programs, the category that gained (relatively) the most when including producing instructions was the silent stores.

It should be noted that the sum of the different sources of program waste are not strictly additive. Some may be counted in two categories (e.g., an instruction that is both redundant and dead). In fact, we see a high incidence of this effect when we include the producers, because many unnecessary instructions are also producers of other types of unnecessary instructions. We only count these instructions once for the *all* category. The total can also be more than additive, because a producer instruction may produce a value which is then consumed by multiple unnecessary instructions of different types, and is only considered unnecessary in the case that we are considering all its dependents as unnecessary, and thus only be counted in the *all* case.

These results indicate that the potential for energy savings is much more significant when including the chains of instructions that together produce an unnecessary value. The actual instructions that produce the value represent only a fraction of the wasted energy. Considering all unnecessary instructions, the integer benchmarks waste 37.7% of total energy. The total for the floating point benchmarks is greater at 47.0%.

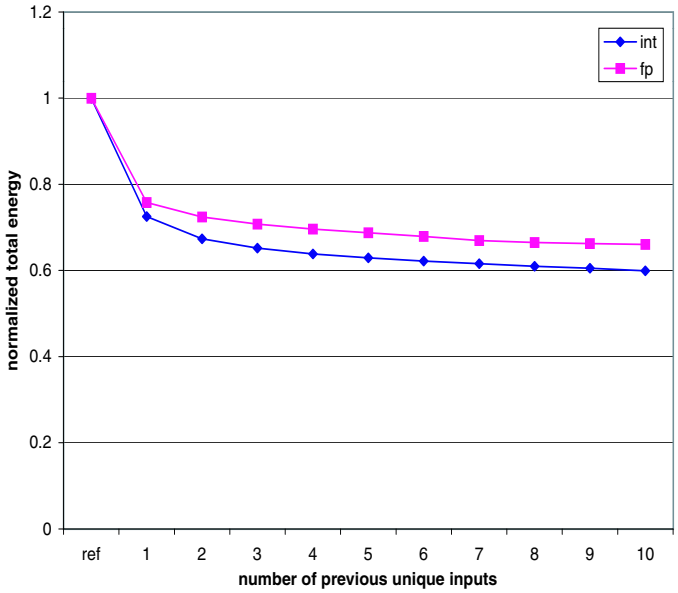

Figure 3. Average energy for benchmarks when removing energy of instructions that have same inputs as one of the last n unique sets of inputs seen by the same instruction.

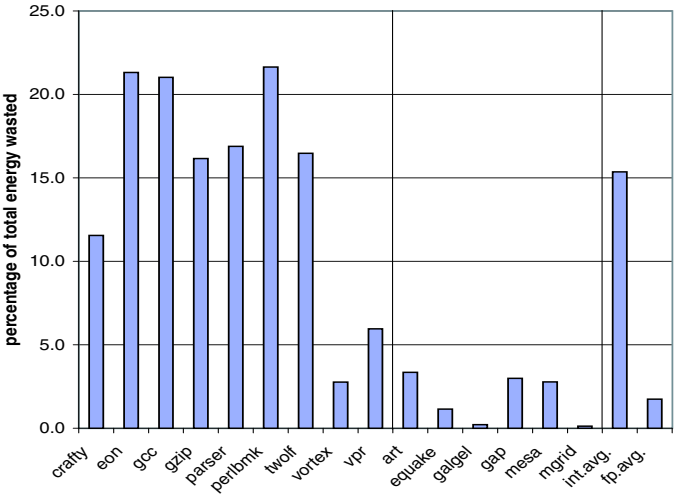

Figure 4. Percentage of total energy that is wasted due to misspeculated instructions.

4.2. Predictable Instruction Redundancy

The previous section examined instructions that had a particular type of redundancy – they wrote values that were already in the destination location. Another type of redundancy is instructions which repeatedly produce the same value, or operate on the same inputs. These instructions are not strictly unnecessary by our previous definition because they may still change program state; however, there is still potential to exploit this redundancy. There will be some overlap between this redundancy and the silent register operations, if an instruction produces the same value and there have been no intervening writes to the destination register.

Instructions may exhibit either output or input redundancy. Because instruction reuse (which detects that an instruction is being executed with the same inputs as a prior instantiation [33]) has more potential than value prediction to save power (value prediction is a speculative technique that requires all predicted instructions be executed to check the prediction), we focus on input redundancy rather than output redundancy. In this section, we explore the potential for power savings when instructions displaying varying degrees of input operand reuse are eliminated from the instruction stream.

Figure 3 graphs the normalized total energy when removing energy of instructions that commonly see the same inputs as previous values. In this experiment, if an instruction has the same inputs as one of the last few unique sets of inputs seen by that same instruction, then its energy cost is removed. The x-axis represents the number of previous unique input combinations that an instruction was compared against. `Ref` indicates the baseline simulations where the energy costs of all instructions are counted. The graph shows that a significant amount of energy could be eliminated for instructions that exhibit this behavior. When comparing instructions only against the inputs of the last instruction instance, energy was reduced by 27.5% and 24.2% for the integer and floating point benchmarks, respectively, by exploiting the predictable redundancy. When comparing against the last two unique inputs, these numbers increase to 32.6% for the integer and 27.6% for the floating point.

In a separate experiment, we found that many instructions are actually statically redundant, or virtually so. For the integer benchmarks, we found that 5.3% of total energy is consumed by instructions which never see more than one set of operands, and 9.4% is used on instructions which have only 2 different sets of inputs.

5. Speculation Waste

Speculative execution enables higher performance on modern processors. For high performance, a long pipeline requires the prediction of branches and the execution of subsequent instructions. In cases where branches are incorrectly predicted, the misspeculated instructions are flushed from the pipeline and refetching occurs down the correct path. Although there is no program correctness penalty associated with executing misspeculated instructions, there are energy costs associated with the instructions.

Figure 4 shows the effect on processor energy usage of misspeculation. In this experiment, we simulate the benchmarks 2 times. In the first simulation we obtain the energy results for all instructions entering the processor pipeline. In the second set of simulations, we only account for the energy of those instructions which are committed. The effect of the misspeculation is much more significant in the integer benchmarks because of the increased number of difficult to predict

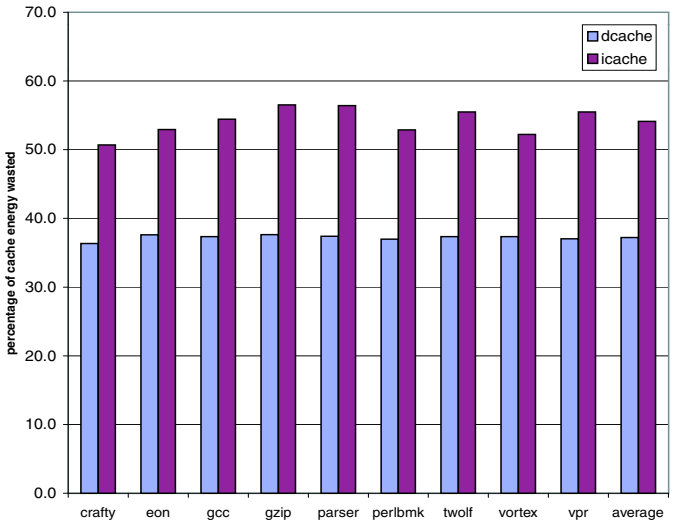

Figure 5. Potential cache energy savings with optimal cache sizing for the integer benchmarks.

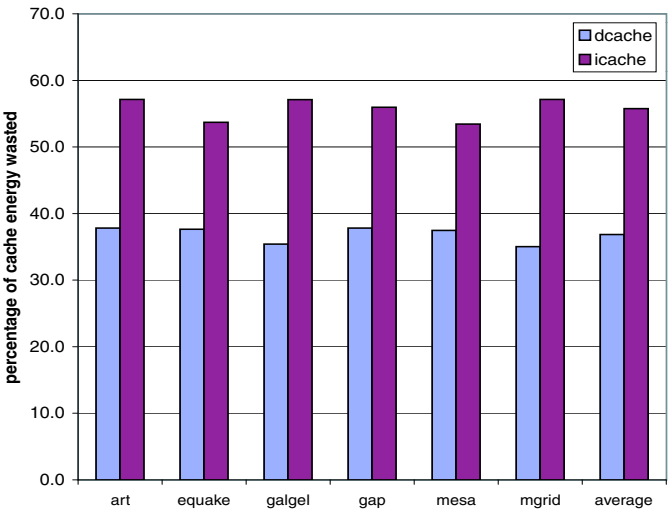

Figure 6. Potential cache energy savings with optimal cache sizing for the floating point benchmarks.

branches. For our benchmarks, there are on average 3.3 times as many mispredicted branches in the integer benchmarks as in the floating point benchmarks.

In this work, we have not studied the energy effects of other forms of speculative execution (e.g. load speculation [13] or value speculation [22]). We focus on control speculation as it will be an increasingly significant portion of total energy waste as processor pipelines increase in length.

Although speculative execution results in the execution of uncommitted instructions, speculation is necessary to obtain high performance. Eliminating speculation is not practical, but controlling the level of speculation provides the potential for efficient processor design [23, 31].

6. Architectural Waste

In this section we study how the architectural design of a processor can contribute to the energy wasted during program execution. We define architectural waste to occur when the size of a processor structure is not optimal for the current state of the particular application currently executing. Suboptimal structure sizing occurs when a resource on the processor is larger (or in some cases smaller) than it needs to be for a particular program.

The sizes of structures on a processor architecture are determined based upon overall performance requirements along with technological limitations. In this section, we study how much energy is wasted because of suboptimal sizing for particular applications. This knowledge is important in gauging the energy potential of architectures which propose adaptive reconfiguration in order to reduce power consumption.

The structures we look at are the instruction and data caches, and the instruction issue queues. These represent a subset of all processor structures that could be adaptively sized, but do represent key structures which consume a significant amount of power and are typically sized as large as possible for peak performance. Other memory-based structures that could be studied in a similar manner include the register file, TLBs, reorder buffer, and the branch predictor. A study of optimal structure sizing provides insight into the potential energy savings to be gained given oracle knowledge of program behavior.

6.1. Cache Resizing

Because different programs have different memory behavior and access patterns, a cache of fixed size and configuration is not optimal for all cases. In cases where the processor is executing a tight loop, a large instruction cache is not necessary. This is also true for the data cache if the memory footprint of the application is very small. In these instances smaller caches would provide the same performance, but with lower energy requirements.

In order to quantify the amount of energy wasted due to suboptimal cache sizes, we need to model a cache that is of the perfect size for the application currently being executed. To simulate the optimally sized cache, we simulate a continuum of cache sizes and configurations (24 different instruction and 24 different data caches). The cache sizes range from 2KB to 64KB with associativities ranging from 1 way to 8 way. Each cycle we select the smallest (lowest power) cache from among those (typically several) that hit on the most accesses. To compute the energy savings, we compare the optimal results against our baseline, a 64 KB 2-way cache.

In figure 5 we show the results of this experiment for the integer benchmarks. The results for the floating point benchmarks are shown in figure 6. The data represents the fraction of cache energy that is wasted due to a cache that is not of

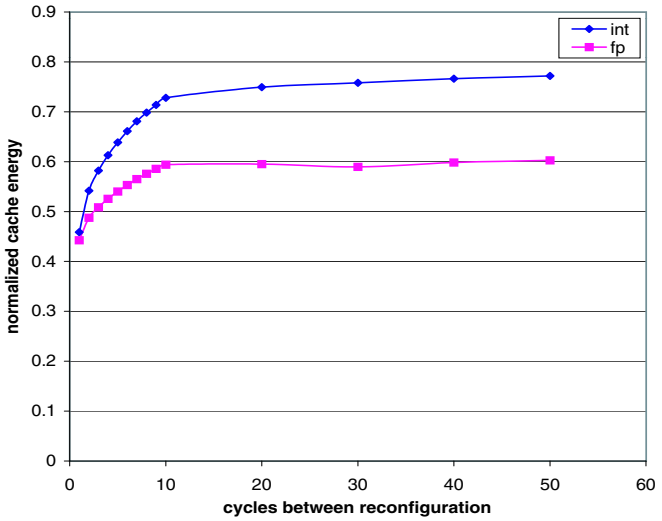

Figure 7. Effect on instruction cache energy when reconfiguring at different intervals.

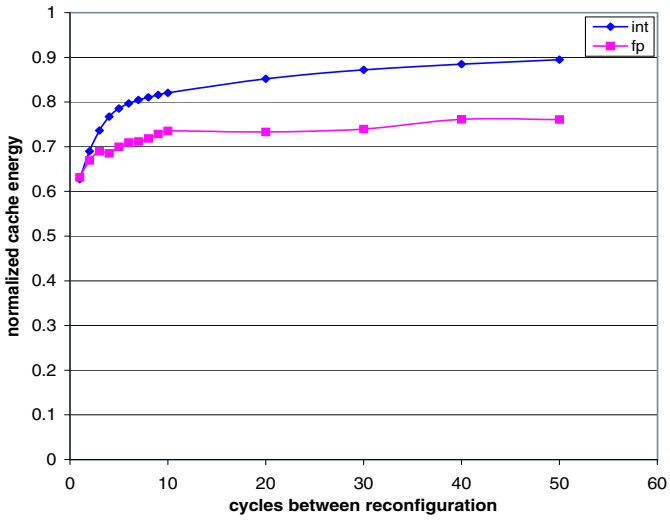

Figure 8. Effect on data cache energy when reconfiguring at different intervals.

optimal size at the time of access. The energy modeled represents energy due to cache accesses and no other effects such as leakage.

The results show potential energy reductions for the integer applications of 54.1% of the total instruction cache power and 37.2% for the data cache. For the floating point applications, the potential reduction is 55.7% and 36.8% for the instruction and data caches, respectively.

In addition, figures 7 and 8 show results where the granularity of reconfiguration is varied from every cycle to once every 50 cycles. When the granularity is larger than one cycle, we select for that entire interval the largest cache that was selected over the range. This could be pessimistic, as in some cases the best power might be achieved with a smaller cache and a few more accesses allowed to proceed to the second level cache, but it is a definition of optimal that is straightforward to compute. There is a slight dip in the floating point results due to a particular interval value that resonates strongly with the timing of one particular benchmark (`art`).

Both the data and instruction cache have the potential for large savings via optimization and adaptability. Most of the benchmarks put considerably less pressure on the instruction cache, and the results indicate smaller caches are needed for instructions.

These results indicate that reconfiguration must occur very often to get most of the available benefit. However, that need not discourage work in this area. Other paradigms besides whole cache-level reconfiguration fit into this category. For example, techniques that adapt individual cache lines (e.g., powering down lines unlikely to be accessed again [18]) potentially do reconfigure the cache on a cycle-by-cycle basis.

6.2. Issue Queue Resizing

In an out-of-order processor, the size of the issue window greatly affects performance. A larger window size allows a processor to extract more parallelism from the code sequence it is currently executing. More instructions can issue because of the increased number of potentially available instructions. Although a large window is helpful, there are instances where the instruction stream contains much less parallelism than the maximum processor issue width. When this occurs, the larger issue queue cannot be fully utilized. Additionally, the issueable instructions are frequently located in a small portion of the issue queue [14]. In this case the wakeup logic and selection logic are much larger than necessary. It is these instances for which we consider an instruction queue design to be wasteful. This section quantifies that waste.

In order to measure the waste due to oversized issue queues, we compare the energy used by a fixed size issue queue versus one that can be dynamically resized. We assume that the dynamically resized queue is optimally resized every cycle. An optimally sized queue is one where the queue contains the fewest number of entries required to maintain the same performance (that is, issue the same number of instructions over that interval) as a large fixed size queue. In these experiments, the fixed size instruction queue contains 64 entries. The energy shown for the integer benchmarks is for only the integer queue because of the small number of floating point instructions in these benchmarks. The data shown for the floating point benchmarks is for the sum total energy of the integer and floating point queues.

Figure 9 shows the amount of instruction queue energy waste compared to an optimally sized queue. The fraction of instruction queue energy wasted for the integer benchmarks is

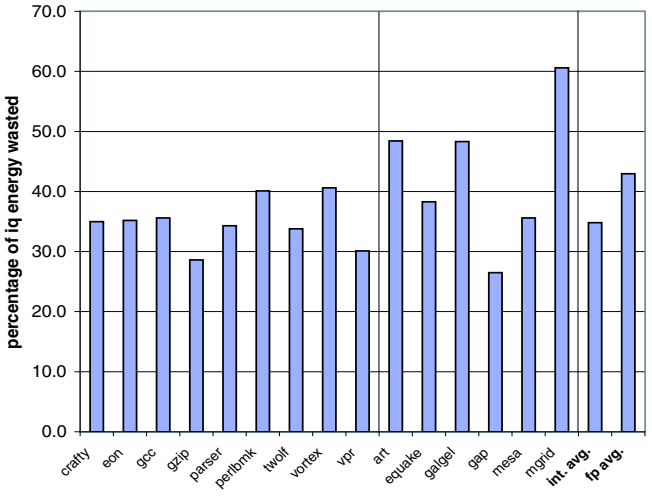

Figure 9. Potential for energy savings available for optimally sized instruction queues.

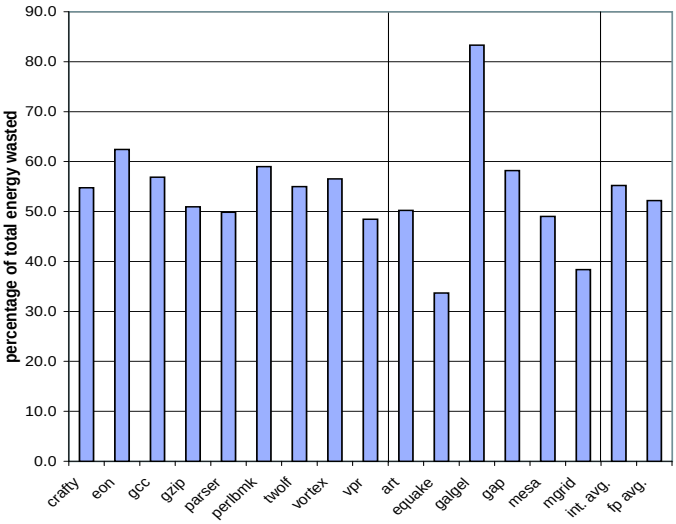

Figure 10. Percentage of total energy wasted due to program and architectural waste.

34.8% on the average. For the floating point benchmarks, the waste is 43.0%.

The fraction of energy wasted by the instruction queue is significant and indicative of the limited parallelism available. In another experiment we found that the difference between reconfiguring the queue size every cycle versus every 50 cycles makes only a 15% difference in the amount of waste. This indicates the waste is more related to longer term effects such as program phases [4] rather than factors like memory latency, and coarse-grain reconfigurability is still useful.

7. Removing Program, Speculation, Architecture Waste

We now show the results of removing program waste, speculation waste, and architecture waste. In these results, we run the simulations removing all energy costs associated with all the instruction types that are considered to be unnecessary computation. In addition, all energy due to speculation and suboptimal structure sizing are removed as well. The predictable redundancy from Section 4, however, is not included in these results.

Figure 10 shows the results for the integer and floating point benchmarks, respectively. For the integer benchmarks, the overall energy waste ranges from 48.4% to 62.4%, with an average of 55.2%. For the floating point benchmarks, the overall energy savings ranges from 33.7% to 83.3%, with an average of 52.2%.

These results indicate that there are certainly significant gains to be had for architecture-level power optimizations. However, it also shows that for a very wide class of these optimizations, the total savings available are not huge. Order of magnitude decreases in energy are not going to come from exploiting wasted execution bandwidth and dynamic reconfiguration of conventional processors. They will need to come from more profound changes to the way we architect processors. Note that our analysis of dynamic resizing is not complete, as we ignore some structures that can be made adaptable. However, the effect of the other structures is not going to dramatically change the conclusion here.

Interestingly, the total energy wasted for the integer and floating point benchmarks is similar, but comes from different sources. For the integer benchmarks, speculation is much more of a factor than for the floating point benchmarks. In both type of benchmarks, the biggest contributor was due to program waste. Regardless of how much speculation is removed from a processor or the sizing of processor structures, the energy cost of executing sequences of unnecessary instructions is large.

The results reported here focused on whole-processor energy. The optimally sized structures, in particular, are limited to a small effect because they only target one piece of the architecture. For environments where power density is the largest power/energy concern, these techniques can have a bigger impact if they specifically target the right component. If these opportunities are exploited in ways that maintain performance, power clearly is reduced as well.

8. Conclusion

In this research, we examine the limits of architecture-level power reduction on a modern CPU architecture. We assume that there are three opportunities to save energy – do not do unnecessary computation (unnecessary speculation or dead computation), do not do redundant computation, and do not power architectural structures that are not needed (eliminate architectural waste via optimal sizing of structures).

This paper quantifies the energy lost to three sources of waste: program waste, speculation waste, and architectural waste. We identify many types of unnecessary instructions and account for the energy cost of those instructions, as well as other instructions that only produce values for unnecessary instructions. In the benchmarks we studied, we demonstrate that 37.7% of the energy used in executing the integer benchmarks is due to program waste. For the floating point benchmarks, 47.0% of the energy is because of program waste. It also shows that to take full advantage of program waste requires eliminating both the redundant instructions and their producers.

Speculation waste averages over 15% for the integer benchmarks, but is much lower for the floating point.

Architectural waste occurs when processor structures are larger than required and cannot be dynamically sized. An optimal instruction cache can consume 55% less energy than a 64KB instruction cache. Similarly, an optimal data cache can use 37% less than the baseline 64KB data cache.

When all of these sources of waste are eliminated, the total energy of the processor has the potential to be reduced by just over a factor of two for both the integer and floating point benchmarks. While this represents a significant opportunity, it also indicates that radical advances in power and energy efficiency require more significant architectural change than the adaptive techniques characterized by this limit study.

References

[1] D. Albonesi. Selective cache ways: on-demand cache resource allocation. In *32nd International Symposium on Microarchitecture*, Dec. 1999.

[2] R. I. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 64–69, New York, Aug. 10–12 1998. ACM Press.

[3] R. I. Bahar, G. Albera, and S. Manne. Using confidence to reduce energy consumption in high-performance microprocessors. In *International Symposium on Low Power Electronics and Design 1998*, Aug. 1998.

[4] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *28th Annual International Symposium on Computer Architecture*, May 2001.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, Dec. 2000.

[6] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[7] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *HPCA1999*, Jan. 1999.

[8] 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*, June 2000.

[9] M. Burtsher and B. Zorn. Exploring last n value prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[10] A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[11] A. Buyuktosunoglu, D. Albonesi, P. Bose, P. Cook, and S. Schuster. Tradeoffs in power-efficient issue queue design. In *International Symposium on Low Power Electronics and Design*, Aug. 2002.

[12] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems*, Nov. 2000.

[13] B. Calder and G. Reinman. A comparative study of load speculation architectures. *Journal of Instruction Level Parallelism*, May, 2000.

[14] D. Folegnani and A. Gonzalez. Reducing power consumption of the issue logic. In *Workshop on Complexity-Effective Design*, May 2000.

[15] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[16] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *25th International Symposium on Microarchitecture*, Dec. 1992.

[17] S. Ghiasi, J. Casmira, and D. Grunwald. Using ipc variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity-Effective Design*, June 2000.

[18] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[19] H. Kim, A. K. Somani, and A. Tyagi. A reconfigurable multifunction computing cache architecture. In *IEEE Transactions on Very Large Scale Integration Systems*, Aug. 2001.

[20] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[21] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[22] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[23] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[24] M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *30th International Symposium on Microarchitecture*, Dec. 1997.

[25] S. Onder and R. Gupta. Load and store reuse using register file contents. In *15th International Conference on Supercomputing*, June 2001.

[26] H. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, Feb. 2002.

[27] M. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi , and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[28] S. Raasch, N. Binkert, and S. Reinhardt. A scalable instruction queue design using dependence chains. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[29] P. Ranganathan, S. Adve, and N. Jouppi. Reconfi gurable caches and their application to media processing. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[30] E. Rotenberg. Exploiting large ineffectual instruction sequences. Technical report, North Carolina State University, 1999.

[31] J. Seng, D. Tullsen, and G. Cai. Power-sensitive multithreaded architecture. In *International Conference on Computer Design 2000*, Sept. 2000.

[32] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[33] A. Sodani and G. Sohi. Dynamic instruction reuse. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[34] A. Sodani and G. S. Sohi. An empirical analysis of instruction repetition. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[35] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[36] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip paralle lism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[37] D. Tullsen and J. Seng. Storageless value prediction using prior register values. In *26th Annual International Symposium on Computer Architecture*, pages 270–279, May 1999.

[38] J. Yang and R. Gupta. Energy-effi cient load and store reuse. In *International Symposium on Low Power Electronic Design*, Aug. 2001.

[39] S.-H. Yang, M. D. Powell, B. Falsafi , K. Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Seventh International Symposium on High Performance Computer Architecture*, Jan. 2001.

[40] S.-H. Yang, M. D. Powell, B. Falsafi , and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Eighth International Symposium on High Performance Computer Architecture*, Feb. 2002.

[41] A. Yoaz, R. Ronen, R. S. Chappell, and Y. Almog. Silence is golden? In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan. 2001.