Software-Controlled Operand-Gating

Ramon Canal[1], Antonio González[1,2], James E. Smith[3]

[1]*Departament d´Arquitectura de Computadors - Universitat Politècnica de Catalunya*
{rcanal,antonio}@ac.upc.es

[2]*Intel Barcelona Research Center Intel Labs – Universitat Politècnica de Catalunya*
antoniox.gonzalez@intel.com

[3]*Department of Electrical and Computing Engineering University of Wisconsin-Madison*
jes@ece.wisc.edu

Abstract

Operand gating is a technique for improving processor energy efficiency by gating off sections of the data path that are unneeded by short-precision (narrow) operands. A method for implementing software-controlled power gating is proposed and evaluated. The instruction set architecture (ISA) is enhanced to include opcodes that specify operand widths (if not already included in the ISA). A compiler or a binary translator uses statically available information to determine initial value ranges. The technique is enhanced through a profile-based analysis that results in the specialization of certain code regions for a given value range. After the analysis, instruction opcodes are assigned using the minimum required width. To evaluate this technique the Alpha instruction set is enhanced to include opcodes for 8, 16, and 32 bit operands. Applying the proposed software technique to the SpecInt95 benchmarks results in energy-delay^2 savings of 14%. When combined with previously proposed hardware-based techniques, the energy-delay^2 benefit is 28%.

1. Introduction

Power consumption has become a critical design consideration and is expected to be one of the most important constraints for development of future microprocessors. In current CMOS technology, most energy consumption occurs when transistors switch or when a memory access takes place [6]. In general, dynamic energy consumption is proportional to the switching activity. Thus, an important energy conservation technique is to reduce switching activity by "gating off" portions of logic and memory that are not required for correct processing.

In recent research [2][12], it is proposed that certain portions of functional units can be gated-off for short precision operands. In particular, arithmetic only needs to be performed on the numerically *significant* bits of operands. Portions of arithmetic units operating on the *insignificant* bits (typically leading ones or zeros) can be gated off. In [9] this technique, *operand gating*, is extended to all stages of the processor pipeline, including buses, register files, and caches.

In this paper, we propose and study two software-based approaches to operand gating, *Value Range Propagation* and *Value Range Specialization,* that exhibit a different set of hardware/software tradeoffs. It is assumed that the instruction set architecture (ISA) contains opcodes that specify operand lengths (e.g. *load byte, add halfword*). This feature is already available in some conventional instruction sets and could be added as an extension to others. At compile time, or as part of static binary translation, an enhanced version of *value range propagation* is used to determine bounds on the *useful* value ranges of all variables. Then, through proper opcode assignment, only useful portions of data need to be computed, communicated, and stored, thereby saving power. This approach adds much less hardware complexity than the previously proposed significance compression methods, but requires static analysis by the compiler or translator and may require additional instruction opcodes to specify operand widths (depending on the base ISA).

In this work, a key distinction is made between "useful" data widths and "conventional" data widths. In the proposed software method, higher-level analysis of the code can determine the number of bits that are actually useful for determining the final program results. In some cases this means that widths of intermediate values are reduced by eliminating bits that strictly speaking are significant, but which are unnecessary. For example, if the last instruction of a chain of dependences is AND R1, 0xFF, R2 (i.e. R2←R1&0xFF), and the value of R1 is not used anywhere else in the program, one can conclude that only the least significant byte of R1 is needed, and thus the chain of dependent instructions leading up to the AND need to compute just one byte. By focusing on useful bits, the software approach can potentially save more energy than traditional value range propagation mechanisms.

Value Range Propagation (VRP) as just proposed is similar in essence to previous proposals [5][13][18][20]. Our extensions include provisions for "useful" value propagation. The useful value range concept was exploited in [13] through the use of directives written in the program rather than being derived from the program analysis as it is done in this work. Budiu et al. [5] used "useful" information on a per-bit basis, not a data word basis. Other extensions proposed in this paper include accounting for

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

wrap-around behavior in the case of overflow and the analysis of all the program code (including library code). At the same time, the technique implemented works at the binary level, not with a HLL representation as the previously cited work did. To the best of our knowledge, profiling techniques to enhance value range propagation have not been studied so far. Thus, another novelty of this work is the proposal of the Value Range Specialization (VRS) technique. VRS is a profile-based technique similar in concept to Value Specialization [7][8] that leverages profiling analysis to estimate the run-time value range of a given set of candidate operands. After a cost-benefit analysis, some of these candidates may be specialized for a given range followed by a value range propagation step. This may result in range reduction for certain operands in some portions of the code. Value Range Specialization differs from previous proposals because (1) it uses energy-based heuristics to decide which instructions to specialize, and (2) it specializes on value ranges (not just single values). A more detailed analysis of related work is in Section 5.

This paper is organized as follows. Section 2 describes the proposed Value Range Propagation technique. Section 3 describes the profiling technique, namely Value Range Speculation. The performance evaluation and the hardware implementation issues are presented in Section 4. The related work is discussed in Section 5. Finally, conclusions are presented in Section 6.

2. Value Range Propagation

A compiler or binary translator implements the proposed VRP technique. It first finds individual instructions where value ranges can be immediately determined. It then propagates this information to other instructions and estimates a conservative value range for each integer register. Methods are given for propagating value ranges within loops and across procedure boundaries. Value range information can then be used to determine the number of bits that must be computed and stored in order to maintain the semantics of the original HLL program. Finally, opcodes are assigned to specify the needed value ranges.

In our study, based on a 64-bit architecture, we assume that opcodes may specify operand widths of a byte, halfword, word, and doubleword. Many conventional ISAs already support many of the needed opcodes; otherwise opcode sets will need to be enhanced.

VRP is always done in a conservative manner (and thus requires no hardware or software recovery techniques). All the decisions concerning unknown ranges are always in the conservative worst-case direction, ensuring the correctness of results. If there is a case where a given value is used in more than one place with different ranges, the widest range is assumed.

Furthermore, value ranges are not propagated through memory. To perform accurate VRP through memory, address alias analysis is required. To keep things simple, all memory values are assumed to be 64-bit values (unless the specific data declaration states otherwise).

2.1 Finding Initial Value Ranges

Following is a list of the cases where bounds on value ranges for individual operands can be immediately determined, regardless of other instructions.

- Instructions that are declared to have narrow-width operands; for example, in terms of a HLL like C: "int a; a=a+1" where an int is 32 bits. The compiler would select a 32-bit operand for *a* and a 32-bit addition if available as an opcode. Alternatively, if binary translation is being used and the original binary has a narrow-width opcode (i.e. add_long), the binary translator can use this information (the 32-bit addition opcode) to infer that the range of the result value will be 32 bits.
- Assignments of the type (VAR=constant). The value range of VAR is set to the single constant value.
- If-condition statements whose evaluation implies a bound on the tested variable along the true or false path. For example "if (X >= 7) then…." places a lower bound on X along the true path and an upper bound on X along the false path.

2.2 Forward and Backward Value Propagation

Given the initial value range information, additional information can be derived for other instructions via a propagation process, as follows. The propagation alternates between forward and backward traversals of the program's data-dependence graph until a stable state is attained or a limit on the number of traversals is reached. During a forward propagation, the dependence-graph is traversed in a top-to-bottom style. For each instruction, the range of the output operand (if any) is determined based on the range(s) of the input operands. In a backward propagation, the traversal of the dependence-graph is done in a bottom-top manner. During this traversal the range of every instruction's input operand is set depending on the range of its output operand, the type of operation, and previous ranges of input values. The following subsections describe value range propagation for some of the more common instruction types.

2.2.1 Addition

Given the value range information of the inputs (*RangeIn1* and *RangeIn2*); the value range information of the output is:

In a forward traversal:

RangeOut.MinVal=(RangeIn1.MinVal+RangeIn2.MinVal)

RangeOut.MaxVal=(RangeIn1.MaxVal+RangeIn2.MaxVal)

If an input value can be produced by different instructions, there is an additional step of defining a range for each of the potential input values. Then, the minimum value for a given input variable will be calculated as the minimum value of all the possible producers and the maximum value as the maximum of all the possible producers. This is the conservative safe approach.

In a backward traversal:

RangeIn1.MinVal=(RangeOut.MinVal-RangeIn2.MaxVal)
RangeIn1.MaxVal=(RangeOut.MaxVal-RangeIn2.MinVal)
RangeIn2.MinVal=(RangeOut.MinVal-RangeIn1.MaxVal)
RangeIn2.MaxVal=(RangeOut.MaxVal-RangeIn1.MinVal)

If there is a case where the output value is used in more than one instruction, the above expressions are applied to all dependent instructions and the minimum and maximum values calculated are chosen. A similar approach is used in the other arithmetic instructions.

In arithmetic operations, such as the addition, overflows may occur. In this case, we assume that conventional two's complement arithmetic is used (i.e. overflows wrap around). If overflow is possible then the calculated range takes the wrap around behavior into account. Although in many cases this may be overly conservative, it ensures correctness of the generated code.

2.2.2 Loads

In a forward traversal, the range of the output value (the loaded value) is set to the maximum and minimum values that the instruction can load, depending on the instruction's opcode. During a backward traversal, the range of the loaded value is set for those instructions that use it, possibly reducing the conservative range assumed in the forward traversal.

2.2.3 Stores

Stores are ignored for forward traversals because they do not produce a result, whereas during backward traversals, the stored value may be constrained to a limited range depending on the store width specified by the opcode.

2.2.4 Branches

Unconditional branches do not provide information about value ranges. On the other hand, the comparison(s) upon which conditional branches depend are used to determine value ranges for each path. For example, consider the following code.

```
if (a<=100) then {
  /* within the if condition */
} else {
  /* within the else condition */
}
```

In the "*within the if condition*" piece of code, the maxium value of "*a*" is set to 100, and in the "*within the else condition*" piece of code, the minimum value of "a" is set to 101.

2.2.5 "Useful" Range Propagation

Some instructions constrain the range of the values due to their operations. Important cases follow.

- Logical operations. For example, AND R1, 0xFF, R2 (i.e. R2←R1&0xFF); OR R1, 0xFFFFFFFF00000000, R2 (i.e. R2←R1|0xFFFFFFFF00000000).
- Mask operations (already present in the ISA)**.** For example, MSKBL R1, 0, R2 which extracts the least significant byte of R1 and copies it to R2; the rest of bytes of R2 are set to zero.
- Limited width fields (i.e. shift amounts). For example, SRL R1, R2, R3 (i.e. R3←R1>>R2) constrains the range of R2 because the useful range for the shift amount is between 0 and 63 in the assumed ISA.

A conventional VRP would assume that the range of input values is defined to include all possible runtime values. As noted in the introduction, we are only interested in the useful values, i.e. the ones that affect program results. So for example, in the case of AND R1, 0xFF, R2 our VRP method would backward propagate the fact that just the low order byte of R1 is needed because all other bytes are set to 0 regardless of their previous value.

In order to ensure correctness when propagating "useful" range information, the technique must ensure there is no other point in the program where a wider range of the operand is semantically relevant. In the AND example given above, if R1 is used somewhere else in the program execution with a wider range, the wider range is used despite of the constraint derived from the AND operation. Similarly, "useful" backward propagation through arithmetic instructions is disallowed in order to avoid hiding overflows; for example, in the case of a loop whose upper bound is unknown, a value within the loop that is incremented may result in an overflow even if only the first byte is used.

2.2.6 Example

Figure 1 is a simple example that illustrates value propagation. Steps 1 through 7 and 9 occur during forward propagation and the 8th step during backward propagation. At step 6, the loop trip count is calculated according to the algorithm described below. *INTmin* and *INTmax* stand for the minimum and maximum possible values for an integer value; these are the default values for any integer value. In step 8, *a1in* refers to the input *a1* value. In the first step, *a0* is assigned the base address of the vector. Because it is an unknown value, the widest possible range is chosen. In step two, *a1* is assigned a 0 and the range is set to the specific value. In step 3, *a3* is assigned the product of *a1* and 4. Because *a1* is 0, the multiplication results in 0. Then, *a0* is added to *a3*, and the widest possible range is chosen for *a2*. Then, there is a store, which does not change any value ranges. Then *a1* is incremented (step 6).

At this point, the loop is detected and the loop trip count is calculated (as will be explained in next section); then the value range of *a1* at the jump is set (step 7). Once the forward propagation is done, the algorithm starts the upward propagation. In step 8, the range of *a1* as an input value is set (according to the range of the output <1,100> and the increment). In the following and last step, the value range of *a3* is updated according to the new range of *a1*.

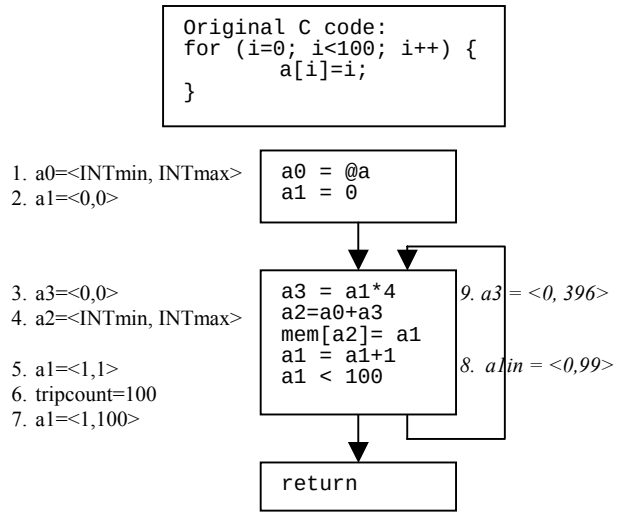

Figure 1: Example of value range propagation

2.3 Loops

Loops require special treatment. The range of values generated by instructions in a loop depends on the number of iterations. Consequently, a loop trip count estimation technique is implemented. This technique focuses on loops (typically "*for*" loops), where the iterator is of the form $x=ax+b$ (where a and b are constants and x is the iterator). This includes, for example, loops of the form: "*for (i=constant; i<constant; i=ai+b)*". Some loops that are not included are those having more than one iterator and loops that depend on a comparison to finish. This introduces limitations on the technique, depending on the source code being analyzed.

Sometimes certain parts of a given loop are executed more or less often than others parts. In these cases, a detailed loop trip count for each section can often be computed. For example, given the loop:

```
for (i=0;i<100;i++) {
  if (i<50) then ...
  else ...
}
```

The number of times each region is executed can be determined because it depends directly on the loop trip count. Nevertheless, there might be cases where this "local" trip count cannot be statically estimated:

```
for (i=0;i<100;i++) {
  if (a[i]==0) then ...
  else ...
}
```

If the trip count is partially known or not known at all, then the worst-case range is assumed. In Section 2.2.6 an example of value range propagation using the loop trip count was shown.

2.4 Other considerations

Our implementation of VRP includes interprocedural analysis. In this case, all the values passed from one function to another through registers keep their range information. At a procedure entry point, the possible ranges of the registers are analyzed, and the most conservative range is calculated. For return values, the range of value(s) returned in register(s) is set. Because value propagation through memory is not taken into account parameters passed by-reference do not have value range information.

In the case of the memory hierarchy, two approaches are possible: (1) extend the data values with size information (two extra bits that tell us whether the value is 8,16,32 or 64-bit long) and store them in the cache or (2) sign-extend values before moving them to the cache. In this work, we have assumed the first approach because it yields more energy benefits. Finally, narrow values are always kept in 2's complement to keep information about the sign.

3. Value Range Specialization

Value Range Specialization is a compile-time technique based on profiling. The technique has three steps:

1. Identification of instructions (candidates) where specialization may be profitable using basic block profiles (i.e. basic block counts).
2. Computation of the ranges of values for the candidates with the support of profiling data.
3. Specialization of the candidates that are deemed profitable.

The first step defines the candidates for specialization. In other words, it defines the set of instructions that have the better chance of being profitable specialization points. These candidates are then profiled. With the profiling information on the value ranges, the benefits of specializing each candidate is then evaluated. If there is energy savings expected from specializing the candidate for a certain range, the specialization phase clones the section of code under consideration, it adds the tests for the specialized code, and propagates the new range to the specialized region.

3.1 Computing the Energy Savings of Specialization

Let us define *InstSaving(I,r,min,max)* as the energy saved when the input register *r* of instruction *I* has a given value range [*min,max*]. *InstSaving* is calculated in the following way: given the range of the input operands (one of which is *r*). The range of the output register is set; then, if the width of the output register has changed (meaning it may need a narrower opcode), the energy savings are computed. These instruction-type dependent energy savings have been empirically defined for each instruction type and operand-width through the observation of its energy requirements (see Section 4.1 for the experimental framework). The energy savings for a given instruction *I*, denoted as *Savings(I,r,min,max),* is the energy saved for all dependent instructions on the output register *r* of instruction *I.*

$$Saving(I,r,\min,\max) = \sum_{\forall D \in Uses(I,r)} \left[\begin{array}{l} InstCount(D) x InstSaving(D,r,\min,\max) + \\ Saving(D,r',\min',\max') \end{array} \right]$$

Where *r'* is the output register of instruction *D* and its range is [*min',max'*]. *Uses(I,r)* are all the instructions that use the output register *r* of instruction *I*. *InstCount(D)* is the number of times instruction *D* is executed.

The savings of each instruction type have been empirically computed by determining the energy-savings (in nJoules) of a given instruction type with different operand width. In this case, the whole SpecInt 95 benchmark suite was run to completion with their reference inputs. Table 1 depicts the energy savings for ALU operations:

This function, *Savings(I,r,min,max)*, will be useful both when looking for candidates to profile and then, together with the profiling data, for estimating the run-time energy savings.

Table 1: Energy savings for ALU operations (nJoules)

Source Width → Dest. Width ↓	64	32	16	8
64	-	-1	-3	-6
32	1	-	-2	-5
16	3	2	-	-3
8	6	5	3	-

3.2 Computing the Cost of Specialization

The benefits of specialization have to be weighed against the costs incurred due to runtime tests. The cost of such tests depends on the actual range being tested. For instance, if the minimum and the maximum of a given range are the same value, just one comparison is needed; otherwise, two tests are needed (one to check the minimum and one to check the maximum). Due to the way tests are implemented in the Alpha architecture, testing for a zero value can be done in one single instruction but testing for a non-zero value has to be done with two instructions.

In order to compute the cost in terms of energy, each instruction needed in the test is given an energy requirement in relation to its instruction-type (branches, comparisons, and additions).

$$InstCost(I,r) = \left(\begin{array}{l} Nbranches * CostBranch + \\ NComparisons * CostComparison + \\ NAdds * CostAdd \end{array} \right)$$

$$Cost(I,r) = InstCount(I) * InstCost(I,r)$$

Nbranches, NComparisons and *Nadds* are, respectively, the number of branches, the number of comparisons and the number of additions needed to perform the test, and *CostBranch, CostComparison* and *CostAdd* are the energy costs for each of these instructions.

3.3 Identifying the Candidates for Specialization

In order to avoid profiling the values generated by every single instruction in the program code, we attempt to identify candidates for which some potential savings are likely. To accomplish this, a preliminary benefit analysis is done assuming that the cost of specialization is a single comparison (the minimum possible cost). This preliminary analysis significantly reduces the number of candidates and, therefore, the amount of required profiling.

Given the set of candidates to be profiled, we use the scheme proposed by Calder et al. [7]. The technique adds a function in the program that is called at the profiling points and stores the actual value in a fixed-size table every time it is called. If the value is already in the table, the count of that value is incremented. Otherwise, if the table is not full, the value is added. If the table is full the value is ignored. Periodically, the table is cleaned by evicting the least frequently used values from the table: this allows new values to enter the table. The total number of times the profiling point is executed is also kept in a separate counter.

3.4 Specialization of the Candidates

Specialization is done in two steps. First, the profile data is analyzed and the set of candidates is reduced to those that still provide benefits, considering the run-time range information. Then, the program is transformed accordingly. The benefit of specializing is computed through the formulas presented earlier. For each candidate the energy savings and the energy costs are computed using the profile information. Specializing a given instruction *I*, for a range of [*min,max*] of its output register

r, is worthwhile if the overall benefit given in the following expression is positive.

$$Savings(I,r) * Freq(\min, \max) - Cost(I,r)$$

Where *Freq(min,max)* is the frequency that the range of the value of *r* is within the range of the specialized region, i.e., it is the frequency that the program path will go through the specialized code.

There are two possible transformations of the code. In the case where the range [min,max] results in specialization for a single value (i.e. min=max), a value specialization method is used. Otherwise, a value range specialization method is used. The value specialization method is based on the one proposed by Muth, et al. [16]. In the case of the value range specialization, the method is a variation of the single value specialization adapted to ranges.

VRS basically duplicates the regions of code that are affected by the specialization, and then inserts tests to dynamically select the region that will be executed: either the specialized or the non-specialized one. The tests consists of two comparisons and an AND operation followed by a conditional branch. The condition tested is (x>=min && x<=max) where x is the value that is being specialized. This condition ensures the correctness of the execution of the specialized code.

4. Evaluation

4.1 Experimental Framework

To evaluate the proposed techniques, we use an extended version of Wattch [4] for power analysis. The extensions include activity counts for all the blocks to allow proper data-specific power modeling. The main architectural parameters of the out-of-order machine are described in Table 2. VRP and VRS were implemented in the Alto [15] binary optimizer. Some modifications were done in the optimizer by expanding the use-def algorithm to allow for inter-basic-block and inter-procedural, forward and backward traversals. In order to implement VRS, the profiling part of Alto was modified by inserting the capability of profiling value ranges, computing the cost/benefit equations in terms of energy, and modifying the specialization function to insert the correct specialization code for value ranges. We used the programs from the SpecInt95 suite with their reference inputs (and train inputs to perform profiling). All benchmarks were compiled with the HP-Alpha C compiler. The resulting binaries were optimized at the maximum level and post-processed with our binary optimizer in order to perform VRP, the overhead of the VRP was minimal (ranging from 0.02% to 0.08% increased processing time). All benchmarks were run to completion both for profiling analysis and for energy evaluation.

Table 2: Machine parameters

Parameter	Configuration
Fetch Width	4 instructions
I-cache	64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty.
Branch Predictor	Combined predictor of 1K entries with a Gshare with 64K 2-bit counters, 16 bit global history, and a bimodal predictor of 2K entries with 2-bit counters.
Decode/Rename width	4 instructions
Max. in-flight instructions	64
Retire width	4 instructions
Functional units	3 intALU + 1 int mul/div + 3 fpALU + 1 fp mul/div
Issue mechanism	4 instructions Out-of-order Window based
D-cache L1	64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty 3 R/W ports
I/D-cache L2	256 KB, 4-way set associative, 64-byte lines, 6-cycle hit time 16 bytes bus bandwidth to main memory, 16 cycles first chunk, 2 cycles interchunk
Physical registers	96

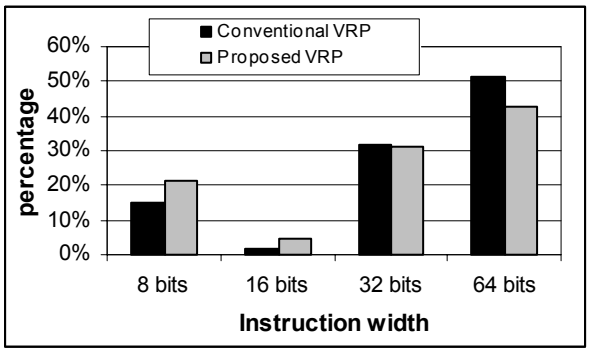

Figure 2: Dynamic instruction distribution according to value range

4.2 Benefits of "Useful" Value Range Propagation

Figure 2 shows the distribution of the run-time instructions (on average for SpecInt95) according to the widths determined by value range propagation. The extended value range propagation technique that distinguishes useful ranges from actual ranges (labeled in the figure as *Proposed VRP*) can identify more instructions with narrow operands than the *conventional VRP*. For instance, the number of 64-bit instructions is reduced from a 51% to a 42%.

4.3 Required Opcode Extensions

Depending on the initial instruction set, some new opcodes may have to be implemented to fully support the proposed VRP technique. In this section, we analyze extensions required by the Alpha ISA.

The technique as presented applies only to integer computations. In addition, branch instructions are not taken into account because they manipulate addresses (i.e. wide data). The Alpha ISA already supports byte, halfword, word and double word memory operations (Load[1] and Store). In Table 3, the distribution of other instructions is presented, ordered by their dynamic percentage of
occurrence (for SpecInt95). The first column lists the operation type, the second column gives the percentage of dynamic instructions of the given type, and the remaining columns give the various data widths as a percentage of instructions *of that type*, i.e. as a percentage of the column 2 percentages. Hence, 24 percent of the ADD instructions operate on 8 bits, or about 6.65 percent of all dynamic instructions.

Because the MUL operation is rarely used and almost half the time it uses 64 bits, there is no advantage to implementing narrow-width MUL instructions. Similarly, there are very few 16-bit operations overall. Only ADD is likely to be important enough to warrant a 16-bit version.

Table 3: Distribution of operation types

	Percentage of run-time instructions	64b	32b	16b	8b
ADD	27.66	58.04	14.37	10.98	24.03
MSK	5.18	35.50	13.41	13.57	37.63
CMP	3.78	6.18	7.79	20.53	64.84
SHIFT	2.75	29.91	19.56	22.90	32.23
SUB	2.35	13.87	15.76	16.82	60.78
AND	1.92	16.11	8.73	27.03	48.94
OR	1.79	23.77	5.90	3.53	68.62
XOR	1.15	17.04	7.52	29.77	43.89
CMOV	0.80	18.42	20.04	25.33	39.61
MUL	0.18	47.95	23.39	7.04	26.87

Overall, new opcodes added to the Alpha ISA are: byte and halfword addition; byte substraction; byte and word logical operations (and, or, xor), and byte and word shifts, conditional moves and comparisons.

If narrow data-width opcodes are not implemented, value range propagation must ensure that whenever a wider instruction is used, the values read at run time contain significant data for all the input bytes. For example, it would be unacceptable for the result of a 16-bit load to be used as input to a 32-bit multiplication.

[1] Although the byte and halfword load is unsigned it has no effect on the value range propagation.

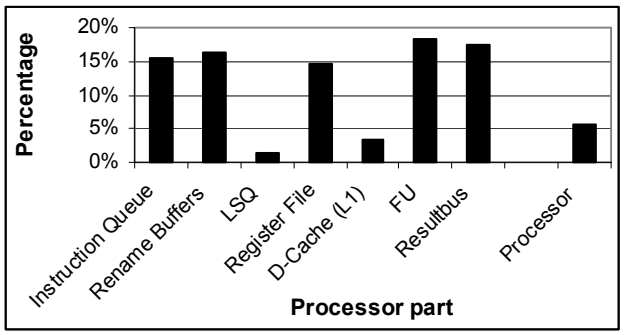

Figure 3: Energy savings with VRP

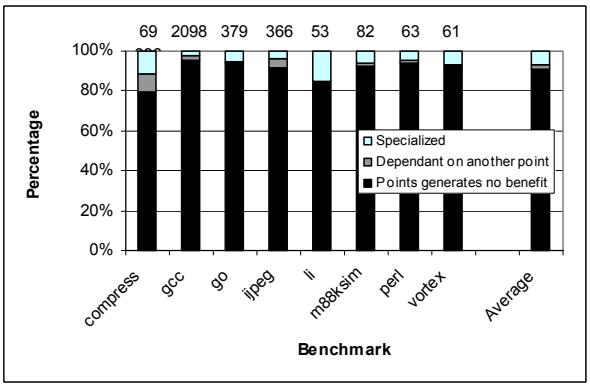

Figure 4: Distribution of the points profiled after specialization

4.4 Energy Savings

The VRP mechanism does not affect the performance of the benchmarks because it just re-encodes the instructions with narrower opcodes. These narrower opcodes are then used to gate-off the portions of the datapath that are not needed for the computation of the final results. Figure 3 shows the energy savings of the proposed VRP mechanism. The results for the rename logic, branch prediction, instruction cache and second level cache are not given because are not affected by VRP. Nevertheless, the power consumption of these components is part of the "processor" column, which includes the whole processor.

The power savings is up to 18% for the most data intensive structures (i.e. functional units), but for most other structures the savings are around 15% (instruction-queue, rename buffers, register file and the result busses). The memory management structures (LSQ and L1 data cache) show a minor improvement because they handle memory addresses (typically, large values). Overall, the savings in these structures turn into an overall energy savings close to 6% on average for the SpecInt95 suite.

4.5 Benefits of Value Range Specialization

As explained in Section 3, VRS is supported by data

profiling. Figure 4 shows the distribution of the points (i.e. instructions) that were profiled. The number on the top of

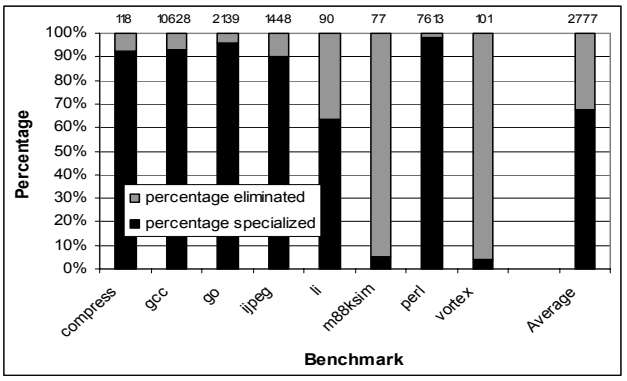

Figure 5: Distribution of the specialized instructions at compile-time

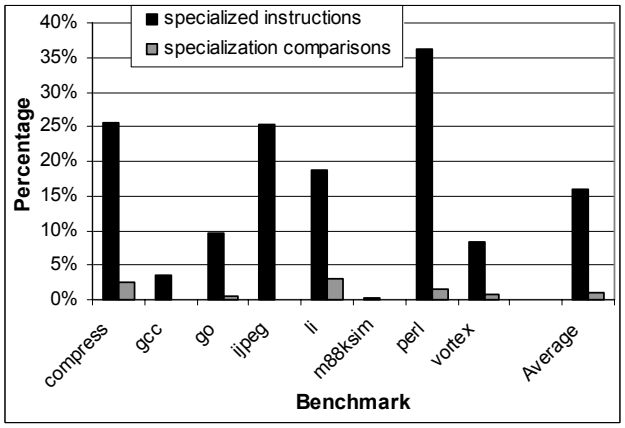

Figure 6: Distribution of run-time instructions

each bar is the total number of points profiled for each benchmark. Several filters were implemented in order to select only those profiled points that may result in an energy benefit. As shown in Figure 4, most of the profiled points are eliminated because they produce no benefit (88% of the points). The other reason for eliminating a point is that it is included in a region optimized as the consequence of another point. On average this happens for only 2% of the points. In the end, the number of specialized points is 7% of those profiled. This means that on average 15 points are specialized per benchmark, and individual benchmarks range from 3 (perl) to 55 (gcc).

Figure 5 shows the distribution of static instructions specialized for each benchmark and the average for all the benchmarks. In most cases instructions are specialized by establishing a new, more concise value range, however, there is also a significant number of instructions (especially in m88ksim and vortex) removed from the specialized sections of the code. This is a consequence of specializing for a given value and applying constant propagation.

At run time, the distribution of specialized instructions is shown in the first column of each benchmark in Figure 6. The second column reports the percentage of instructions needed to specialize a point (comparisons, etc). As shown in Figure 5, m88ksim and vortex eliminate almost all the

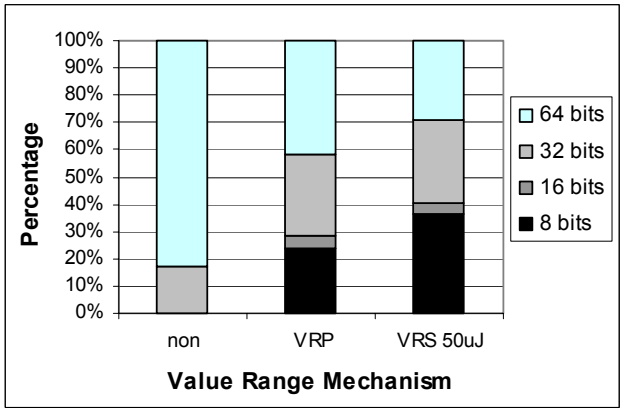

Figure 7: Run-time instructions according to width

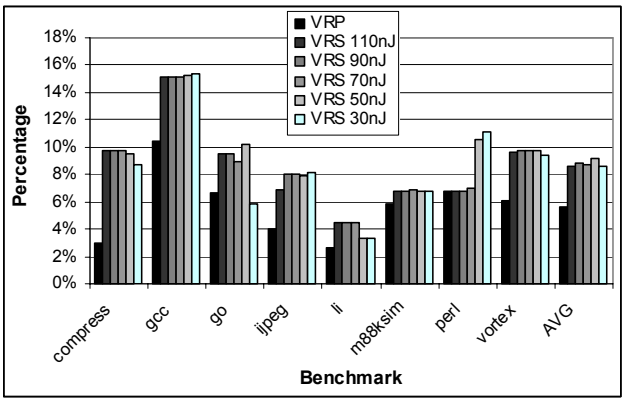

Figure 8: Energy savings for Spec95

specialized instructions, which results in a minimal run-time occurrence of specialized instructions. On average, more than 15% of the executed instructions are specialized -with a maximum of 35% for perl; while the comparisons represent 1% of the executed instructions on average.

Another interesting statistic is the distribution of the size of run-time instructions for the different value range propagation mechanisms. Figure 7 shows the distribution of the run-time instructions on average for SpecInt95. The first column is the baseline in which no mechanism is implemented. In this case, most of the instructions deal with 64-bits or 32-bits. When VRP is implemented, the number of 64-bit instructions diminishes to 40%, and it further decreases to 30% with VRS. On the other hand, the VRS mechanism increases the number of 8-bit instructions.

Figure 8 shows the energy savings in relation to the baseline (the architecture without any value range mechanism). In the case of the VRS mechanisms, different configurations have been studied depending on the cost (in nanoJoules) of specializing. In most of the benchmarks, all

five alternatives of VRS perform similarly (in terms of energy), which means that the candidates chosen for specialization are essentially the same in these configurations.

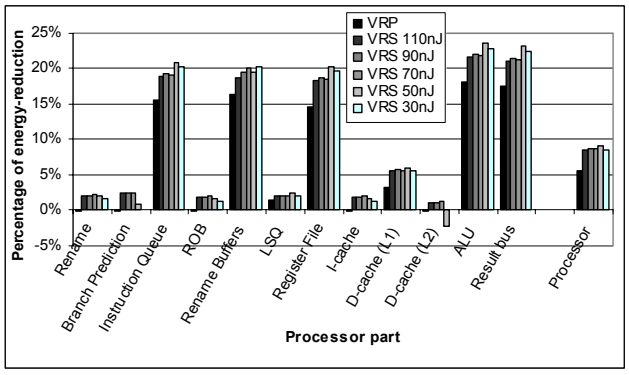

Figure 9: Energy benefits for the different parts of the processors

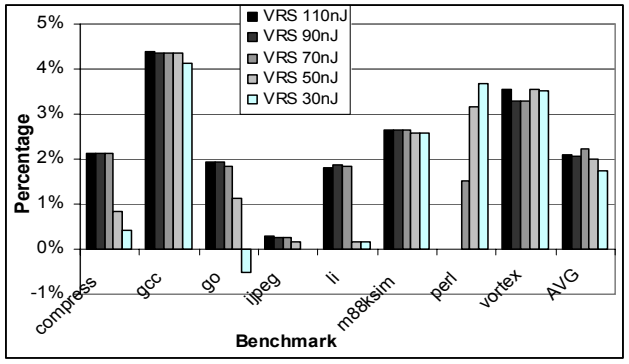

Figure 10: Execution time savings

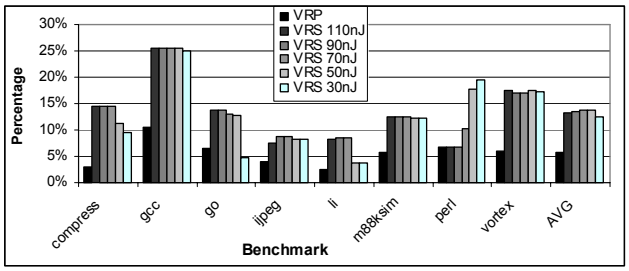

Figure 11: Energy-Delay^2 benefits for the Spec95

Detailed energy benefits for all parts of the processor are shown in Figure 9. Due to the nature of the mechanism, the parts that benefit most from value range mechanisms are those that directly manipulate data values (i.e. instruction queue, rename buffers, register file, functional units and result buses). Because VRS modifies code (by adding comparisons to perform specialization and removing instructions in the specialized sections), results for all parts of the processor may realize some benefit. Minimal energy benefits arise from reduction in number of instructions, but more impressive benefits arise from the data-dependent structures – the ones the technique focuses on. Overall, the energy benefits of the VRS mechanism are around 9% while most of the data intensive structures get over 20% of energy savings.

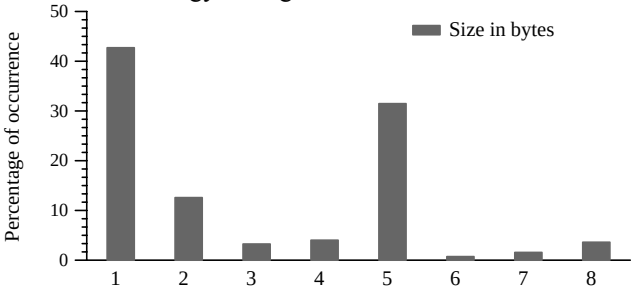

Figure 12: Data size distribution for the SpecInt 95

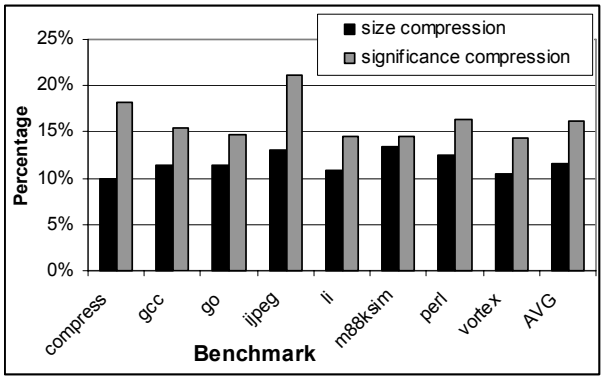

Figure 13: Energy Savings for the different hardware approaches (Average SpecInt 95)

Because the code is modified by inserting the comparisons eliminating the instructions in the VRS sections of specialized code, it is important to see the impact on the execution time of the benchmarks with the VRS technique. As noted earlier, the VRP mechanism does not affect performance because it does not add/remove any instruction to/from the code; it just re-encodes the instructions with narrower opcodes. Figure 10 shows the reduction in execution time. Except for one configuration of VRS with benchmark go, the rest of the binaries perform better when VRS is included.

In order to compare the benefits of using the VRS mechanism, the energy-delay^2 metric [2] provides a fair comparison of all the design points taking into account both energy and execution time. Figure 11 shows the improvement in energy-delay^2 for all the benchmarks in SpecInt 95. On average the benefits of the VRP mechanism are a little above 5% but when using the VRS mechanism the benefits rise to almost 15%. For *gcc* the benefit rises to 25%.

4.6 Comparison with a Hardware Approach

To compare the proposed scheme with a hardware approach, we implemented the scheme given in [9] adapted to work on a 64-bit architecture. Two data compression methods are used. The first is *significance*

compression where seven tag bits are added per data word (64 bits) to indicate the number of significant bytes. The second method is *size compression* that uses two bits per data word

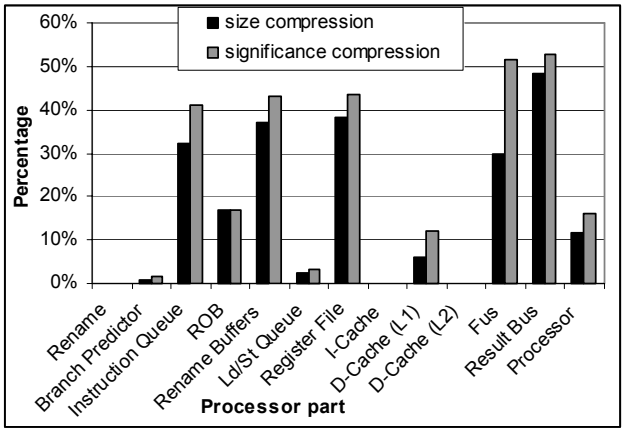

Figure 14: Energy savings for each processor part (Average SpecInt 95)

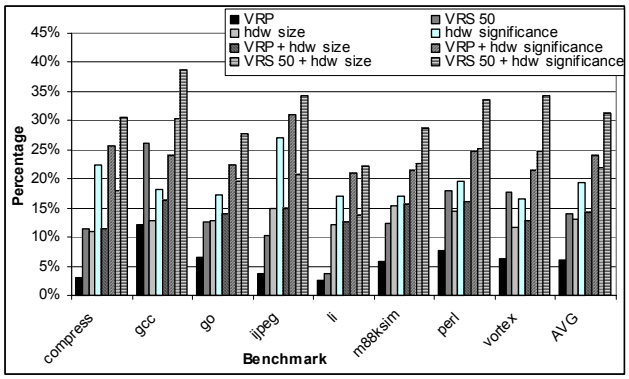

Figure 15: Energy-delay2 savings for different hardware and software configurations

to indicate whether the value is 1, 2, 5 or 8 bytes long. This choice is based on analyzing the value range distribution for SpecInt95 (see Figure 12). This encoding achieves an average size of 26.7 bits per value. The choice of 5 bytes rather than the more natural 4 bytes is heavily influenced by memory addresses that are between 33 and 40 bits long (note the peak at 5 bytes in Figure 12). Overall, the distribution in Figure 12 shows a large potential for dynamic hardware-implemented operand-gating techniques because of the large number of narrow values (i.e. 43% just need 1 byte to be represented).

Figure 13 shows the energy reduction of the hardware approaches. On average, overall energy is reduced by 15%. The hardware approach has the advantage of enabling multiple-size operands in the functional units. For example, an addition of a 16-bit plus a 32-bit value producing a 64- bit value could be possible. Furthermore, the data-width of the same instruction for several different executions can be different (this being a big difference with respect to the software schemes presented here). Overall, the hardware approach has more opportunities to reduce the energy consumption despite the cost of keeping several bits per data word. Figure 14 shows the savings for each structure of the processor. Most benefit arises from those structures directly manipulating values.

Figure 15 presents the energy-delay^2 savings when combining the hardware and software schemes. Both hardware and software schemes cooperate to achieve a higher percentage of energy-delay^2 savings. The energy-delay^2 savings of the VRS mechanism are (on average) very close to those of the *significance compression* mechanism and better than the *size compression* mechanism. For some benchmarks (gcc, perl and vortex), the VRS mechanism has even better performance in terms of energy-delay^2 than the hardware mechanisms. The reasons behind this competitive performance are the reduction in the execution time, the minimal extra cost (in hardware) of the software mechanism, and the accuracy of value range analysis. The benefit of using the profiling technique is even higher when a hardware scheme is also used because the additional energy savings of the hardware scheme are added to the reduction in execution time of the profiling mechanism. In this sense, *gcc* achieves almost a 35% energy-delay^2 benefit when combining the *significance compression* and VRS

4.7 Hardware and Software Trade-offs

Both hardware and software approaches to operand gating have advantages and drawbacks. In general, the hardware approach requires no change in the ISA and no recompilation or binary translation. On the other hand, software approaches require simpler microarchitecture support. The changes to the microarchitecture in order to support the hardware mechanism are basically the following: (1) The use of tag bits (2 or 7 as explained earlier in this section) that are stored with all values (in the register file and in the cache). (2) The tag bits of a register value must be read before the data in order to know which bytes are significant and require a register file access. This may introduce a delay in the register file access (this potential penalty was not considered in the evaluation above). (3) The functional units must be able to sign extend some source operands (e.g., when adding a 16-bit value with a 32-bit one). The functional units must generate the tag bits for results. The software approach encodes the width of the values used in the opcode and thus requires minimal hardware changes.

In terms of value range propagation, the proposed software approach can only identify and propagate useful value ranges through the static program code, and may eliminate significant data bits in the process if they are not required for semantically correct results. On the other hand, a dynamic hardware-based significance compression mechanism is able to detect value ranges more accurately than a static software method. This suggests a *cooperative*

hardware-software approach where both methods are used in the same processor. To implement this cooperative approach, value compression is initiated through both compiler-generated instructions and hardware-generated compressed values at runtime. In both cases, two significance compression tag bits follow values in the pipeline. Although the compiler generates opcodes for 8, 16, 32 and 64-bit wide data, when a hardware compression mechanism is used, the manipulated values may have 8, 16, 40 or 64 bits within the microarchitecture. The cooperative hardware and software approach achieves a 28% overall energy-delay^2 savings in relation to the baseline architecture.

Overall, the software scheme seems more appropriate to an environment where the ISA already supports multiple-size operations and where hardware overhead is unaffordable. On the other hand, for scenarios where recompilation is an issue and where hardware changes are not critical, the hardware approach may be best. Finally, only for environments where power is very critical, one may be willing to pay the overheads of a combined software and hardware approach in order to achieve the extra 12% energy-delay^2 reduction that the combination provides over the hardware-only scheme.

5. Related Work

Techniques using value range propagation have been used in high-level code transformations [5][13][18][20]. Our work applies such techniques at a much later code development step due to the use of a binary optimizer [15]. The technique proposed here is more CPU specific but totally compiler independent. For instance, forward propagation and loop analysis have some precedent in high-level program analysis mainly used during symbolic analysis for different purposed such as parallelizing compilers [1]. Other applications of value range propagation include VLSI synthesis targeted at custom processors (i.e. that execute only a certain application) [10][13][20]. In [10] the authors propose a datapath-width optimization framework based on runtime information concerning the size of operands. This information is used to rewrite the source code where each data type has a width component. "Useful" value range propagation has not been previously implemented, although Budiu et al. [5] implemented useful bit-width computation (where each bit was tagged whether it was useful or not). Alternatively, value range propagation has been used for branch prediction [18], although in this case backward propagation and loop-carried expressions were not considered.

Brooks et al. [4] suggest a runtime mechanism that detects the widths of the instructions that will be executed and packs them so they are executed at the same time taking advantage of the 64-bit wide functional units. Loh [12] suggests a similar approach by speculating about data-width locality. Nakra et al. [17] propose a similar approach for VLIW architectures.

Data value compression has been studied for different functional blocks. Sato et al. [19] showed the power savings in a data value predictor when storing narrow data. Villa at al. showed the effect of compressing zero values in the cache [21]. Yang and Gupta studied efficient I/O [22]. Data value compression was used for a broader scope in [9]. In that case, the overall microarchitecture was redesigned to work on narrower data.

Alternative approaches for narrow values include vector or multimedia extensions like MMX-SSE, AltiVec and 3DNow!. These approaches are most efficiently when used directly by the programmer. As presented in Nakra's work [17], unless the compiler is able to compact different arithmetic operations in one vector instruction; the compiler is very inefficient when producing the vector instructions. Nevertheless, the approach presented in here could improve these vector extensions and thus, the energy-efficiency of the processor.

6. Conclusions

Operand gating has been shown to be an effective way to increase the processor energy-efficiency. A software technique for operand gating has been proposed. With minimal extensions to the ISA, the software approach is able to extract width information from the binary code and then propagate it to the microarchitecture. By extending the propagation through profiling to further constraint the value ranges, multiple versions of code for certain program segments are created and the most efficient one is dynamically chosen based on the actual values of the operands. The proposed technique achieves an overall 14% energy-delay^2 reduction for the SpecInt95 set. It achieves a much better reduction for data-intensive structures where the energy-delay^2 benefits are over 20%.

A hardware-only scheme was investigated for comparison. It requires some extensions to the microarchitecture and achieves an average energy-delay^2 benefit of 15%. The hardware scheme can reduce the energy for any data that has a small number of significant bits. Because the values are checked dynamically at run-time, operand gating is optimized for each particular instance of any operand. On the other hand, the software approach has to make conservative assumptions for two reasons. First, it has to assume the worst-case range when the compiler does not know the potential values of a variable. Second, it uses a unique range for each static instruction that includes all the values of the corresponding dynamic instances of the static instruction. However, a software scheme has a number of advantages. For instance, software analysis can detect useful bits, which in general are more restrictive than significant bits. This suggests that a combined hardware-software approach can further

reduce the energy consumption. Our experiments show an average energy-delay^2 benefit of 28%.

Acknowledgments

This work has been supported by the Spanish Ministry of Education contract CICYT- TIC2001-0995-C02-01. The research conducted in this work has been done using the resources of the CEPBA. Ramon Canal would like to thank his fellow PBCs for their patience and precious help.

References

[1] W.J. Blume, "Symbolic Analysis Techniques for Effective Automatic Parallelization", Ph.D. Thesis University of Illinois at Urbana-Champaign, 1995.

[2] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, P. Cook, "Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors", IEEE Micro v 20-6, Nov/Dec 2000 pp. 26-44

[3] D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance", in Proc. of 5th. Int. Symp. on High- Perf. Comp. Arch.., 1999.

[4] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimization", in Proc. of the 27th Annual International Symposium on Computer Architecture, June 2000.

[5] M. Budiu, S. Goldstein, M. Sakr and K. Walker. "BitValue inference: Detecting and exploiting narrow bitwidth computations". In Proceedings of the EuroPar 2000 European Conference on Parallel Computing, Munich, August 2000.

[6] G. Cai and C.H. Lim, "Architectural Level Power/Performance Optimization and Dynamic Power Estimation", in the Cool Chips tutorial of the 32nd Int. Symp. on Microarchitecture 1999.

[7] B. Calder, P.Feller and A. Eustace, "Value Profiling", in Proc. Of the 30th International Symposium on Microarchitecture, Dec. 1997, pp. 259-269.

[8] B. Calder, P. Feller and A. Eustace, "Value Profiling and Optimization", Journal of Instruction-Level Parallelism 1 (1999), pp. 1-6.

[9] R. Canal, A. González and J.E. Smith, "Very Low Power Pipelines using Significance Compression", in Proc. of the 33rd International Symposium on Microarchitecture, Dec. 2000.

[10] Y. Cao, H. Yasuura, "A System-Level Energy Minimization Approach Using Datapath Width Optimization", in Proc. of the International Symposium on Low Power Electronics and Design, August 2001.

[11] R. Iris Bahar and S. Manne, "Power and Energy Reduction via Pipeline Balancing", in Proc. of the 28th Annual International Symposium on Computer Architecture, June 2001.

[12] G.H. Loh, "Exploiting Data-Width Locality to Increase Superscalar Execution Bandwith", in Proc. of the 35rd International Symposium on Microarchitecture, Istanbul, Turkey, November. 2002.

[13] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber and T. Sherwood, "Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators", IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 20 n. 11 , Nov 2001 pp. 1355 -1371 .

[14] S. Manne, A. Klauser and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction", in Proc. of the 25th Int. Symp on Comp. Architecture, June 1998, pp.132-141.

[15] R. Muth, S. Debray, S. Watterson and K. De Bosschere "Alto: A Link-Time Optimizer for the Compaq-Alpha", Software Practice and Experience 31:67-101, January 2001.

[16] R. Muth, S. Watterson, S. Debray, "Code Specialization based on Value Profiles", in Proc. 7th International Static Analysis Symposium, June 2000

[17] T. Nakra, B. Childers, and M.L.Soffa, "Width Sensitive Scheduling for Resource Contained VLIW processors", Workshop on Feedback Directed and Dynamic Optimizations (MICRO33), Dec. 2001.

[18] J. Patterson. "Accurate Static Branch Prediction by Value Range Propagation". In Proceedings of the Conference on Programming Languages Design and Implementation, pp 67-78, June 1995.

[19] T. Sato and I. Arita, "Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values", in Proc. of the 2000 Int. Conf. on Supercomputing, May 2000, pp.196-205.

[20] M. Stephenson, J. Babb and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation", in Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2001), pp. 108-120.

[21] L. Villa, M. Zhang, and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction", in Proc. of the 33rd International Symposium on Microarchitecture, Dec. 2000.

[22] J. Yang and R. Gupta. "FV encoding for Low PowerData I/O". In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, pages 84-87, Hungtinton Beach, August 2001.