# Value-Based Clock Gating and Operation Packing: Dynamic Strategies for Improving Processor Power and Performance

DAVID BROOKS and MARGARET MARTONOSI
Princeton University

The large address space needs of many current applications have pushed processor designs toward 64-bit word widths. Although full 64-bit addresses and operations are indeed sometimes needed, arithmetic operations on much smaller quantities are still more common. In fact, another instruction set trend has been the introduction of instructions geared toward subword operations on 16-bit quantities. For example, most major processors now include instruction set support for multimedia operations allowing parallel execution of several subword operations in the same ALU. This article presents our observations demonstrating that operations on "narrow-width" quantities are common not only in multimedia codes, but also in more general workloads. In fact, across the SPECint95 benchmarks, over half the integer operation executions require 16 bits or less. Based on this data, we propose two hardware mechanisms that dynamically recognize and capitalize on these narrow-width operations. The first, power-oriented optimization reduces processor power consumption by using operand-value-based clock gating to turn off portions of arithmetic units that will be unused by narrow-width operations. This optimization results in a 45%-60% reduction in the integer unit's power consumption for the SPECint95 and MediaBench benchmark suites. Applying this optimization to SPECfp95 benchmarks results in slightly smaller power reductions, but still seems warranted. These reductions in integer unit power consumption equate to a 5%–10% full-chip power savings. Our second, performance-oriented optimization improves processor performance by packing together narrow-width operations so that they share a single arithmetic unit. Conceptually similar to a dynamic form of MMX, this optimization offers speedups of 4.3%–6.2% for SPECint95 and 8.0%–10.4% for MediaBench. Overall, these optimizations highlight an increasing opportunity for value-based optimizations to improve both power and performance in current microprocessors.

Categories and Subject Descriptors: B.2 [**Hardware**]: Arithmetic and Logic Structures; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*RISC/CISC, VLIW architectures*

General Terms: Design, Experimentation, Performance

This research was supported in part by research funds from DARPA grant DABT63-97-C-1001 and NSF grant MIP-97-08624 and a donation from Intel Corp. In addition, Margaret Martonosi is partially supported by an NSF CAREER Award, and David Brooks is supported by an NSF Graduate Research Fellowship.
Authors' address: Department of Electrical Engineering, Princeton University, Princeton, NJ.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2000 ACM 0734-2071/00/0500–0089 $5.00

Additional Key Words and Phrases: power-efficient microarchitecture, clock gating, operation packing, multimedia instruction sets, bitwidth characterization, narrow-width operations

## 1. INTRODUCTION

In recent years there has been a shift toward 64-bit instruction sets in major commercial microprocessors. The increased word widths of these processors were largely motivated because addresses were getting larger; however, the size of the actual data has not increased as quickly. As high-end processor word widths have made the shift from 32 to 64 bits, there has been an accompanying trend toward efficiently supporting subword operations. Subword parallelism, in which multiple 8- or 16-bit operations are performed in parallel by a 64-bit ALU, is supported in current processors via instruction set and organizational extensions. These include the Intel MMX [Peleg and Weiser 1996], HP MAX-2 [Lee 1996], and Sun VIS [Tremblay et al. 1996] multimedia instruction sets, as well as vector microprocessor proposals such as the T0 project [Asanovic et al. 1996].

All of these ideas provide a form of SIMD (single instruction-multiple data) parallel processing at the word level. These instruction set extensions are focused primarily on enhancing performance for multimedia applications. Such applications perform large amounts of arithmetic processing on audio, speech, or image samples which typically only require 16 bits or less per datum. The caveat to this type of processing is that thus far these new instructions are mainly used only when programmers hand-code kernels of their applications in assembler. Little compiler support exists to generate them automatically, and the compiler analysis is limited to cases where programmers have explicitly defined operands of smaller (i.e., char or short) sizes.

This article proposes hardware mechanisms for dynamically exploiting narrow width operations and subword parallelism without programmer intervention or compiler support. By detecting “narrow bitwidth” operations dynamically, we can exploit them more often than with a purely static approach. Thus, our approach will remain useful even as compiler support improves.

In this work we explore two optimizations that take advantage of the core “narrow width operand” detection that we propose. For both techniques, we explore both a basic and extended version of the optimization. The basic approach only operates in cases that it is guaranteed to succeed. In the extended version of the proposals, we demonstrate speculative techniques that can improve the efficiency of the optimizations.

The first optimization that we propose watches for small operand values and exploits them to reduce the amount of power consumed by the integer unit. This is accomplished by an aggressive form of clock gating. Clock gating has previously been shown to significantly reduce power consump-

tion by disabling certain functional units if instruction decode indicates that they will not be used [Gonzalez and Horowitz 1996]. The key difference of our work is to apply clock gating based on operand values. When the full width of a functional unit is not required, we can save power by disabling the upper bits. With this method we show that the amount of power consumed by the integer execution unit can be reduced for the SPECint95 suite with little additional hardware.

The second proposed optimization improves performance by dynamically recognizing, at issue time, opportunities for packing multiple narrow operations into a single ALU. With this method the SPECint95 benchmark suite shows an average speedup of 4.3%–6.2% depending on the processor configuration. The MediaBench suite showed an average speedup of 8.0%–10.4%.

The primary contributions of this work are threefold: a detailed study of the bitwidth requirements for a wide range of benchmarks, and two proposals for methods to exploit narrow width data to improve processor power consumption and performance. In Section 2 we further discuss the motivations for our work and place it in the context of prior work in multimedia instruction sets, power savings, and other methods of using dynamic data. Section 3 describes the experimental methodology used to investigate our optimizations. Section 4 details the power optimization technique based on clock gating for operand size and presents results on its promise. In Section 5, we describe the method for dynamically packing narrow instructions at issue-time. In Section 6, we describe speculative techniques to improve the benefits of the first two optimizations. Finally, Section 7 concludes and discusses other opportunities to utilize dynamic operand size data in processors.

## 2. MOTIVATION AND PAST WORK

### 2.1 Application Bitwidths

In this study we show that a wide range of applications are frequently calculated using small operand values. Figure 1 illustrates this by showing the cumulative percentage of integer instructions in SPECint95 in which both operands have values that can be expressed using less than or equal to the specified bitwidth. (Section 3 will discuss the Alpha compiler and SimpleScalar simulator used to collect these results.) Roughly 50% of the instructions had both operands less than or equal to 16 bits. We will refer to these operands as narrow width; an instruction execution in which both operands are narrow width is said to be a "narrow-width operation." Since this chart includes address calculations, there is a large jump at 33 bits. This corresponds to heap and stack references. (Larger programs than SPEC might have this peak at a larger bitwidth.) The data demonstrate the potential for a wide range of applications, not just multimedia applications, to be optimized based on narrow-width operands. While other such work,

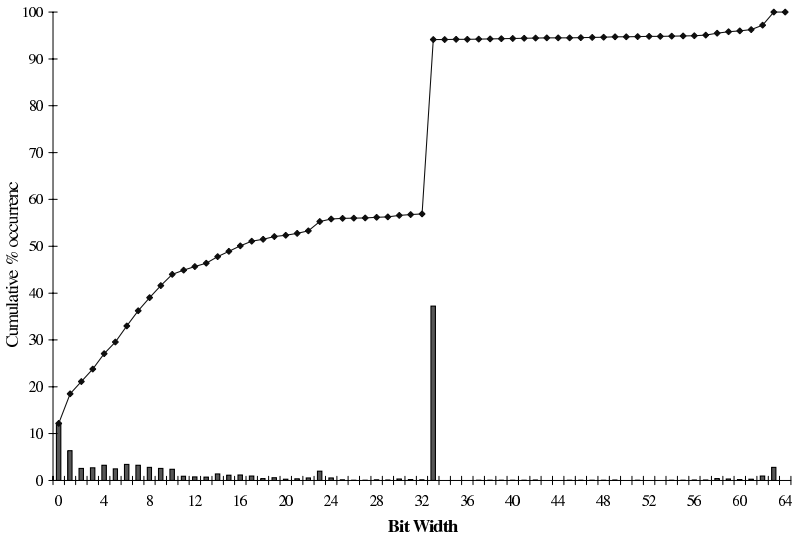


Fig. 1. Bitwidths for SPECint95 on 64-bit Alpha.

e.g., protein-matching [Alpern et al. 1995], required algorithm or compiler changes, we focus here on hardware-only approaches.

### 2.2 Observing and Optimizing Narrow Bitwidth Operands

The basic tenet behind both of the optimizations proposed here is that when operations are performed with narrow-width operands, the upper bits of the operation are unneeded. To decrease power dissipation, clock gating can disable the latch for these unneeded upper bits. Alternatively, to improve performance, we propose "operation packing," in which we issue and execute several of these narrow operations in parallel within the same ALU. In either case, the crux in exploiting narrow-width operands lies in recognizing them and modifying execution. Sections 4 and 5 will discuss hardware approaches for tagging result operands as "narrow-width" as they are produced, and for storing these tags along with source operands as we stage subsequent instructions waiting for issue.

### 2.3 Disadvantages of Static Compiler Analysis

Part of the motivation for this work was the fact that *static* analysis of input operand sizes has several disadvantages. First, RISC instruction sets, such as the Alpha instruction set that we consider in this study, typically do not include instructions that specify the operand size information for each operation. For example, the Alpha ISA does not include add instructions that operate on 8-bit or 16-bit quantities. Thus the compiler could not embed operand size information without instruction set extensions. More importantly there are many cases where it is impossible to know what the true operand bitwidths (as opposed to the declared operand sizes) will be until run-time. Actual operand sizes depend very much on the input data presented. Operand sizes for particular instructions can also vary over the program run even with the same input data, which makes the task of the compiler even more difficult.

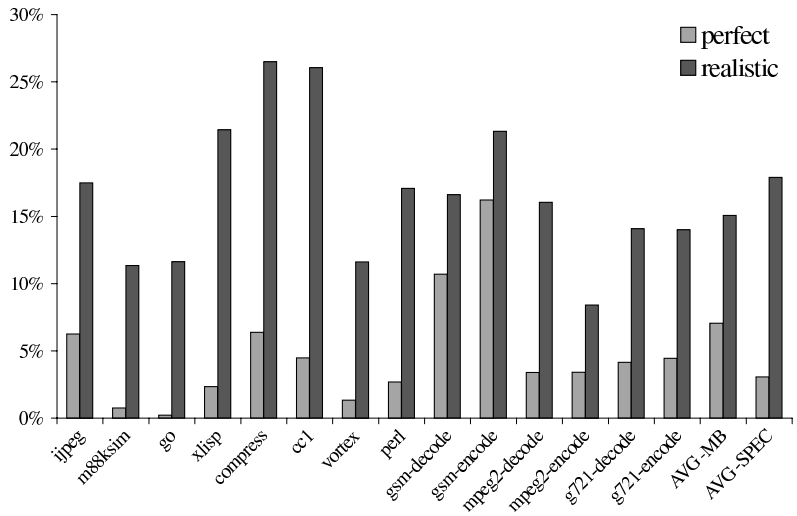


Fig. 2. Percentage of instructions whose operand precision changes from less than 16-bit to greater than 16-bit over a single program run. Data is presented for both perfect and realistic branch prediction.

Figure 2 shows the percentage of PC values where operand width changes as the instruction is executed repeatedly within a single run. In particular, the figure shows how often an instruction fluctuates from having less than 16-bit operands to greater than 16-bit operands as it executes repeatedly within a single program run. Figure 2 thus demonstrates some of the difficulty that a compiler would encounter in determining the operand-widths of operations statically. In particular, it is interesting to note that with perfect branch prediction, the instruction operand sizes are far more predictable than with realistic branch prediction. This is because with perfect branch prediction only the true execution path is seen. With imperfect branch prediction, uncommon paths, like error conditions, may be executed (but not committed) if the branch predictor points that way. Along these paths, operand statistics may be markedly different. Compile time analysis must conservatively analyze all potential paths to ensure that operations can truly be packed. This may include uncommon error conditions and other extreme cases. As a result, the compiler runs into much of the same diverse operand values as seen by imperfect branch prediction.

Overall, compiler dataflow analysis for operand sizes must be conservative about possible operand values. Programmer hints about operand sizes can aid the compiler. It is unrealistic, however, to assume that programmers will provide these hints on codes other than small multimedia kernels.

From Figure 1 it is clear that many opportunities exist to exploit narrow-width data for subword parallelism and aggressive clock gating. Searching for subword parallelism in applications is somewhat analogous to the search for instruction-level parallelism (ILP) in applications. In the late 80's and early 90's, most general-purpose superscalar microprocessors

were statically scheduled, and the compiler was responsible for uncovering ILP in programs. Current microprocessors implement aggressive dynamic scheduling techniques to uncover more ILP. This evolution was necessary to feed the wider-issue capabilities of these processors. In a similar manner, more subword parallelism can be uncovered with the dynamic approaches we propose than if one relies solely on compiler techniques.

### 2.4 Related Work

The notion of disabling the clocks to unused units to reduce power dissipation in high performance microprocessors has been discussed in Gowan et al. [1998] and Tiwari et al. [1998b]. In the CAD community, similar techniques have been demonstrated at the logic level of design. Guarded evaluation seeks to dynamically detect which parts of a logic circuit are being used and which are not [Tiwari et al. 1998a]. Logic precomputation seeks to derive a precomputation circuit that under special conditions does the computation for the remainder of the circuit [Alidina et al. 1994]. Both of these techniques are analogous to conditional clocking, which can be used at the architectural level to reduce power by disabling unused units.

There has been other work in specializing for particular operand values at runtime. The PowerPC 603 includes hardware to count the number of leading zeros of input operands to provide an "early out" for multicycle integer multiply operations. This can reduce the number of cycles required for a multiply from five for 32-bit multiplication to two for an 8-bit multiplication [Gerosa et al. 1994]. At a higher level, value prediction seeks to predict result values for certain operations and speculatively execute additional instructions based on these predicted operand values [Lipasti et al. 1996]. Memoing is another high-level technique that exploits data redundancy to eliminate power dissipation of long-latency integer and floating-point operations [Azam et al. 1997]. Memoing is the idea of storing the inputs and outputs of long-latency operations and reusing the output if the same inputs are encountered again.

Finally, there has also been other work in exploiting narrow bitwidth operations. Razdan and Smith propose a hardware-programmable functional unit which augments the base processor's instruction set with additional instructions that are synthesized in configurable hardware at compile time [Razdan and Smith 1994]. Since all synthesized instructions must complete in a single cycle, bitwidth analysis is performed at compile time to highlight sequences of narrow-width operations that are the best candidates for implementation.

Tong et al. [1998] have proposed sacrificing computational accuracy for reduced power consumption. Their analysis shows that certain floating-point programs suffer very little loss of accuracy with a significant reduction in bitwidth. They propose minimizing the bitwidth representation of floating-point data to reduce power consumption in the floating-point unit. Our work differs from this technique, because we include hardware structures to dynamically detect opportunities to capitalize on narrow bitwidth

Table I. Baseline Configuration of Simulated Processor: Processor Core

| Parameter | Value |
|---|---|
| RUU (register update unit) size | 80 instructions |
| LSQ (load store queue) size | 40 instructions |
| Fetch Queue Size | 8 instructions |
| Fetch width | 4 instructions/cycle |
| Decode width | 4 instructions/cycle |
| Issue width | 4 instructions/cycle (out-of-order) |
| Commit width | 4 instructions/cycle (in-order) |
| Functional Units | 4 Integer ALUs (performing arithmetic, logical, shift, memory, branch, and shift operations)<br>1 integer multiply/divide<br>1 FP add, 1 FP multiply, 1 FP divide/sqrt |

operations ensuring that program will produce the same results as without the optimization.

## 3. METHODOLOGY

In Sections 4, 5, and 6 of this article we present the results for the power and performance optimizations that we propose for dynamically exploiting small operand values. This section lays the groundwork by detailing the experimental methodology used for obtaining those results.

### 3.1 Simulator

We have used a modified version of SimpleScalar's *sim-outorder* to collect our results. SimpleScalar provides a simulation environment for modern out-of-order processors with 5-stage pipelines: fetch, decode, issue, write-back, and commit. Speculative execution is also supported. The simulated processor contains a unified active instruction list, issue queue, and rename register file in one unit called the reservation update unit (RUU) [Sohi and Vajapeyam 1987]. The RUU is similar to the Metaflow DRIS (deferred-scheduling, register-renaming instruction shelf) [Popescu et al. 1991] and the HP PA-8000 IRB (instruction reorder buffer) [Hunt 1995]. Separate banks of 32 integer and floating-point registers make up the architected register file and are only written on commit. Tables I–III summarize the important features of the simulated processor. The baseline configuration parameters roughly match those of a modern out-of-order processor.

Most of the changes made to the simulator for this study are localized to the issue and decode stages. In the decode stage, bitwidths are calculated for dynamic data and stored in the reservation station entry to be used during the issue stage. In the issue stage, this data is used to decide if instructions can be issued and executed in parallel based on the data from the decode stage. While these changes reflect the simulator implementation, subsequent sections discuss how our ideas would be implemented in an actual processor.

Table II. Baseline Configuration of Simulated Processor: Branch Prediction

| Parameter | Value |
|---|---|
| Branch Predictor | Combined, Bimodal 4K table, 2-Level 1K table, 10-bit history 4K chooser |
| BTB | 2048-entry, 2-way |
| Return-address stack | 32-entry |
| Mispredict penalty | 2 cycles |

Table III. Baseline Configuration of Simulated Processor: Memory Hierarchy

| Parameter | Value |
|---|---|
| L1 data-cache | 64K, 2-way (LRU), 32B blocks, 1 cycle latency |
| L1 instruction-cache | 64K, 2-way (LRU), 32B blocks, 1 cycle latency |
| L2 | Unified, 8M, 4-way (LRU), 32B blocks, 12-cycle latency |
| Memory | 100 cycles |
| TLBs | 128 entry, fully associative, 30-cycle miss latency |

### 3.2 Benchmark Applications

A goal of this study is to demonstrate and exploit the prevalence of operations with narrow bitwidths even in applications outside the multimedia domain. For this reason we evaluate the SPECint95 suite of benchmarks as well as several benchmarks from the MediaBench suite [Lee et al. 1997]. For the power optimization we also consider eight of the SPECfp95 benchmarks.

We have compiled the benchmarks using the DEC/Compaq *cc* compiler with the following optimization options as specified by the SPEC Makefile: -migrate -std1 -O5 -ifo -non_shared. In particular, the -O5 setting, along with numerous other optimizations, provides vectorization of some loops on 8-bit and 16-bit data (char and short).

For this study we used the reference inputs for the SPEC95 suite. We did not want to use the test or training inputs because our data-specific optimizations might be unfairly helped by smaller data sets. Using the reference inputs, the SPEC95 benchmarks run for billions of instructions, which, if simulated fully, would lead to excessively long execution times. Thus we have adopted a methodology similar to that described in Skadron et al. [1999]. We warm up the architectural state using a fast-mode cycle-level simulation that updates only the caches and branch predictors during each cycle. The warmup period also avoids the effects of smaller operand sizes that are prevalent within program initialization. Using the results of Skadron et al. [1999] to identify representative sections of the program run based on cache and branch prediction statistics, we then simulate a 100 million instruction window using the detailed simulator. Table IV lists the reference input that we have chosen for the SPEC95 benchmarks, and the number of instructions for which we warm up the caches and branch predictor. Table IV also describes the applications chosen from the MediaBench suite. For the MediaBench suite, *gsm*, *g721*,

Table IV. Characteristics of the SPEC95 and MediaBench Benchmarks Studied

| Benchmark | Family | Number of Warmup Instructions or Description | Input Data |
|---|---|---|---|
| cc1 | SPECint | 221M | cccp.i |
| perl | SPECint | 601M | scrabble game |
| ijpeg | SPECint | 824M | vigo.ppm |
| compress | SPECint | 2576M | bigtest.in |
| m88ksim | SPECint | 26M | dhrystone |
| li | SPECint | 271M | All inputs |
| vortex | SPECint | 2451M | persons.1k |
| go | SPECint | 926M | 9stone21 |
| applu | SPECfp | 1410M | applu.in |
| apsi | SPECfp | 1400M | apsi.in |
| fpppp | SPECfp | 1000M | natoms.in |
| hydro2d | SPECfp | 375M | hydro2d.in |
| mgrid | SPECfp | 1410M | mgrid.in |
| su2cor | SPECfp | 2500M | su2cor.in |
| turb3d | SPECfp | 1000M | turb3d.in |
| wave5 | SPECfp | 1410M | wave5.in |
| adpcm | Media | 16 bit PCM < − > 4-bit ADPCM coder | clinton.pcm |
| mpeg2 | Media | MPEG digital compressed format encoding | rec%d |
| gsm | Media | Audio and speech encoding with GSM std. | clinton.pcm |
| g721 | Media | Voice compression using the G.721 standard | clinton.pcm |

and *mpeg2-decode* were run to completion while *mpeg2-encode* was simulated for 100 million instructions after a 500M-instruction warmup period.

## 4. POWER OPTIMIZATIONS

### 4.1 Clock Gating

Dynamic power dissipation is the primary source of power consumption in CMOS circuits. In CMOS circuits, dynamic power dissipation occurs when changing input values cause their corresponding output values to change. Only small leakage currents exist as long as inputs are held constant. Clock gating has been used to reduce power by disabling the clock and thereby disabling value changes on unneeded functional units. In static CMOS circuits, disabling the clock on the latch that feeds the input operands to functional units essentially eliminates dynamic power dissipation. Power consumption on the critical clock lines is also saved because the latch itself is disabled. In dynamic or domino CMOS circuits, the same effect can be obtained by disabling the clocks that control the precharge and evaluate phases of the circuit. The use of clock gating may introduce additional clock skew and can complicate timing analysis which provide challenges for circuit designers performing the implementation. Despite these difficulties, conditional clocking is commonly used in current microprocessors [Gowan et al. 1998].

Currently most work on clock gating has focused on using the decoded opcode to decide which units can be disabled for a particular instruction.

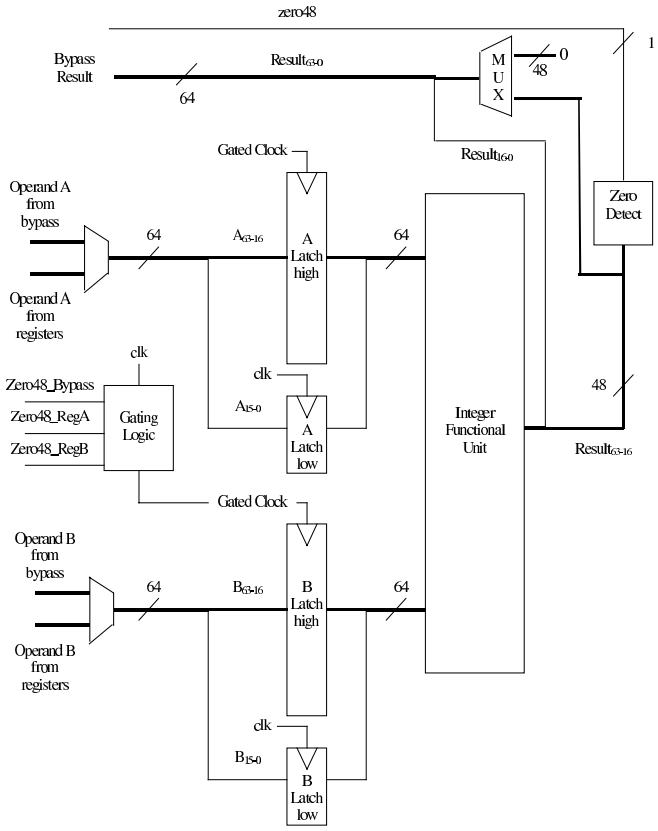


Fig. 3. Clock gating architecture.

For example, *nop*'s allow most of the units to be disabled since no result is being computed. As another example of opcode-based clock gating, consider an "add byte" instruction. Since the opcode *guarantees* that only the lower portion of the adder is needed, the top part of the functional unit can be disabled.

4.1.1 *Proposed Architecture.* Our approach proposes a more aggressive clock gating approach and quantifies its benefits. At run-time, it determines instances when, based on the input operands, the upper bits of an operation are not needed; in those cases, it disables the upper portion of the functional unit. The key differences from prior approaches are that (1) our approach is operand-based, not opcode-based, and (2) our approach is dynamic, not static. (One could, of course, use our method *in addition to* prior opcode-based approaches.) Different runs of the program, or even different executions of the same instruction, can dissipate different amounts of power depending on the operands seen.

There are several different possible hardware implementations for this technique. Figure 3 is a diagram of one possible implementation. This unit recognizes that the upper bits of both input operands are zeros. For example, in an addition operation, if both input operands have all zeros in their top 48 bits, these bits do not have to be latched and sent to the

functional units. We already know that the result of this part of the addition will be zero, and thus zeros can be multiplexed onto the top 48 bits of the result bus, rather than computed via the adder. In this architecture the low 16 bits are always latched normally. The high 48 bits are selectively latched based on a signal that accompanies the input operand from the reservation stations or the bypass network. This signal, called *zero48* in Figure 3, denotes that the upper 48 bits are all zeros and is created by zero detection logic when the result was computed. Since some operands come directly from the cache, there must also be a zero-check during load instructions. We believe such zero-detect hardware and corresponding flags within the reservation stations are already present in some processors; for example, to recognize divide-by-zero exceptions early. However, in some processors it may not be possible to perform zero-detects on incoming loads, and in these cases the hardware will not recognize an opportunity to gate the clock. For the SPECint95 suite, 13.1% of power saving instructions have one or more operands that come directly from a load instruction; these are the instructions that would be missed if zero-detect were omitted on loads. The percentages for the media benchmarks are much lower at 1.5%.

The gated clock signal used to disable the upper 48 bits of the functional unit is generated based on the *zero48* signals of the respective operands and is combined with an AND gate in parallel with data bypass muxing. In the case of functional units designed with static logic this signal can be used to disable the upper 48 bits of the preceding latches thus effectively reducing the switching activity to zero. For functional units design with dynamic logic, the *zero48* signal would be placed into the latches and used in the next cycle to disable the clock on the upper 48 bits of the functional unit.

In Figure 3 the *zero48* signal is generated after the functional unit completes the specified operation. In processors with architecturally visible zero-flags such as the Intel x86, Motorola 68K, and IBM/Motorola PowerPC architectures, this approach would be feasible because there would be no additional serial delay introduced. However, in other architectures in which adding a zero-detect in the execute stage would affect cycle time, another implementation is possible. This implementation relies on the fact that if we know that the two source operands of an operation are 16 bits or less, then it is relatively easy to determine whether or not the result will be 16 bits or less. For example, with an arithmetic operation, if the carry-out signal of the 16th bit is zero and the two source operands are 16 bits or less, then we know that the result will be 16 bits or less. Thus, the *zero48* signal can be computed after the carry-out of the 16th bit is generated, well before the final adder result is finished. Finally, in some cases a designer might not want to insert the *zero48* signal into the register file or reservation stations. In this case, the 48-bit zero-detects could be inserted after register fetch while waiting for the bypass results to be returned. This relies on the fact that register read generally takes place in the first half the cycle and writeback occurs in the second half of the cycle.

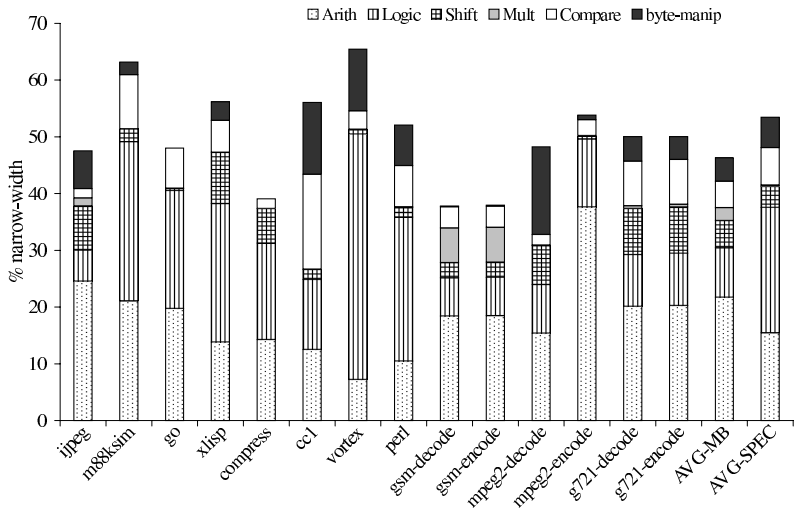


Fig. 4. Operations with both operands 16 bits or less.

In order for any power saving technique to be useful, it must save more power than it consumes. In our technique, the new power dissipated is mainly in the zero-detection logic and in widening the mux onto the result bus. The primary power savings stems from selectively clock-gating the functional units based on the results of the zero-detection logic. In the following subsections we evaluate these costs and benefits in more detail.

4.1.2 *Bitwidth Analysis of Benchmarks.* The success of our approach relies on the frequent occurrence of narrow bitwidth operands. Figure 4 shows, for each benchmark, the percentage and type of operations whose input operands are both less than or equal to 16 bits. (Both operands must be small in order for the clock gating to be allowed.) The breakdown by operation type is another important metric. Intuitively, disabling the upper bits on an adder or multiplier will save more power than turning off the upper bits on the less power-hungry logical functions. Figure 4 shows that for most benchmarks arithmetic and logical operations dominate the number of narrow-width operations. In most of the benchmarks multiplies are rather infrequent although they do account for 6% of the narrow-width operations in *gsm*.

Recall that Figure 1 illustrated how address calculations result in many operations with bitwidths of 33. Roughly 94% of SPECint95 compute operations had bitwidth requirements of 33 or less with 37% occurring at the 33-bit mark. From this data it makes sense to include a second control signal for clock gating of operands that are 33 bits or less. The zero detect logic can be shared so that the extra hardware requirements are minimal. This modification is also useful for optimizing the multiplication of two 16-bit numbers. In these cases a 32-bit result can occur, so the 33-bit mux onto the result bus would be used as shown in Figure 5. Figure 5 also shows the expanded clock gating architecture with clock gating at the 16-bit and 33-bit boundaries. The operand latches have been further partitioned and

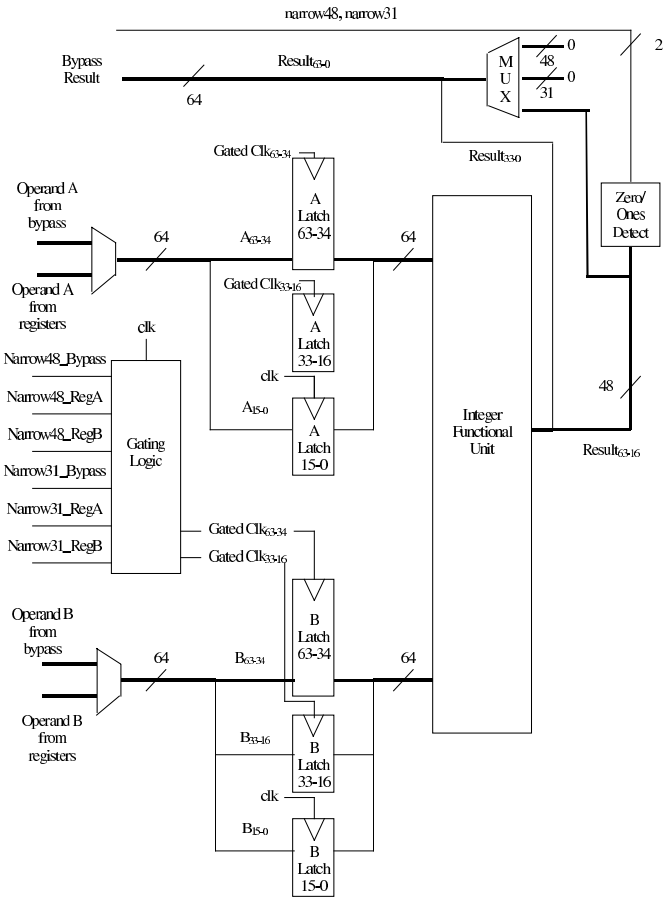


Fig. 5. Expanded clock gating architecture with 33-bit gating.

an additional clock gating signal is generated. In Sections 4.3 and 4.4 we discuss the choice of the bitwidths to clock gate in more detail.

Negative numbers provide another source of narrow-width data for operand-based clock gating to exploit. In the Alpha architecture that we considered in this study, the fundamental datum is the 64-bit quadword. Quadword integers are represented with a sign bit occupying the most significant bit [Bhandarkar 1996]. Numbers are expressed in two's complement form which simplifies arithmetic operations. The techniques presented in this article rely on determining when data requires less than the full word width of the machine. For positive numbers, this can be accomplished by performing a zero detect on the high order bits. For negative numbers in the two's complement representation, leading 1's signify the same thing that leading 0's do for positive number—essentially unneeded data. Thus a ones detect computation (simply an AND of the high-order bits) must be performed in parallel with the zero detect computation to detect narrow bitwidth negative numbers. An additional signal does not need to be stored in the register file because this information can be derived by sampling one of the higher order bits. Figure 5 shows the zero

Table V. Estimated Power Consumption of Functional Units at 3.3V and 500MHz (mW)

| Device | 32-bit | 48-bit | 64-bit |
|---|---|---|---|
| Adder (CLA) | 105 | 158 | 210 |
| Booth Multiplier | 1050 | 1580 | 2100 |
| Bitwise Logic | 5.8 | 8.7 | 11.7 |
| Shifter | 4.4 | 6.6 | 8.8 |
| Zero-Detect | — | 4.2 | — |
| Additional Muxes | — | 3.2 | — |

and ones-detect logic which creates the signals *narrow31* and *narrow48* (analogous to the *zero48* from Figure 3).

### 4.2 Power Results: Overview

The amount of power that is saved by our approach depends on both the type and frequency of narrow-width operations. In order to quantify the amount of power saved, we use previously reported research to estimate the amount of power that various functional units use [Borah et al. 1996; Ng et al. 1996; Zimmermann and Fichtner 1997; Callaway and Swartzlander 1997]. From these sources we obtain power estimates assuming dynamic logic and relatively fast carry look-ahead adders. We assume that the power scales linearly with the number of bits of the units based [Nagendra et al. 1996]. We assume that the multiplier is pipelined with its power usage scaling linearly with the operand size. Again, the zero-detect for 33 bits can be computed within the 48-bit zero-detect so no additional power is consumed. Table V summarizes the values that we have assumed for different size devices. The functional units in current high-end microprocessors are likely to use even more power, but detailed numbers are not yet available in the literature. For this analysis though, the important factor is the ratio of the respective functional units to each other.

Figure 6 summarizes the amount of power saved and expended by the integer execution units. We arrived at these numbers by determining the amount of power saved and expended per operation executed and multiplying by the average issue rate. These results include all loads, stores, branches, and other integer execution unit instructions that are not part of the set of instructions that our optimization applies to. Among the SPECint95 benchmarks, our technique saves the most power for *ijpeg* and *go*. *Ijpeg* has a large number of narrow-width arithmetic operations. *Go* includes a large number of address calculations and is helped the most by adding the extra signal to detect 33-bit operations. The media benchmarks tend to save even more power than the SPECint95 benchmarks. This is primarily because of the larger number of arithmetic operations. *GSM*, in particular, has a relatively large number of narrow bitwidth multiply operations. The amount of power used by the zero detection circuitry is small and nearly constant for all benchmarks. In no case does the amount of power used for zero detection exceed the amount of power saved.

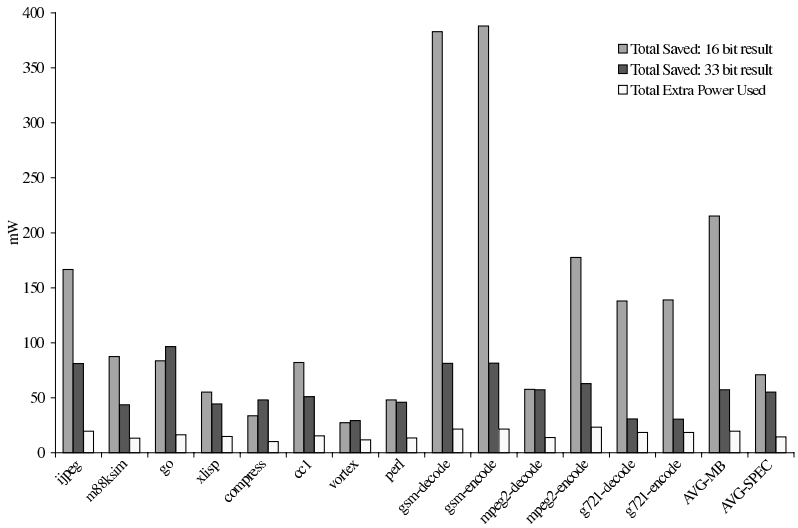


Fig. 6. Net power saved by clock gating at 16 and 33 bits. Total extra used is the amount used by zero detection and muxing. Net savings is equal to the amount saved at 16 bits plus the amount saved at 33 bits minus the amount used.

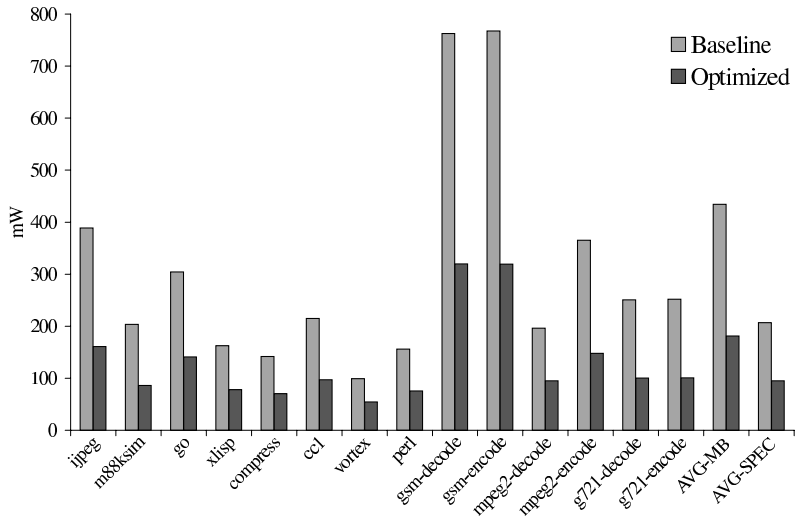


Fig. 7. Power usage of integer unit.

Figure 7 shows the total amount of power that is saved by the integer unit with our optimization. For the baseline system, we assume that all operations use the amount of power that a 64-bit device would use. (We assume basic clock gating in which, for example, multipliers are turned off for add instructions and vice versa.) For the SPECint95 benchmark suite, the average power consumption of the integer unit was reduced by 54.1%. For the media benchmarks, the reduction was 57.9%.

While a 50–60% power reduction seems exceptional, it is important to note that the integer unit's contribution to total power varies depending on the CPU. In some high-end CPUs much of the power is spent on clock distribution and control logic, and thus the integer unit represents only

about 10% of the power dissipation [Gowan et al. 1998]. In such a processor, our optimizations will lead to 5–6% power reductions on average. As control is streamlined, either in DSPs or via explicitly parallel instruction computing (EPIC) as in future Intel processors [Dulong 1998], the integer unit is a larger factor in the processor's total power dissipation, as much as 20–40% [Kojima et al. 1996]. In these cases, the total power savings from our technique will approach 20%. In all processors, our approach promises a relatively easy way to prune power from the integer unit where this is important. We also note that our power savings estimates are somewhat conservative. The clock gating technique also reduces the switch capacitance seen by the clock distribution network, and this can lead to a further power reduction. Although this effect can be significant, it cannot be quantified without a chip floorplan.

### 4.3 Selection of Gating Boundaries

In the previous subsections, data has been presented for clock gating at 16-bit and 33-bit boundaries. The choice of the 33-bit mark was motivated because the empirical data demonstrated that a large number of operations exist with both source operands 33 bits or less, primarily due to address calculations. The reason for choosing the 16-bit mark is more arbitrary and reflects the need to balance two trade-offs in the selection of the boundary at which to clock gate. First, if the boundary is chosen to be too large, the amount of power saved will not be as significant as possible. On the other hand, if the boundary is selected to be too small, not enough operations will be eligible for clock gating at that boundary.

In this subsection, we systematically investigate the selection of the clock gating boundary. In this analysis, we limit the number of boundaries that are clock gated to one or two. We also assume that the power dissipation of the functional units scales linearly at the bit-level. In the next section, we investigate the potential for clock gating at more than two points with finer granularities.

Figure 8 shows the integer unit power reduction by having one clock gating boundary at the specified bitwidths. The data is shown as a percentage power reduction relative to the original integer unit power. Clearly, if we are only allowed to clock gate at one point, we should clock gate at 33 bits. Clock gating at points beyond 33 bits does not make sense for this set of benchmarks, because they rarely utilize the upper portion of the functional units. Section 4.5 will discuss floating-point benchmarks in more detail. We also note that future applications written for 64-bit CPUs may use larger values more frequently, but we typically expect this usage to grow slowly from the 33-bit mark as addressing needs grow.

Figure 9 shows the power savings assuming that we now are able to clock gate at two points. One of the two points is always chosen to be 33 bits, capturing the large number of address calculations. The second point varies, with each bar measuring the total amount of power saved by clock gating at that bitwidth as well as at 33 bits. Figure 9 demonstrates that

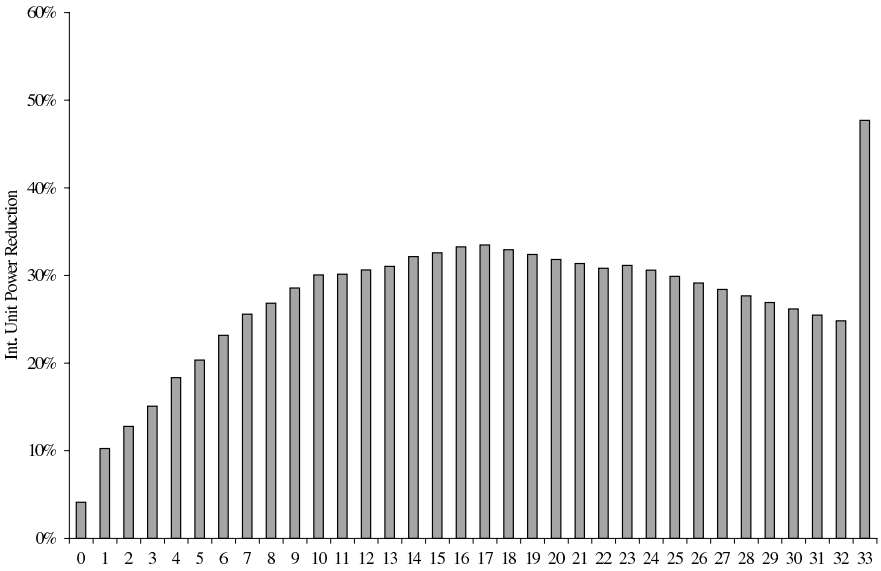


Fig. 8. Integer unit power reduction by selecting to gate at one bitwidth.

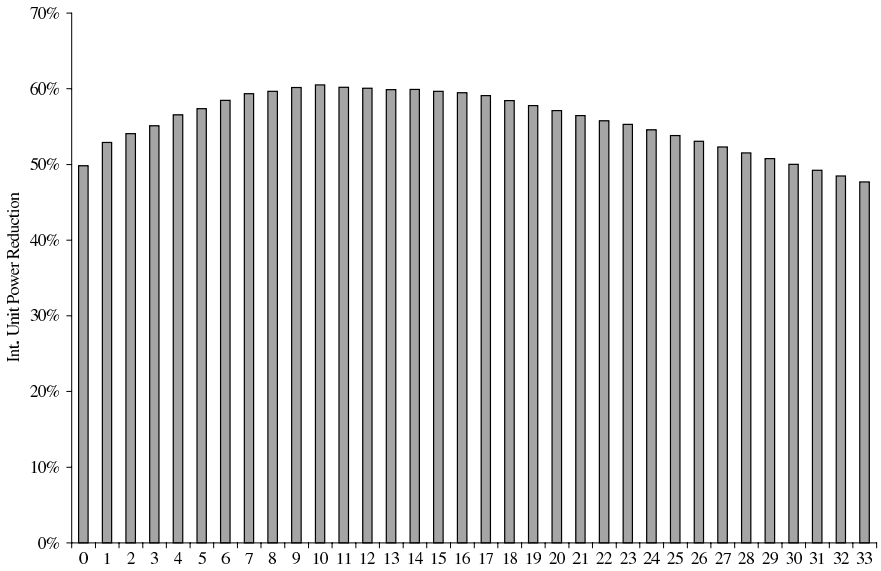


Fig. 9. Integer unit power reduction by clock gating at 33 bits as well as at the specified bitwidth.

choosing the clock gating point to be anywhere from 10 to 17 results in very little difference in the total amount of power saved. Thus our original choice of clock gating at 16 bits was reasonable. On the other hand, certain benchmarks display a preference for clock gating at a particular bitwidth. This can affect the total amount of power saved significantly. For example, the optimal selection of clock gating boundary for *m88ksim* is 5 bits. Clock gating at this bitwidth would save approximately 10% more power than our default selection of 16 bits.

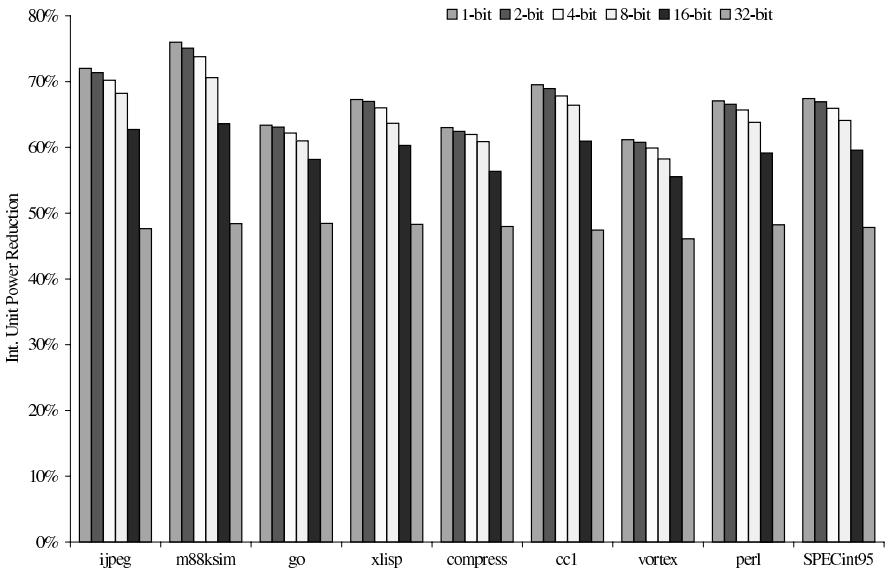


Fig. 10. Percent of integer unit power saved with varying clock gating granularities.

## 4.4 Selecting the Number of Clock Gate Boundaries

In the previous subsection we investigate the optimal selection of clock gating boundaries for one and two points. In this subsection, we investigate the potential for clock gating at multiple points at finer granularities. For example, instead of clock gating just at 16 bits and 33 bits, as in our original proposal, another choice might be to clock gate four bitwidths: the 8-bit, 16-bit, 24-bit, and 33-bit boundaries.

Figures 10 and 11 show the percent of the integer unit power saved by clock gating at the specified granularities for the SPECint95 and multimedia benchmarks. In these figures, the last bar assumes clock gating at only the 33-bit boundary. The second to last bar is similar to our original proposal, in which we clock gate at 16 bits and 33 bits. The remaining three bars show the improvement by clock gating at additional, finer granularities. These figures show the diminishing marginal returns for clock gating as we approach 1 bit of granularity. The data suggests that our original proposal with two boundaries at 16 and 33 is close to optimal. If additional boundaries are desired, then 8-bit boundaries provide slightly better power savings.

## 4.5 Value-Based Clock Gating in Floating-Point Benchmarks

In this section we discuss value-based clock gating within floating-point benchmarks. Here we will consider clock gating on both integer data, as in the previous sections, and within certain types of floating-point operations.

4.5.1 *Clock Gating Integer Code in Floating-Point Benchmarks.* Floating-point benchmarks often contain a significant percentage of integer code in addition to floating-point operations. Integer code in floating-point benchmarks is often used for loop index variables and address computa-

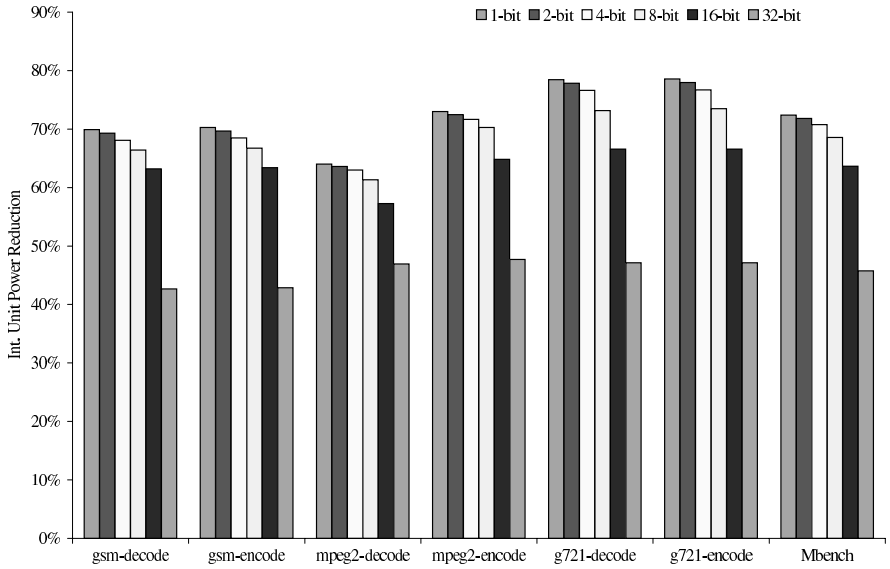


Fig. 11. Percent of integer unit power saved with varying clock gating granularities.

tions. In the integer benchmarks that we studied, roughly 50% of the instructions are integer computations that are available for clock gating. In the floating-point benchmarks approximately 25% of the instructions are integer computations. The integer computations within these benchmarks tend to have a larger percentage of arithmetic operations which consume more power than the other classes of instructions. Thus the power consumption within the integer unit is significant within these benchmarks.

Figure 12 presents the data for functional width analysis on the integer code within SPECfp95. This graph is similar to Figure 1 in which we present the data for SPECint95. The main difference between the two graphs is that the spike at 33 bits is larger, corresponding to the fact that address calculations will be a larger percentage of the integer code than within floating-point programs. Still, about 37% of the operations require 16 bits or less to perform their computation.

We next present data on the power saved by clock gating the floating-point benchmarks. We assume that we will definitely want to clock gate at 33 bits. Figure 13 shows that the optimal mark for placing the second clock gating mark is at the 11-bit mark. However, the difference between choosing the 11-bit mark and the 16-bit mark that we chose before is only 2%, so we can use 16 bits to keep the same hardware structure as the original proposal for the integer benchmarks.

Figure 14 shows the total power used by the integer assuming the baseline and clock gated configurations. The percentage savings of the clock gated configuration is still over 50%. However, as expected the total power used and saved within the integer unit is less than before. Hence the optimization would have less of an effect on the overall power dissipation of the processor for these floating-point programs.

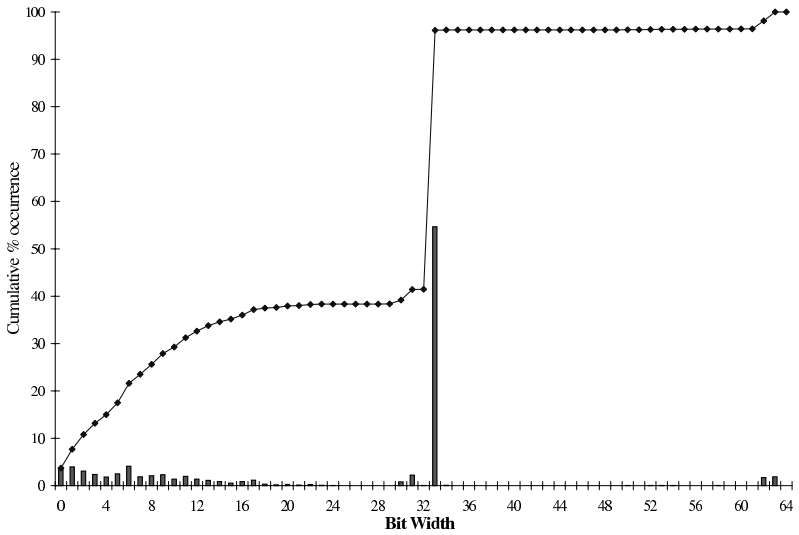


Fig. 12. Bitwidths for integer computation in SPECfp95 on 64-bit Alpha.

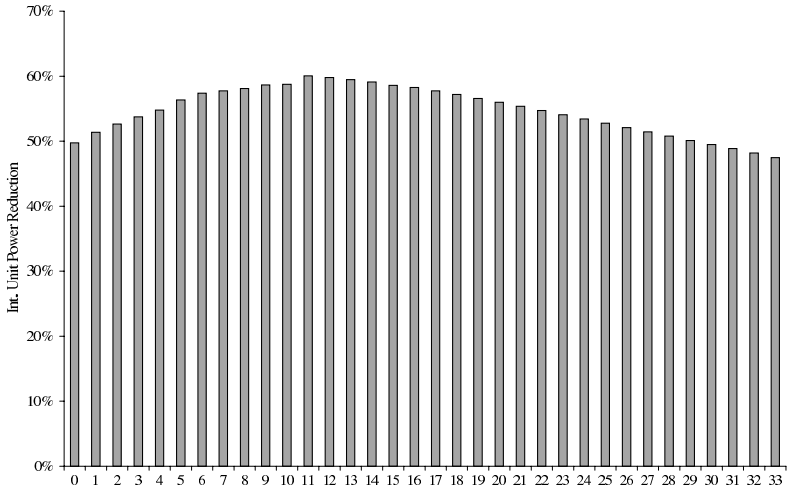


Fig. 13. SPECfp95 integer unit power reduction by clock gating at 33 bits as well as at the specified bitwidth.

4.5.2 *Clock Gating Floating-Point Operations.* Applications with floating-point code tend to have higher overall power dissipation. This is because floating-point operations are much more complex and hence use more power. For example, floating-point programs tend to have a larger number of power hungry multiplication operations. We will focus on these multiplication operations in this section for two reasons. First, in floating-point arithmetic, multiplication is simpler than addition and subtractions in that it does not require shifting an operand to align them before performing the computation. Essentially, the mantissas of the input operands are multiplied together and the exponents of the input operands are added together. Second, since multiplication is more expensive in terms of power dissipation there is more potential for power savings.

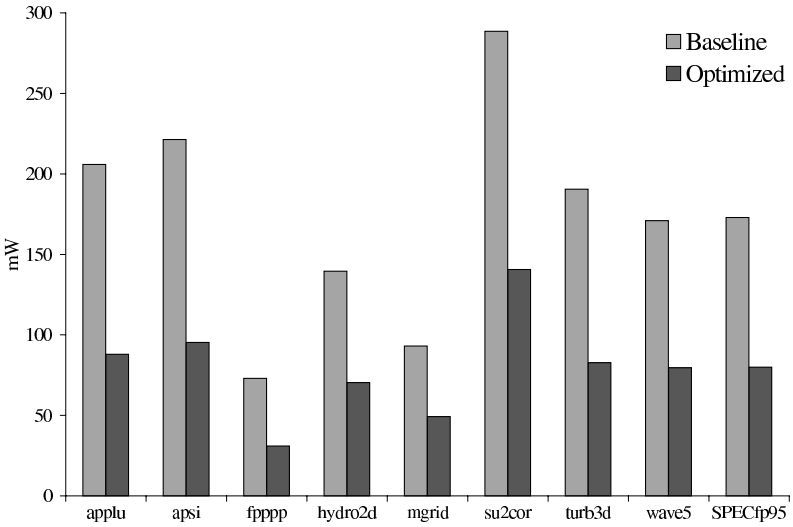


Fig. 14. SPECfp95 power usage of integer unit.

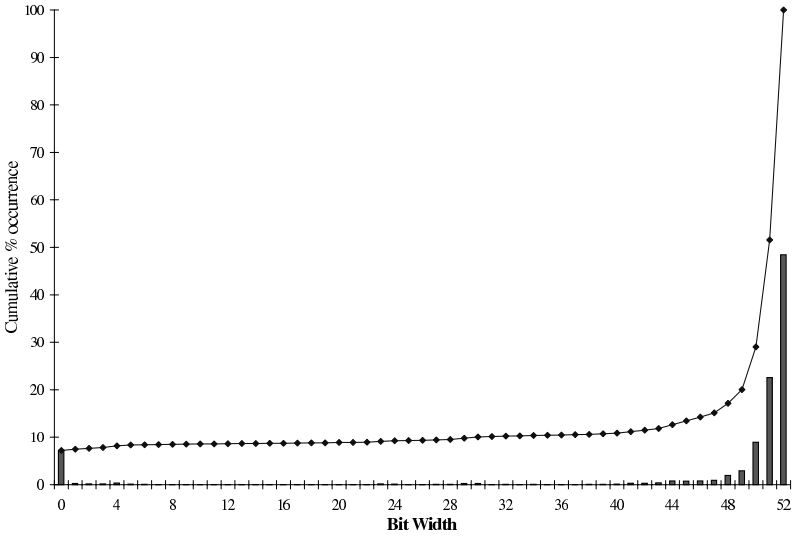


Fig. 15. Bit width analysis of the 52-bit mantissa in double-precision floating-point multiplication.

According to IEEE Standard 754, 64-bit double-precision, floating-point arithmetic uses a mantissa of 52 bits, an exponent of 11 bits, and one sign bit [IEEE Standards Board 1985]. We consider clock gating on input operands of the 52-bit integer multiplication operation that occurs in double-precision multiplication. In single-precision operations, the lower 29 bits are all zeros. Single-precision multiplication uses the same functional units as double-precision multiplication and would present many additional opportunities for clock gating. However, we do not consider them here because traditional opcode-based clock gating techniques would be sufficient to capture these situations.

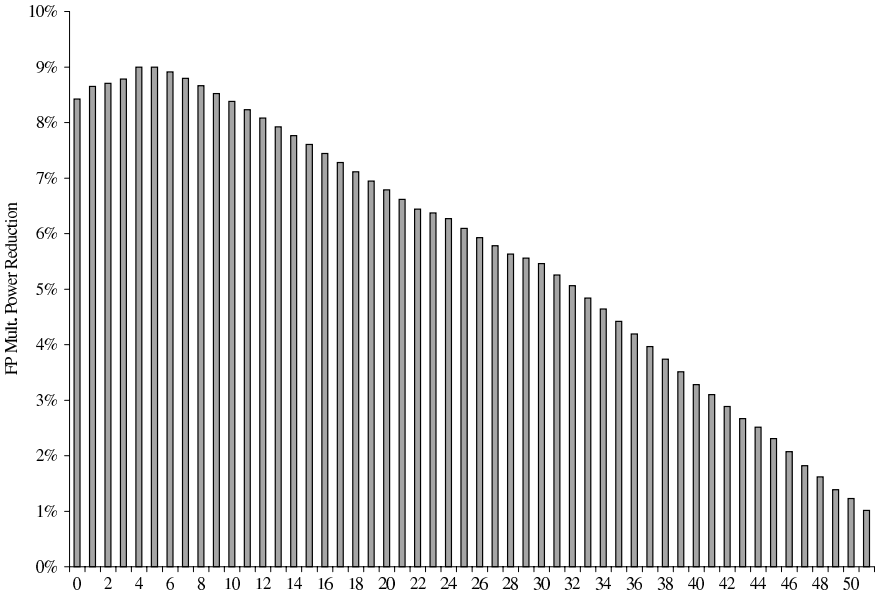


Fig. 16. FP multiplier power reduction by selecting one clock gating point in the 52-bit mantissa.

Figure 15 presents the bit width analysis for the 52-bit mantissa in double-precision floating-point multiplication. Most often the operations require nearly the full 52 bits of precision. However, roughly 10% of the operations require less than 4 bits of precision. Despite the small number of instructions that are amenable to clock gating, being able to clock gate nearly the full width of the multiplication saves an appreciable amount of power. Figure 16 shows the power saved by selecting one clock gating point within the 52-bit mantissa. By selecting gating at the 4-bit boundary, approximately 9% (18mW) can be saved. This compares to about 125mW saved by clock gating operations in the integer benchmarks, and about 100mW saved by clock gating integer operations in the floating-point benchmarks.

## 5. PERFORMANCE OPTIMIZATION: OPERATION PACKING

Section 4 discussed a power optimization based on the dynamic recognition of small input operands. In this section, we present a technique to increase performance by exploiting the same type of dynamic data. Both techniques rely on dynamically recognizing zeros in the upper bits of the input operands to take advantage of the unused upper bits in the functional units. Since the power optimization involves clock gating functional units and the performance optimization involves executing instructions in parallel, only one technique can be used at a time. However, because the techniques share a common hardware base, one could implement both and choose between them. For example, one could use thermal sensory data to have the processor switch between the two techniques, depending on current thermal or performance concerns. This effectively allows one to

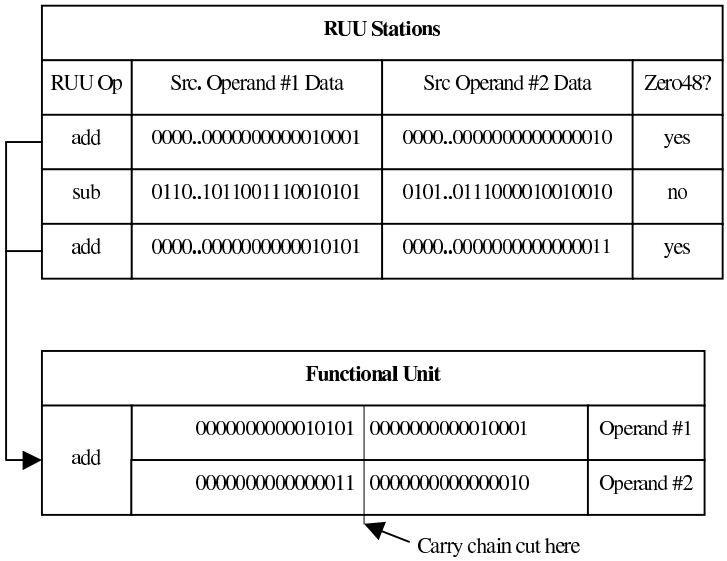


Fig. 17. Packing two add instructions with narrow operand widths.

trade performance for reduced power dissipation when necessary. Related but simpler approaches are already found in commercial processors; for example, the IBM/Motorola PPC750 is equipped with an on-chip thermal assist unit and an on-chip temperature sensor which responds to thermal emergencies by controlling the instruction fetch rate through I-cache throttling [Sanchez et al. 1997].

## 5.1 Background

Multimedia instruction sets define new instructions to perform a common operation on several subwords in parallel. For example, the Parallel Add instruction in HP-MAX performs four parallel additions on the 16-bit subwords that reside in the two specified 64-bit source registers. Few hardware changes are necessary to support these additional instructions; only the carry chain between the 16-bit chunks must be handled differently. Figure 17 demonstrates how two *add* instructions in the RUU, both with narrow operand widths, can be packed together at issue time into one functional unit. In this example, there are three instructions in the RUU: an *add* with source operand values of 17 and 2, a *sub* with source operands that are larger than 16 bits, and another *add* with source operands of 21 and 3. In this case, the two *add* instructions both have narrow width operands, so a single 64-bit adder can perform the two additions in parallel. The hardware built into the ALUs for the multimedia instruction sets will automatically stop the carry at 16-bit boundaries.

In machines with multimedia extensions, programmers or (less frequently) compilers statically generate code using multimedia instructions. As previously discussed, there are several shortcomings to this method. For those reasons, this section introduces an approach that is akin to dynamically generating multimedia instructions. In this study, we focus on merg-

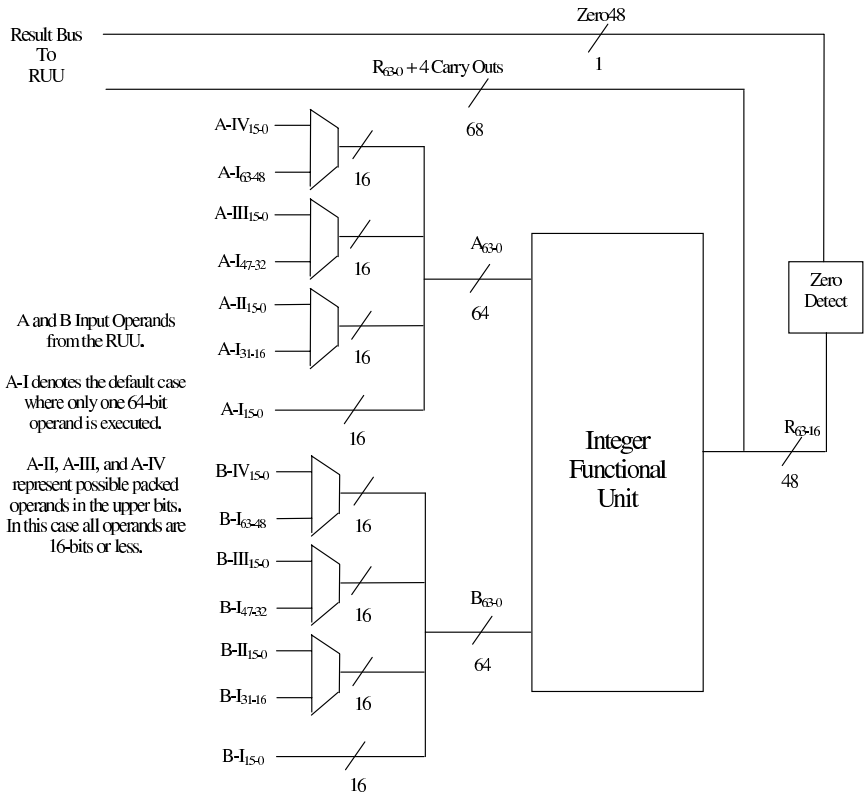


Fig. 18. Datapath modifications.

ing narrow integer operations into parallel subword operations as currently supported by multimedia instruction set extensions. This is a subset of the operations that we explore in Section 4 and consists of the arithmetic, logical, and shift operations. For example, we do not attempt to pack multiply operations, although in some implementations this would be possible.

## 5.2 Proposed Architecture

Figure 18 is a diagram of the proposed changes to the datapath. The most notable changes to the datapath are the additional muxes which move data from the low 16 bits of the source RUU stations onto the higher 16-bit paths of the source operand bus. In addition, 4 extra lines are needed on the result bus for the carry-out that could result when two 16-bit operands are added. These additional carry-out lines are needed because most multimedia instruction sets provide a form of saturating arithmetic which, upon overflow of two 16-bit values, sets the result to the maximum 16-bit value, namely 0xFFFF. Figure 19 details the internal features of the reservation stations and necessary modifications. The additional hardware needed here includes muxes which reverse the effect of the above; data in the higher 16-bit subwords of the result bus are muxed into the low 16-bit boundaries to be written back to the result reservation station. It should be

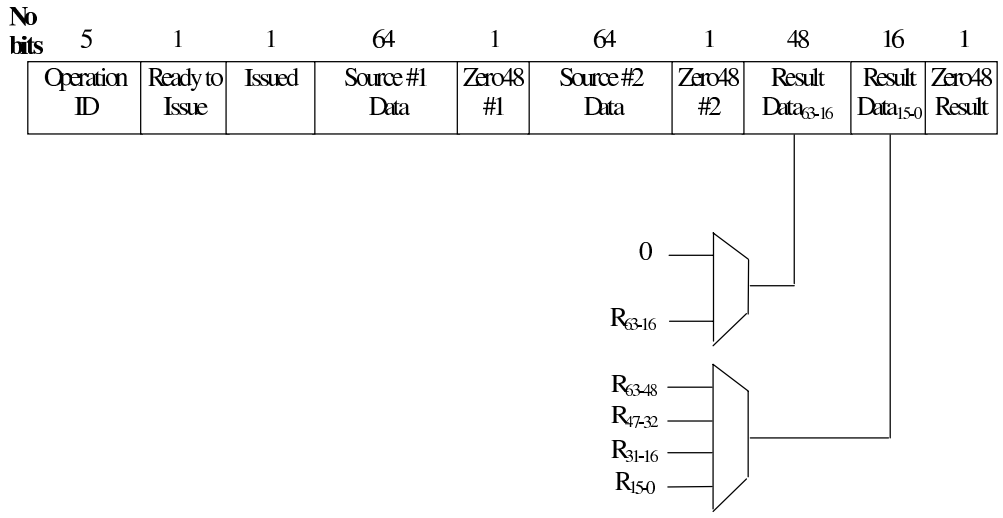


Fig. 19. Reservation station modifications.

noted that much of this "additional" muxing hardware may already exist in processors with multimedia instruction sets as part of the multimedia extensions or the standard forwarding and bypassing logic. For example, the HP MAX-2 instruction set includes instructions to select any field in a source register and right-align it in the target register [Lee 1996]. Instructions also exist to select a right-aligned field from the source register and place it anywhere in the target register. In Figure 18, additional muxes in front of the functional units are used to stage 16-bit subwords into the upper subwords of the functional unit. In most processors, there are already muxes in a similar position which select whether the input source operands come from the reservation stations or from the forwarding path. These bypassing muxes could be widened by one source input to order to support our proposal.

Our core idea is similar to the power optimization discussed in the last section. Each entry in the reservation update unit (RUU) stores an extra bit for each operand indicating that the size of the operand is 16 bits or less. These fields are updated when operands are computed and stored in the RUU buffers. Using these fields, the issue logic can recognize opportunities to pack narrow width operations together to share one integer ALU in the same way that the multimedia instructions do. In order for two operations to be packed, three things must occur. First, both instructions must have satisfied their data dependencies and be ready to issue. Second, both instructions must have narrow width operands. Finally, they must perform the same operation.

The issue logic issues ready instructions from the RUU using its normal algorithm. In most processors, this algorithm issues the oldest ready-to-issue instructions in the RUU. However, when both operands are 16 bits or less, an opportunity for packing exists. The issue logic must keep track of which issuing instructions are available for packing. If other instructions that perform the same operations are available to issue and have narrow-

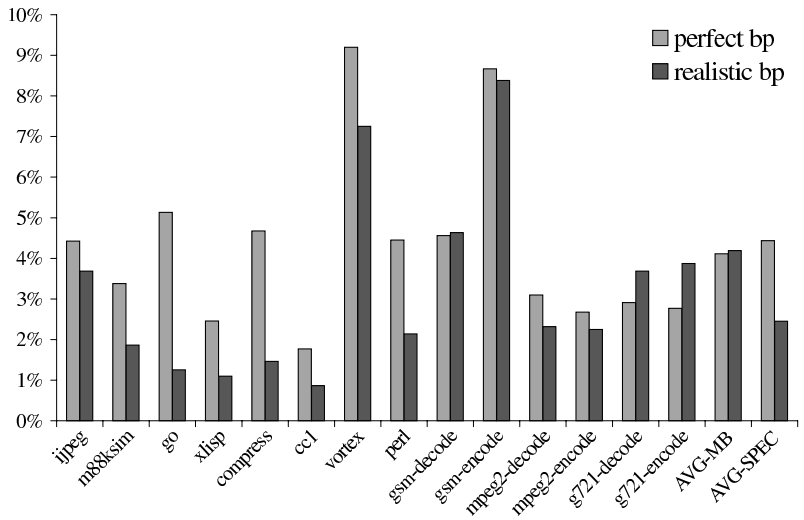


Fig. 20. Speedup due to operation packing.

width operands, these instructions can be packed. The issue logic sets the appropriate muxes to issue the packed instructions in parallel.

After the instructions issue, they execute in the same fashion as packed instructions do in the multimedia instruction sets. When execution completes, the result operands share the result bus and are sent back to their respective RUU station as well as to RUU stations awaiting the results as input operands. This optimization opens up machine issue bandwidth and integer ALUs available for certain integer executions. Much of the required multiplexing hardware already exists within processors designed with multimedia instruction sets. These processors also have functional units that are designed to disable the carry chain at 16-bit intervals. The primary hardware cost for this optimization is in the increased complexity of the logic that decides when packed instructions can issue. Handling negative numbers adds additional complexity to the issue logic, but Section 5.4 discusses methods to simplify the implementation.

### 5.3 Operation Packing Results

In this section we present speedup results for operand packing. We have considered two configurations: the first configuration is exactly the same as the baseline configuration discussed in Section 3. The second configuration increases the decode bandwidth from four instructions per cycle to eight instructions per cycle. The increased decode bandwidth causes the RUU to fill up faster giving more opportunities for packing.

Figure 20 shows the percent speedup over the baseline system with a decode width of four. In this chart, we include results for both perfect and realistic branch predictors. For the SPECint95 benchmarks, moving from perfect to realistic branch prediction shows a performance degradation, because it reduces the number of useful instructions that are ready to issue each cycle. We can see that *go*, notorious for its poor branch prediction, is

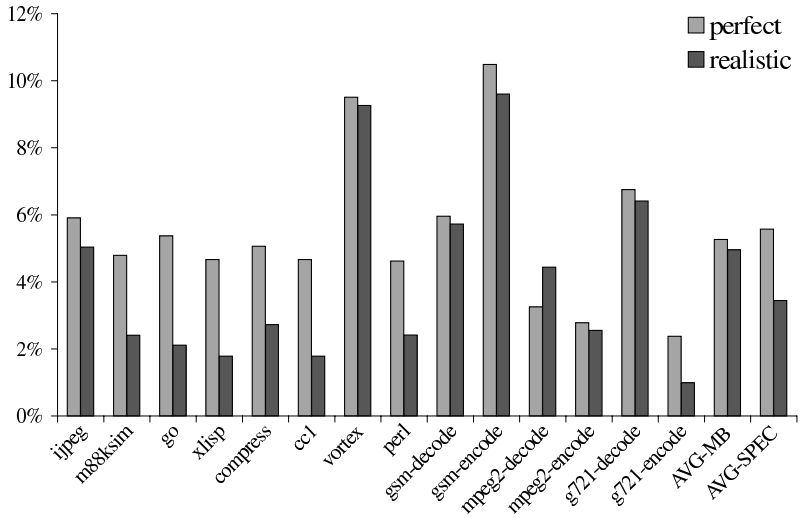


Fig. 21. Speedup due to operation packing with decode bandwidth = 8.

affected the most. *Ijpeg* and *vortex*, on the other hand, see little difference in the speedup between perfect and the realistic predictor. The average speedup across SPECint95 was 4.4% for perfect branch prediction and 2.5% with the realistic predictor. As one might expect, the multimedia benchmarks performed better than SPECint95. Because the multimedia benchmarks had very low mispredict rates, the perfect branch prediction led to only a small difference in speedup between perfect and realistic predictors. In fact, *g721* had higher speedup with realistic branch prediction, due to second order effects related to speculative execution. Speculative instructions that will eventually be squashed still get executed until the branch is resolved. Packing them with other instructions may reduce their use of functional unit resources so that when a key branch is ready to issue, it can. This allows some mispredicted branches to resolve faster. The average speedup for the media benchmarks was 4.1% with perfect branch prediction and 4.2% with the combining predictor.

We have also studied the packing optimization with a decode width of eight. These results are shown in Figure 21. As expected, the optimization performs better with increased decode bandwidth, because the RUU is filled with more useful instructions which have the potential to be packed, issued, and executed in parallel. However, the additional speedup that our method provides was only 1–2% for most of the benchmarks. The average speedup for SPECint95 was 5.6% for perfect branch prediction and 3.4% with the combining predictor. The multimedia benchmarks performed better as well, but not as significantly as SPECint95. The reason for this is that the multimedia applications have a large number of loop-oriented arithmetic operations with few data dependencies. This gives them a larger pool of usable instructions even with the smaller decode bandwidth. The average speedup for the media benchmarks was 5.3% with perfect branch prediction and 5.0% with the combining predictor.

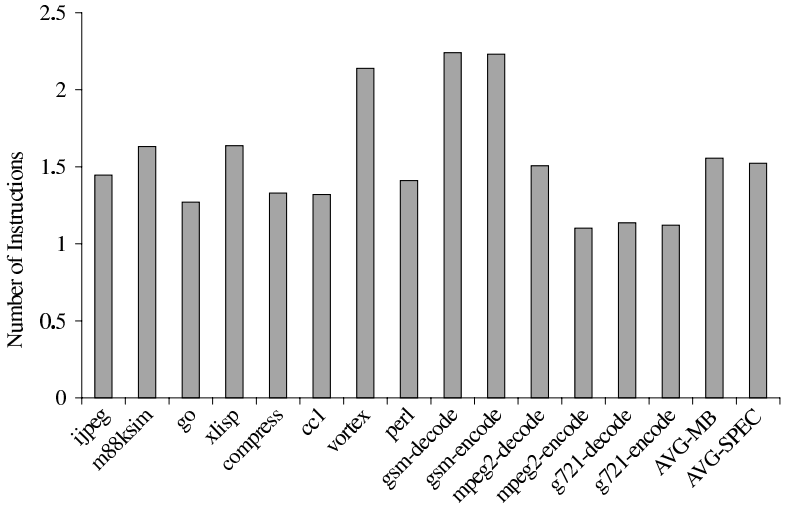


Fig. 22. Average number of instructions packed per packing opportunity.

While some benchmarks, such as *vortex* and *gsm-encode*, achieved impressive speedups of 7–10%, others only benefited by very small amounts. Part of the reason for the small speedups is due to the fact that there are not enough instructions available to be packed per cycle. In Section 6, we present a more aggressive approach for packing instructions that takes advantage of cases where a large number is added to a small number. These cases allow a more speculative approach that we call replay packing, which leads to significantly better speedups.

### 5.4 Hardware Requirements: Discussion

In Section 5.2 we have considered a completely general architecture that can pack up to four narrow-width instruction to be executed in parallel. In addition, these instructions can be selected from anywhere within the 80-entry RUU. There are two main sources of additional complexity with this scheme. First, packing four instructions requires the hardware to route 16-bit subwords from the low 16 bits of the original instruction source operands to the upper three subwords of the multimedia source operands. Second, the issue selection logic becomes more complicated since it must detect when opportunities for packing exist. In many pipelined processors, the execute cycle is often not on the critical path of the processor so it is possible to perform some additional multiplexing without impacting the cycle time. However, in dynamically scheduled processors, the issue logic can be complex; adding too much additional complexity may not be feasible without extending the processor cycle time. In this section, we will consider two possible hardware simplifications to the packing optimization.

One approach that simplifies both the additional muxing hardware and the issue logic is to limit the number of packed instructions to be less than four. Figure 22 shows when the packing optimization applies, the average number of instructions packed is much less than four. In fact the average

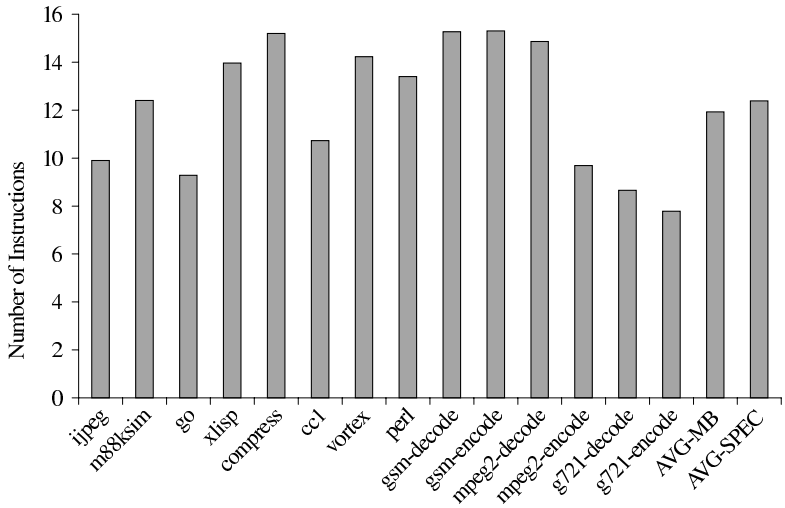


Fig. 23. Average distance (in instructions) between packed instructions.

for both SPECint95 and the multimedia benchmarks is around 1.5. From this data, it seems that by limiting the number of packed instructions to 2 or 3, similar performance benefits could be realized with simpler hardware.

The second proposed simplification limits the scope of instructions to search for packing opportunities. Figure 23 shows the average distance between packed instructions within the 80-entry RUU. The average distance for both sets of benchmarks was approximately 12 instructions. Limiting the distance between packed instructions to 16 instruction blocks would simplify the selection logic during the issue stage. This method would fit in well with the bank selection approach that is used in HP's instruction reorder buffer and MIPS' instruction queues [Gaddis et al. 1996; Vasseghi et al. 1996].

In dynamically scheduled processors that do not have a centralized window of instructions, however, the implementation of the selection logic would be even simpler. For example, the PowerPC architecture has a decentralized instruction queue in which the reservation stations directly precede the functional units [Diep et al. 1995]. In this case, the instructions are already sorted by the type of functional unit that they require. This would simplify the logic that needs to pack instructions of the same type. Furthermore, this decentralized approach generally has fewer instructions to select from, because all of the reservation stations are distributed in front of the functional units.

Another ramification of effectively increasing the issue/execute bandwidth of the machine is that it may cause other parts of the machine to become a bottleneck. For example, by packing operands it could be possible that the writeback/commit stages could be saturated. While it would be possible to share result bus bandwidth with this scheme, increasing the number of ports on the wakeup/dependency check logic can be costly. However, in many cases the amount of instructions that writeback per cycle is smaller than the amount that fetch, decode, and issue/execute per

cycle due to pipeline hazards despite the fact that machines are usually designed to writeback the same number of instructions per cycle that they fetch and execute [Hennessy and Patterson 1996]. Thus, it is unlikely that increasing pressure on the writeback/commit bandwidth would be a serious problem for most benchmarks.

## 6. SPECULATIVE APPROACHES FOR EXPLOITING NARROW-WIDTH OPERANDS

Both the power optimization discussed in Section 4 and the performance optimization discussed in Section 5 require that both input source operands be less than 16 bits to operate most efficiently. For the power optimization, if the first input operand is less than 16 bits and the second operand is greater than 16 bits, yet still less than 33 bits, it will be clock gated at the 33-bit mark rather than the more optimal 16-bit mark. For the performance optimization, the instruction will be excluded from packing entirely if one operand is larger than 16 bits.

The requirement that both input operands be less than 16 bits excludes a large number of arithmetic operations used for memory addressing, loop incrementing, etc. In many of these cases, one of the input operands may be very large, while the other is quite small. When this is true, it is possible that adding them will result in a carry that ripples into the highest bits, but in practice, such large ripple carries occur infrequently. Based on this observation, we present extensions here to both the power and performance optimizations that allow the optimization to proceed speculatively assuming that there will be no overflow from the 16-bit operation; the high 48 bits of the larger source operand can be muxed onto the result bus to proceed into the destination RUU stations. However, in the rare cases that there is overflow from the 16-bit addition, the instruction can be squashed and subsequently reexecuted as a full-width instruction. Such a situation could be handled in a similar manner to "replay traps," which are already available for other reasons in the Alpha 21164 and other CPUs [Bowhill et al. 1995].

### 6.1 Replay Clock Gating for Arithmetic Operations with Varying Operand Sizes

In this section we investigate the benefits of speculatively clock gating operations at the 16-bit mark when one source operand is less than 16 bits and the other source operand is greater than 16 bits. We will call this technique *replay clock gating*.

When the replay clock gate operation succeeds, the power savings are similar to those previously presented. We must also, however, account for the cases when the 16-bit addition has carry-out and the instruction must be reexecuted. These *replay overflows* incur both a performance and a power penalty. Because of this, we would like the percentage of instructions that overflow the 16-bit boundary to be low. Figure 24 demonstrates that for most of the benchmarks this is true. This figure shows the percentage of

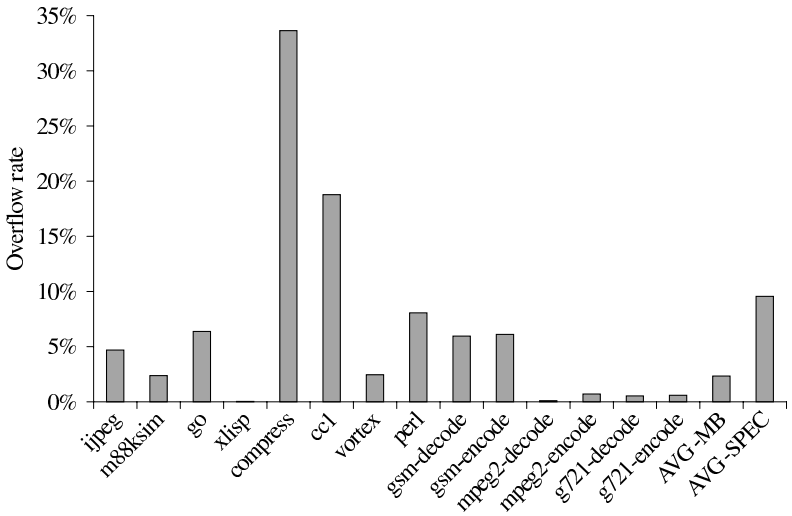


Fig. 24. Percentage of replay clock gated instructions that overflow the 16-bit boundary.

replay clock-gated operations that overflowed the 16-bit boundary. For the SPECint95 benchmark suite, about 9% of the speculatively clock gated instructions did have overflow. The multimedia benchmarks, having more regular data types and ranges, had a overflow rates of only 2%. *Compress* (33%) and *cc1* (18%) had the highest overflow rates.

In computing the net power saved via replay clock gating, we attempted to charge operations with a power cost when they overflow and need to be reexecuted. We also took into account the power cost of reissuing instructions in the previous pipeline stage that were dependent on the squashed instruction. Computing the amount of power used when reexecuting is fairly straightforward; we simply charge the instruction with the cost of a second add (usually 33 bits, assuming 33-bit clock gating was valid). Estimating the amount of additional power consumed to reissue the dependent instruction is more difficult and depends heavily on the actual implementation details of the processor. Palacharla, et al. study the complexity of out-of-order processors. This work investigated the delay of the instruction window wakeup and selection logic using parameterizable delay models based on low-level capacitance estimates of the nodes through the critical path of the circuits [Palacharla et al. 1997]. In a similar fashion, we have developed detailed power models for the out-of-order issue logic. The main differences between our work and that of Palacharla is that we have to compute capacitance estimates for *all* paths through the circuits instead of just the critical path. After computing the capacitance, we arrive at dynamic power estimates by using $P_d = CV_{dd}^2 af$. Using these models we are able to estimate the power dissipation of reissuing the dependent instructions. More details of this power modeling methodology can be found in Brooks et al. [2000].

Figure 25 shows the net savings with and without replay clock gating. The net savings with replay includes the amount of additional power saved on replay clock gated instructions as well as an estimate for the amount of

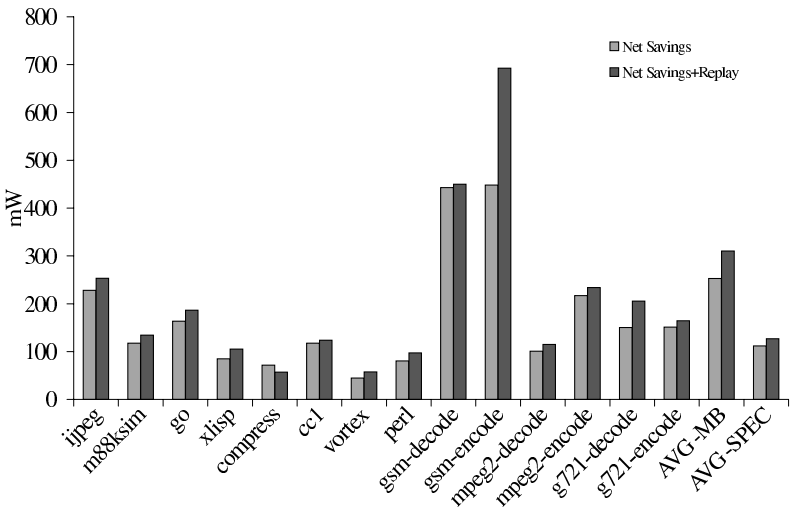


Fig. 25. Net savings with and without replay clock gating.

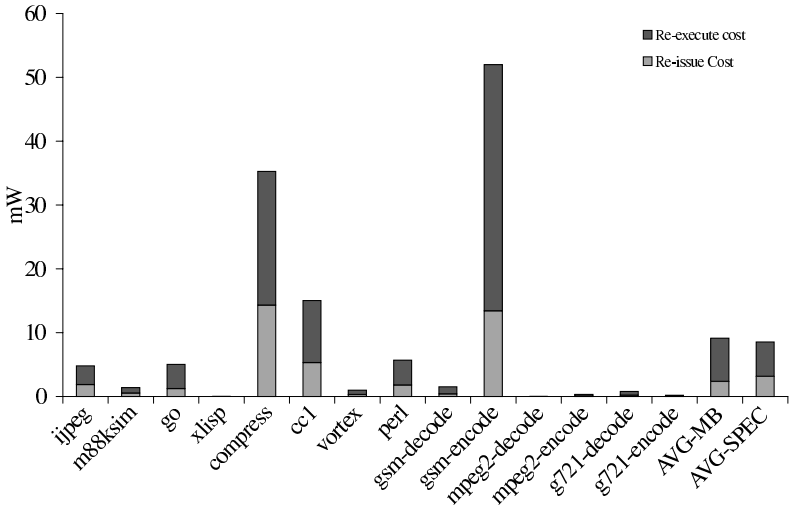


Fig. 26. Additional power used when replay overflow occurs.

extra power dissipated due to replay overflows. The amount of additional power saved was approximately 12% for SPECint95 and 21% for the multimedia benchmarks. However, as expected the benchmarks did not perform uniformly. In fact, the net savings for *compress* was 20% lower when using replay clock gating; its unusually large number of replay overflows incur additional power consumption. Figure 26 shows the two components of the additional power used when replay overflow occurs. The power used to reexecute instruction is about 2–3 times higher.

In addition to consuming additional power, reissuing and executing instructions can lead to performance degradation. All of the benchmarks we considered, except *compress*, performed within 0.5% of the baseline system when using replay clock gating. *Compress* suffered a 4% performance degradation due to the large number of replay overflows.

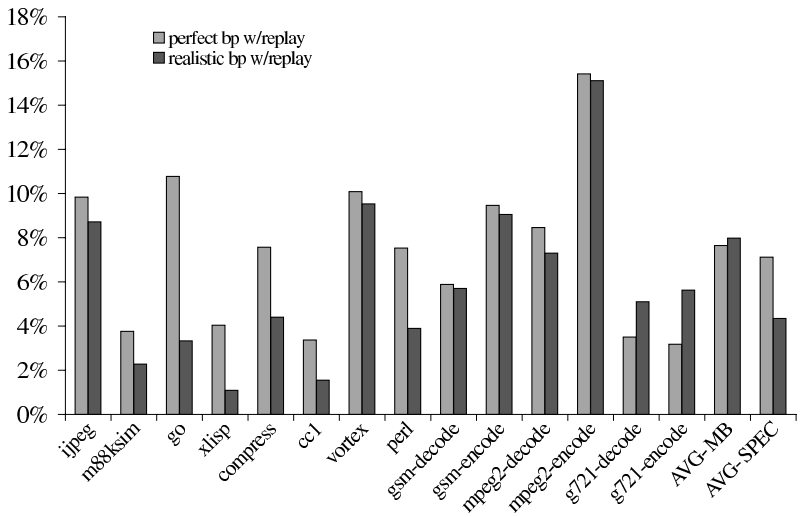


Fig. 27. Speedup due to operation packing with replay packing.

Overall, replay clock gating has mixed results. For most of the applications in the benchmark suite, the additional power savings benefits are attractive. However, for *compress* the performance degradation is noticeable.

### 6.2 Replay Packing for Arithmetic Operations with Varying Operand Sizes

While speculatively applying clock gating has mixed results, speculative approaches for operation packing are quite successful. We call this technique *replay packing*. Replay packing has a more significant impact than replay clock gating because in the method for packing presented in Section 5, operations with one operand greater than 16 bits are totally excluded. In contrast, for the power optimization discussed in Section 4, the clock gating optimization is often still applicable at the 33-bit clock gating boundary.

In this section we present the speedup results for the replay packing optimization. We consider the same two configurations discussed in Section 5.3. The performance simulator has been modified to perform replay packing and to reissue and execute instructions when replay overflow occurs.

Figure 27 shows the percent speedup that operation packing with replay packing provides over the baseline system with a decode width of four. The average speedup across SPECint95 was 7.1% for perfect branch prediction and 4.3% with the realistic predictor. Again, the multimedia benchmarks performed better than SPECint95. The average speedup for the media benchmarks was 7.6% with perfect branch prediction and 8.0% with the combining predictor. The replay packing technique provides nearly double the speedup over operation packing alone. *Compress* and *mpeg2-encode* were affected the most by the addition of replay packing. *Compress* went from having one of the worst speedups without replay packing (1.46%) to about average (4.40%) with replay packing. *Mpeg2-encode* benefited even more improving, from 2.25% to 15.11%.

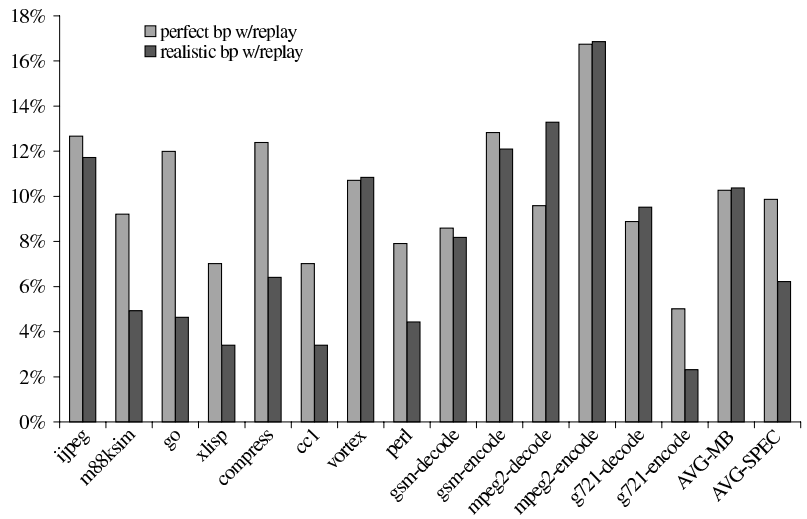


Fig. 28. Speedup due to operation packing with replay packing and decode = 8.

We have also studied the packing optimization with replay packing and a decode width of eight. The results are shown in Figure 28. The increased decode bandwidth had a larger effect on the speedups than without replay packing. The combination of increasing the pool of usable instructions and using replay packing to allow more packed instructions proved to be effective. Most of the benchmarks show a 2–3% increase in speedup with the increased decode bandwidth. The average speedup for SPECint95 was 9.9% for perfect branch prediction and 6.2% with the combining predictor. The average speedup for the media benchmarks was 10.3% with perfect branch prediction and 10.4% with the combining predictor.

The addition of replay packing to the operation packing optimization was very effective. As previously mentioned, the packing optimization increases the effective issue bandwidth and number of integer ALUs by packing several instructions and issuing and executing them in parallel. Thus, it is useful to compare our optimization to a machine that simply has more issue and execution bandwidth. Figure 29 compares the number of instructions per cycle (IPC) for three different configurations, all with combining branch prediction and decode width of four. The first is the baseline machine with issue width of 4 and 4 integer ALUs. The second is the baseline machine augmented with our operation packing optimizations (including replay packing). The third machine is the baseline machine with an issue width of 8 and 8 integer ALUs. Note in all configurations the decode and commit bandwidth has been held at four. We see that *ijpeg* and *vortex*, as well as many of the multimedia benchmarks, come very close to achieving the same IPC as the more costly 8-issue/8-ALU implementation.

### 6.3 Summary of Results

Overall, Sections 4, 5, and 6 have explored two optimizations that both exploit the detection of narrow-width operations at run-time. We discussed both speculative and nonspeculative versions of each optimization.

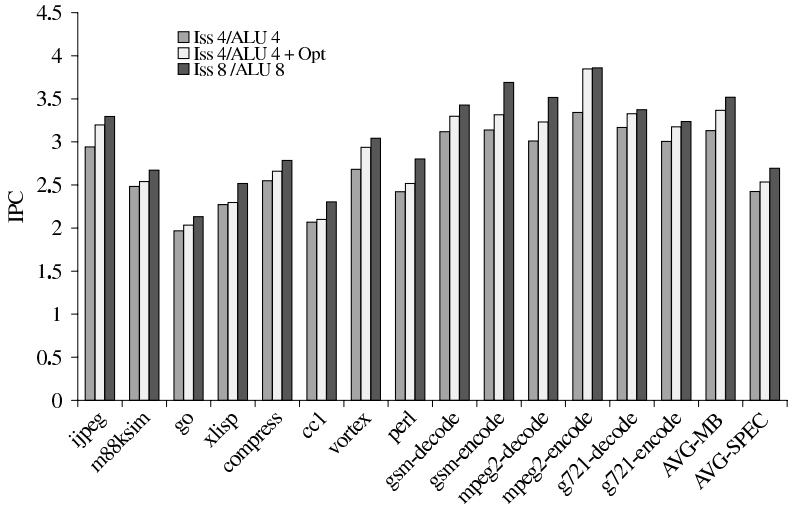


Fig. 29. IPC for the baseline system, the optimized system, and an 8-issue system.

For the power optimization, the nonspeculative version of the clock gating optimization seems like the best choice. While the speculative optimization saved approximately 20% more power, performance may be sacrificed for some applications, since instructions must be reissued after a misspeculation.

The speculative version of operation packing was more successful. Replay packing achieved speedups of 4.3%–6.2% for SPECint95 and 8.0%–10.4% for the multimedia benchmarks. This is a significant improvement over the nonspeculative version of the operation packing optimization.

Both of these techniques are most successful when the 16-bit overflow rate is low as shown in Figure 24. Overflow confidence predictors could be used to decrease the overflow rates by recording the 16-bit overflow history of arithmetic operations to determine whether it is expected to be useful to perform the replay gating/packing. This would decrease the replay overflow rate and hence the benefits of both techniques.

## 7. CONCLUSIONS AND FUTURE WORK

Recently there has been increased interest in supporting operations with operand widths smaller than the maximum supported by functional units in microprocessors. This interest stems first from the increasing use of multimedia applications, but also from the larger 64-bit word sizes on current microprocessors. Most of the past research in this area has focused on increasing performance by discerning instructions with narrow width operands at compile time and generating code that allows such computations to occur with subword parallelism.

Compile-time analysis of operand width is constrained by the fact that the operand range of instructions may vary over the course of a program run depending on the input data. In addition, the compiler must conservatively analyze all potential paths taken. Our work notes that certain

uncommon paths may have markedly different operand size characteristics than the typical path through programs.

In order to augment compile-time analysis, we present two techniques to *dynamically* exploit narrow-width data. The first method reduces power in the integer execution unit with aggressive clock gating, after determining that the upper portion of functional unit is not needed. This results in a 45%–60% reduction in the integer unit's power consumption for the benchmarks that we studied. This equates to a 5%–10% full-chip power savings. The second technique increases performance by dynamically recognizing opportunities to issue multiple narrow width instructions to the same functional unit to be executed in parallel. This technique provides performance speedups of 4.3%–10.4%.

The techniques described in this article both rely on the same core mechanism to achieve their optimization; namely they recognize that the upper bits in the input operands are not needed to perform the computation. Another area offering opportunities for dynamic recognition of low-precision operations is in the memory and I/O hierarchy. These opportunities include: (1) *pin and bandwidth compression* by recognizing that multiple pieces of low-precision data can share the same I/O pins and on-chip wiring, and (2) *low-power caches* which save power by writing 16 bits for the value and one signal bit indicating that the stored value is low precision rather than writing the full 64 bits.

A key characteristic of our current proposals is that they require only a small amount of hardware and no compiler intervention. Because of their common hardware requirements, we foresee systems in which the choice of whether to use the power or performance optimization can also be made dynamically, based on thermal input or other mode controls. More broadly, they represent a further step toward operand-value-based optimization strategies throughout processors.

## REFERENCES

ALIDINA, M., MONTEIRO, J., DEVADAS, S., GHOSH, A., AND PAPAEFTHYMIOU, M. 1994. Precomputation-based sequential logic optimization for low power. *IEEE Trans. Very Large Scale Integr. Syst. 2*, 4 (Dec. 1994), 426–436.

ALPERN, B., CARTER, L., AND SU GATLIN, K. 1995. Microparallelism and high-performance protein matching. In *Proceedings of the 1995 Conference on Supercomputing (CD-ROM)* (San Diego, CA, Dec. 3–8, 1995), S. Karin, Ed. ACM Press, New York, NY.

ASANOVIC, K., KINGSBURY, B., IRISSOU, B., BECK, J., AND WAWRZYNEK, J. 1996. TO: A single-chip vector micropocessor with reconfigurable pipelines. In *Proceedings of the 22nd European Solid-State Circuits Conference*,

AZAM, M., FRANZON, P., AND LIU, W. 1997. Low power data processing by elimination of redundant computations. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design* (ISLPED '97, Monterey, CA, Aug. 18–20), B. Barton, M. Pedram, A. Chandrakasan, and S. Kiaei, Eds. ACM Press, New York, NY, 259–264.

BHANDARKAR, D. P. 1996. *Alpha Implementations and Architecture: Complete Reference and Guide*. Digital Press, Newton, MA.

BORAH, M, OWENS, R. M., AND IRWIN, M. J. 1996. Transistor sizing for low power CMOS circuits. *IEEE Trans. Comput.-Aided Des. 15* (June 1996), 665–671.

BOWHILL, W. J., BELL, S. L., BENSCHNEIDER, B. J., BLACK, A. J., BRITTON, S. M., CASTELINO, R. W., DONCHIN, D. R., EDMONDSON, J. H., FAIR, H. R., GRONOWSKI, P. E., JAIN, A. K., KROESEN,

P. L., LAMERE, M. E., LOUGHLIN, B. J., MEHATA, S., SANTHANAM, S., SHEDD, T. A., THIERAUF, S. C., MUELLER, R. O., PRESTON, R. P., AND SMITH, M. J. 1995. Circuit implementation of a 300-MHz 64-bit second-generation CMOS Alpha CPU. *Digital Tech. J. 7*, 1 (Jan. 1995), 100–118.

BROOKS, D. AND MARTONOSI, M. 1999. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Conference/Symposium on High-Performance Computer Architecture* (HPCA-5, Jan. 1999),

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture* (ISCA-27, June 2000), ACM, New York, NY.

CALLAWAY, T. AND SWARTZLANDER, E. E. 1997. Power-delay characteristics of CMOS multipliers. In *Proceedings of the 13th International Symposium on Computer Arithmetic* (July 1997), ACM, New York, NY.

DIEP, T. A., NELSON, C., AND SHEN, J. P. 1995. Performance evaluation of the PowerPC 620 micro-architecture. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (ISCA '95, Santa Margherita Ligure, Italy, June 22–24), D. A. Patterson, Ed. ACM Press, New York, NY, 163–174.

DULONG, C. 1998. The IA-64 architecture at work. *IEEE Computer 31*, 7 (July), 24–32.

GADDIS, N., BUTLER, J. R., KUMAR, A., AND QUEEN, W. J. 1996. A 56-entry instruction reorder buffer. In *Proceedings of the 1996 International Solid State Circuits Conference Digest on Technical Papers*,

GEROSA, G. 1994. A 2.2W, 80 MHz superscalar RISC microprocessor. *IEEE J. Solid-State Circuits 29*, 12, 1440–1454.

GONZALEZ, R. AND HOROWITZ, M. 1996. Energy dissipation in general purpose microprocessors. *IEEE J. Solid-State Circuits 31*, 9 (Sept.), 1277–1284.

GOWAN, M., BIRO, L., AND JACKSON, D. 1998. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*. 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA.

HUNT, D. 1995. Advanced performance features of the 64-bit PA-8000. In *Proceedings of Compcon* (San Francisco, CA, Mar.), 123–128.

IEEE STANDARDS BOARD. 1985. IEEE Standards for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985. IEEE Standards Office, New York, NY.

KOJIMA, H., GORNY, D., NITTA, K., SHRIDHAR, A., AND SASAKI, K. 1996. Power analysis of a programmable DSP for architecture and program optimization. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E79-C*, 12, 1686–1692.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 30, Research Triangle Park, NC, Dec. 1–3), M. Smotherman and T. Conte, Eds. IEEE Computer Society Press, Los Alamitos, CA, 330–335.

LEE, R. 1996. Subword parallelism with MAX-2. *IEEE Micro 16*, 4, 51–59.

LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. 1996. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VII, Cambridge, MA, Oct. 1–5, 1996), B. Dally and S. Eggers, Eds. ACM Press, New York, NY, 138–147.

NAGENDRA, C., IRWIN, M., AND OWENS, R. 1996. Area-time-power tradeoffs in parallel adders. *IEEE Trans. Circ. Syst. 43*, 10, 689–702.

NG, P., BALSARA, P., AND STEISS, D. 1996. Performance of CMOS differential circuits. *IEEE J. Solid-State Circuits 31*, 6, 841–846.

PALACHARLA, S., JOUPPI, N. P., AND SMITH, J. E. 1997. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture* (ISCA '97, Denver, CO, June 2–4), A. R. Pleszkun and T. Mudge, Eds. ACM Press, New York, NY, 206–218.

PELEG, A. AND WEISER, U. 1996. MMX technology extension to the Intel architecture. *IEEE Micro 16*, 4, 42–50.

POPESCU, V., SCHULTZ, M., SPRACKLEN, J., GIBSON, G., LIGHTNER, B., AND ISAMAN, D. 1991. The Metaflow architecture. *IEEE Micro 11*, 3, 10–13.

RAZDAN, R. AND SMITH, M. D. 1994. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (MICRO 27, San Jose, CA, Nov. 30–Dec. 2), H. Mulder and M. Farrens, Eds. ACM Press, New York, NY, 172–180.

RICHARDSON, S. E. 1992. Caching function results: Faster arithmetic by avoiding unnecessary computation. Tech. Rep. TR-92-1 (Sept). Sun Microsystems Laboratories.

SANCHEZ, H., KUTTANNA, B., OLSON, T., ALEXANDER, M., GEROSA, G., PHILIP, R., AND ALVAREZ, J. 1997. Thermal management system for high performance PowerPC microproessors. In *Proceedings of COMPCON*,

SKADRON, K., AHUJA, P. S., MARTONOSI, M., AND CLARK, D. W. 1999. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Trans. Comput. 48*, 11 (Nov.).

SOHI, G. S. AND VAJAPEYAM, S. 1987. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture* (ISCA '87, Pittsburgh, PA, June 2–5), D. St. Clair, Ed. ACM Press, New York, NY, 27–34.

TIWARI, V., MALIK, S., AND ASHAR, P. 1998a. Guarded evaluation: Pushing power management to logic synthesis/design. *IEEE Trans. Comput.-Aided Des. Integr. Circuits 17*, 10, 1051–1060.

TIWARI, V., SINGH, D., RAJGOPAL, S., MEHTA, G., PATEL, R., AND BAEZ, F. 1998b. Reducing power in high-performance microprocessors. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Eds, ACM Press, New York, NY, 732–737.

TONG, Y., RUTENBAR, R., AND NAGLE, D. 1998. Minimizing floating-point power dissipation via bit-width reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (ISCA '98, Barcelona, Spain, June 27–July 1), M. Valero, G. S. Sohi, and D. DeGroot, Eds. IEEE Press, Piscataway, NJ.

TREMBLAY, M., O'CONNOR, J., NARAYANAN, V., AND HE, L. 1996. The visual instruction set (VIS) in UltraSPARC. *IEEE Micro 16*, 4, 10–20.

VASSEGHI, N. 1996. 200MHz superscalar RISC processor circuit design issues. In *Proceedings of the 1996 International Solid State Circuits Conference Digest on Technical Papers*.

ZIMMERMANN, R. AND FICHTNER, W. 1997. Low-power logic styles: CMOS versus pass-transistor logic. *IEEE J. Solid-State Circuits 32*, 7, 1079–1090.

Received: February 1999; revised: December 1999; accepted: February 2000