On Availability of Bit-narrow Operations in General-purpose Applications

Darko Stefanović and Margaret Martonosi

Department of Electrical Engineering
Princeton University
Princeton NJ 08544, USA
{darko,mrm}@ee.princeton.edu

Abstract. Program instructions that consume and produce small operands can be executed in hardware circuitry of less than full size. We compare different proposed models of accounting for the usefulness of bit-positions in operands, using a run-time profiling tool, both to observe and summarize operand values, and to reconstruct and analyze the program's data-flow graph to discover useless bits. We find that under aggressive models, the average number of useful bits per integer operand is as low as 10, not only in kernels but also in general-purpose applications from SPEC95.

1 Introduction

The purpose of a *bit-set analysis* is to identify which bit positions in various values computed by a program carry essential information and which do not, and, consequently, which bit positions must be computed by instructions and which do not. A typical programming language does not allow one to express bit usage requirements, hence in a compile-time analysis they must be inferred by the compiler. A typical RISC instruction set is too crude to express arbitrary bit usage requirements (at most, it may express a few fixed sub-words), hence in a run-time analysis these requirements must be re-derived from the executing program. Once useful bit-sets are identified for each value, it may become possible to simplify the hardware that implements the program's instructions so that it will produce only the useful bit-sets. With luck, the smaller circuitry will also be faster; and, fitting a larger number of smaller circuits in the same silicon area allows potentially higher execution parallelism. Our goal is ultimately to apply the results as one of the criteria for code partitioning in hybrid fixed-configurable processors.

2 Background

In the early days of computing, narrow-width computation was available in hardware and programmers could write code by hand to exploit it. The integer types of C still support this at the programming language level, albeit in an ill-defined and non-portable manner. With the advent of RISC processors, uniform, word-width operation became the norm; and only recently have we seen support for sub-word operations again, in the multimedia extensions to RISC instruction sets.

During the last decade there has been interest in detecting and exploiting narrow operations through compiler analysis. Razdan and Smith [1, 2, 3] proposed a static analysis to narrow widths for use with a tightly-coupled configurable functional unit. Their analysis is a bit-wise abstract interpretation over the bit positions of each variable in an internal representation of the program, with forward and backward passes to characterize the generation and the use of bit positions.

More recently, Stephenson, Babb, and Amarasinghe [4], and independently Budiu, Goldstein, Sakr, and Walker [5] have constructed static analyses to identify narrow computation, for use in various settings. Stephenson's analysis *Bitwise* is an abstract interpretation that computes data ranges: the minimum and maximum value that may be assumed by variables in an internal representation based on SUIF [6]. This interval analysis is equipped with useful heuristics for recognizing loop induction patterns. It is integrated with powerful pointer and alias analyses [7]. Bit-width savings are reported to be 15–80% (static count) on a set of small (kernel) programs. In conjunction with silicon compilation, it achieves up to 86% reduction in silicon area on some programs.

Budiu's analysis *BitValue* is primarily a bit-wise analysis, somewhat similar to Razdan and Smith's, but it explicitly treats constantness (forward pass) and uselessness (backward pass). It operates on an internal representation, also based on SUIF. Underlying analyses (such as the alias analysis which affects recognized data dependences) are not as sophisticated as in *Bitwise*. *BitValue* analysis optionally includes an ad-hoc loop induction analysis. With both turned on, bit-set savings on the order of 29% (static count) and 30% (dynamic count) are reported on a set of programs from SPEC95 and Mediabench.

Brooks and Martonosi [8], on the other hand, use dynamic profiling to observe the set of operands presented to individual arithmetic instructions. Under the assumption that each operand's high-order bits can be elided to a single sign bit, they find that significant savings are possible in the number of bits needed for the representation, and in integer unit's consumed power, as much as 60%.

In short, previous research has demonstrated the availability of bit-narrow operands. However, different studies have assumed different definitions of which bit positions are useful. For instance, should only high-order bits be considered as potentially useless; or, should bits proven constant by analysis be considered useless; or, what is the granularity of usefulness decisions? As a result the effectiveness of different compile-time analyses cannot be compared, nor can the different definitions be compared. In this study we use run-time knowledge of observed values, so that our precision is a bound on that achievable by compile-time analyses. We are then able to compare different definitions of which bits are useful, and how they are tallied.

Our run-time program analysis incorporates data-flow graph reconstruction and propagation of bit-use information backward in the graph, which allows the elimination of bits as unused, unlike with instruction-by-instruction observations [8]. Since this feature can be turned on or off, we can evaluate the effectiveness of such an analysis.

The broader purpose of the tools built is to discover bit-narrow operands in general-purpose applications, their prevalence being one of the criteria for the selection of code sections for implementation in configurable hardware.

3 Run-time program analysis

3.1 Data-flow reconstruction

The front-end of our tool is a modified version of the *sim-profile* tool of the SimpleScalar architectural simulation toolset [9], for the Alpha instruction set architecture as target (including multi-media extensions). We track the flow of data through an architecture's functional and storage units, and dynamically construct a data-flow graph. Our basic notion is that of a *value*: an instruction uses a number of input values and produces a number of output values. With each value we record the observed register contents—the raw bits of the value.

A value has a *type*, which is inferred dynamically from its usage. Ideally the type of a value is resolved to *integer* or *floating-point*. Occasionally this is not possible, so the type lattice includes the elements integer, floating-point, unknown, and conflict.

We keep track of a program's state, including both registers and memory. Each register contains at most one value. Registers that a program has not touched contain no value. When an instruction creates an output value and stores it in a register, we note that the register contains that value. When an instruction reads an input value from a register, we retrieve the value contained in the register. However, if no value is currently contained in the register, we make a fresh value: this allows tracking to start from initial externally defined register contents (and to restart after a system call). Memory on the Alpha is both byte-addressable and byte-accessible. Our memory model is a mapping from byte addresses to pairs: $a \mapsto \langle v, i \rangle$, where i is the offset of the byte at memory address a within a stored value v. A store instruction causes the mapping to be updated for a number of successive addresses. At any given time, a value is said to have a register presence if it is contained in any register, and to have a memory presence if any of its constituent bytes have a presence anywhere in memory.

The instructions executed by the program are represented by instruction *nodes*. Each node indicates the program address of the instruction, its class and opcode, and its input and output values. Unlike operation nodes in compilers, this is a run-time concept, and each dynamic execution of an instruction results in a distinct node. No instruction nodes are created for instructions identifiable as data transport: register moves and memory loads. In the case of memory loads, this means that the retrieved bytes are checked to see if they exactly match one entire stored value. The values are appropriately bypassed from the producing node to the ultimate using node. Thus the data flow of the computation is reconstructed independent of the data storage decisions and layout.

While instructions are simulated, new nodes and values are added to the graph. Bit-set analysis is performed on the oldest values, and upon analysis, the oldest values (and their nodes) are discarded. The difference between the oldest and the newest nodes, i.e., the lookahead that the bit-set analysis enjoys, also determines the amount of storage the simulator needs and its speed of simulation. With the current prototype implementation we can achieve lookaheads of a few million instructions.

3.2 Bit-set analysis

Our bit-set analysis comprises two independent components: analysis of the dynamic data-flow graph, and computation of a static summary.

The goal of dynamic data-flow graph analysis is to infer that certain bit positions of the analyzed values are not needed for subsequent computation. The main source of such information, in the Alpha instruction set, are logic instructions. For instance, consider an instruction `SRL r1, 8, r2`, which shifts the contents of `r1` by 8 bits and stores the result in `r2`, and suppose that this instruction is the only use of the value present in `r1`. The instruction shifts the uppermost 7 bytes of `r1` into the lowermost 7 bytes of `r2`, and writes zeroes into the uppermost byte of `r2`. The lowermost byte of `r1` is discarded: it is not useful in this instruction. Since this instruction is the only use of the value in `r1`, the lowermost byte of that value need not be computed at all. The information flow in this analysis is backward: all the uses of a value, and their subsequent uses, etc., affect which bits of the value are truly needed. In case of finite lookahead, we conservatively assume that all values that have a presence (in registers or memory) at the time of analysis will have all their bits used by future computation. The result of dynamic data-flow graph analysis is an annotation *useful/useless* on the bits of each dynamic value.

We compute a static summary over all executions of each static instruction in the program, more precisely, of each operand of each static instruction. By observing the bit-values (0 or 1) assumed by a bit-position of an operand over all executions, we annotate a bit-positions with *Always-0* or *Always-1* if it is constant over all executions. If dynamic data-flow graph analysis is performed, we also statically summarize its results and annotate a bit-position with *Sometimes-useful* if it is useful on any execution. If dynamic data-flow graph analysis is not performed, *Sometimes-useful* is assumed for all bit-positions.

3.3 How many bits are useful: definitions

Various bit-set analyses are meaningful only in the context of particular *assumptions* made about the hardware. Because of the static-summarizing step, our analyses are only valid under the assumption that the hardware executing a given static instruction remains unchanged for the duration of the program. This is by no means a universal assumption, but a reasonable one.

While the annotations *Always-0*, *Always-1*, and *Sometimes-useful* are convenient for computing the static summary, in interpreting the results it is more convenient to convert to a three-way classification of bits into *N*: never useful; *C*: sometimes useful and constant; and *V*: sometimes useful but not constant.

We offer several definitions that describe which bits are considered *useless* in an operand:

- Definition 1 (bit-wise, optimistic): All *N* and *C* bits. This definition reflects the savings possible in configurable hardware. In addition to eliminating all reference to outputs marked *N*, a circuit can be synthesized to provide *C* outputs as hard-wired 0 or 1. *C* inputs can be used directly to simplify the circuit description. (It is instructive to consider the interaction of the savings inferred in program analysis, and those in subsequent circuit synthesis [4].)
- Definition 2 (bit-wise, less optimistic): All *N* bits. If joint synthesis of circuit implementations corresponding to larger data-flow subgraphs is not possible, then wires carrying 0s and 1s, though known constants, are necessary, and this definition models that cost.

– Definition 3 (prefix bit-wise, optimistic): All leading *N* or *C* bits. This definition reflects the fact that customizing circuits to take advantage of isolated known-constant bits may not be as practical as generating a family of circuits for any operand width. (Particularly true when we consider narrow functional units for inclusion in a non-configurable processor.) Thus, only high-order bits are considered useless, leaving a contiguous string of useful low-order bits.
– Definition 4 (prefix bit-wise, less optimistic): All leading *N* bits. This definition is conservative both in the sense of definition 2 and of definition 3.
– Definition 5 (data range): All leading sign bits but one. This definition assumes that hardware circuits for widening (with sign extension) are inserted where necessary. In other words, useful bits are those needed for the 2's complement representation of the observed data range, exactly as in [8], and analogous to statically inferred data ranges in [4].

The definitions described which bits were considered useless; all *remaining* bits are useful, and are counted towards the totals we report in the results.

4 Experimental Setup

Our initial focus was on the Raw benchmark suite, in order to compare the results with those of [4]. We also examined a number of longer-running benchmarks, including SPEC95 (both integer and floating-point) and Honeywell ACS [10] suites. We summarize the relevant program characteristics in Table 4. The columns for integer operands refer to the integer values identified and analyzed by our tools.

Benchmarks were compiled on a Compaq (Digital) Alpha 21164 EV56 machine using native C and Fortran optimizing compilers. Each benchmark was simulated using our run-time analysis extensions to SimpleScalar/Alpha. The lookahead size was limited by the memory capacity of the simulator host machine and the memory overhead of the simulator to about 1.5 million instructions. This was sufficient to encompass entire executions of all but one Raw benchmark.

Results are reported for all integer operands. Whereas a bit-wise analysis can infer that certain bits are constant in floating-point operands, this is more difficult to exploit, and is generally eschewed for configurable hardware.

5 Results

Average bit requirements are shown in Figure 1, for the unweighted average, and in Figure 2 for the average weighted by each instruction's execution count. For each benchmark, seven bars are shown with average bit requirements computed according to different analysis modes. In order from the bottom upward in the graph: *def1/bp:* definition 1 of Section 3.3, with dynamic analysis ("back-propagation") as described in Section 3.2; *def2/bp:* definition 2 with dynamic analysis; *def3/bp:* definition 3 with dynamic analysis; *def4/bp:* definition 4 with dynamic analysis; *def1:* definition 1 without dynamic analysis; *def3:* definition 3 without dynamic analysis; *def5:* definition 5 (which does not involve dynamic analysis).

Benchmark	Description	Source lines	Instructions executed	Integer operands (static)	Integer operands (dynamic)
Raw					
adpcm	Multimedia: audio compression	195	288672	1309	584829
bubblesort	Dense matrix	62	2993603	1211	7015682
convolve	Multimedia	74	34248	1383	71967
edge-detect	Multimedia	175	151124	1875	289721
histogram	Multimedia	115	862026	1416	2040775
intfir	Multimedia: Integer FIR filter	64	381020	1160	855144
intmatmul	Dense matrix: matrix multiplication	78	602363	1218	1472609
jacobi	Dense matrix	84	30652	1166	68270
life	Automata: Conway's game of life	150	1460919	1716	3552042
median	Multimedia: median filter	86	745383	1308	1347242
mpegcorr	Multimedia: kernel from MPEG-3	144	25958	1383	63115
newlife	Automata: Conway's game of life	119	736575	1627	1890671
parity	Multimedia	54	313236	1128	751227
pmatch	Multimedia: pattern matching	63	1403699	1178	3263404
sha	Encryption: secure hash algorithm	638	1030098	6572	3007709
sor	Dense matrix: Successive overrelaxation	60	1026951	1300	2516264
Honeywell					
microkernel	Two-dimensional discrete cosine tranform	169	248322172	7606	412699970
timing	CORDIC vector rotation algorithm	219	58295213	7627	104040721
versatility.compress	Wavelet image compression algorithm	528	56202163	8343	113693092
SPECint95 (train inputs)					
126.gcc-jump	GNU C compiler (`jump.i` input)	133049	202205526	150516	268594145
129.compress	Adaptive Lempel-Ziv coding	1422	46186413	6069	83526455
130.li	LISP interpreter running the Gabriel benchmarks	4323	192134942	9343	251585142

Table 1. Properties of benchmarks.

To make the chart legible, we included only three of the 16 Raw benchmarks; the ones not shown behave similarly to *sor*.

The kernels from Raw overall have very low bit requirements by any measure, with the exception of *sha* (which by design constructs numbers spread wide over the integer range). Larger programs from the Honeywell and SPEC95 suites tend to have higher bit requirements. These average bit requirements numbers could, in principle, be compared against those reported from static analyses, as in *Bitwise* and *BitValue*, with the expectation that our run-time analysis represents the limit to which the compile-time analysis may aspire; however, differences in the context of analysis preclude a *direct* numerical comparison.

In addition to averages, the distribution of bit requirements is of interest. For two characteristic programs, the *life* kernel from the Raw suite, and the *gcc* compiler, it is shown in Figures 3 and 4. We divide possible bit requirements into bins for 0, 1, 2, 3–4, 5–8, 9–16, 17–32, and 33–64 bits, and display them as a stacked bar graph. Each of the seven analysis modes results in a different bar; and again the operands may be counted (a) statically or (b) dynamically.[1]

In the average bit requirement plots, as well as in the detailed distributions, we note that models *def1/bp* and *def3/bp* produce nearly identical results; and similarly models *def2/bp*

[1]Drawing the seven exact distribution curves is less informative visually, because some of them are too close to one another to be discernible.

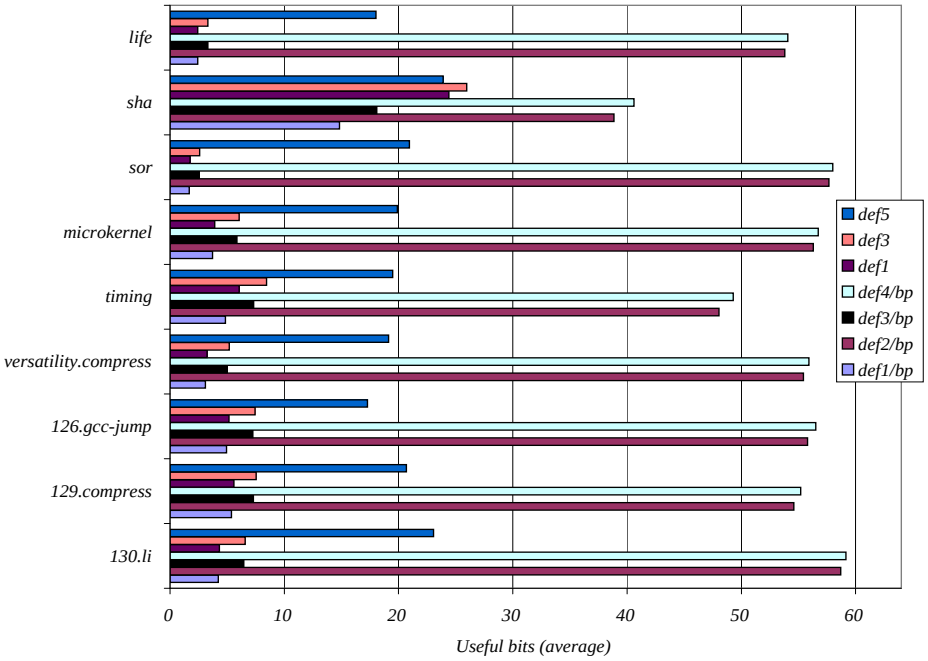


Fig. 1. Average useful bits, static operand count

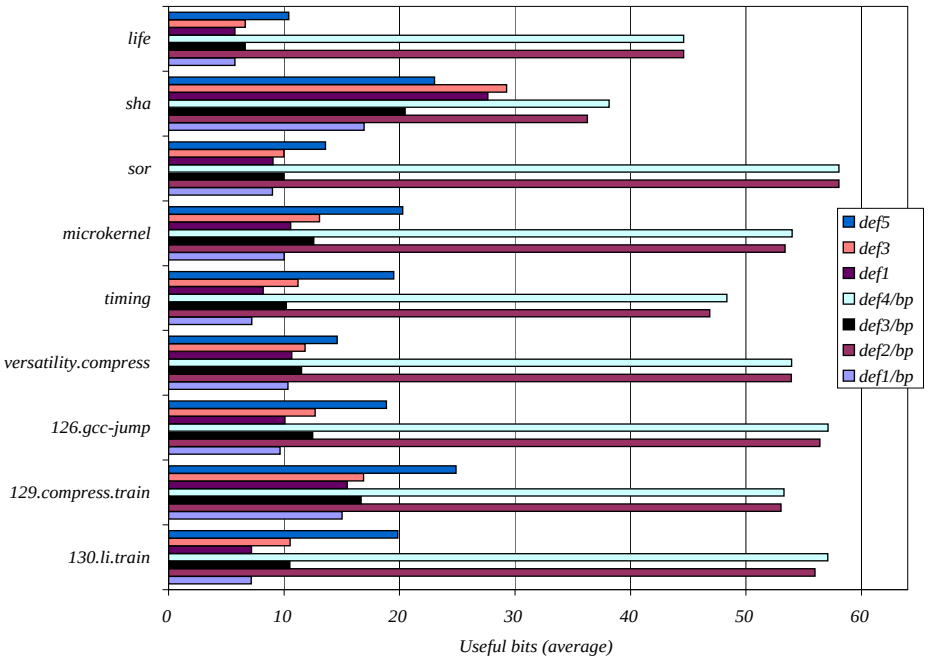


Fig. 2. Average useful bits, dynamic operand count

and *def4/bp*; and models *def1* and *def3*. Thus, the restriction of useless bits to the high-order prefix is of little significance.

Comparing *def1/bp* with *def1*, or *def3/bp* with *def3*, we see that the additional savings that backward-propagated operand-use information gives are very small. This becomes quite clear when we compare *def1/bp* with *def2/bp*, or *def3/bp* with *def4/bp*, where we see that using constantness information makes a huge difference; models *def2/bp* and *def4/bp* make use *only* of the backward-propagated operand-use information, and this is scant.

Lastly, definition 5 is typically weaker than either 1 or 3, i.e., it considers more bits as useful. In light of the fact that greater savings are reported [4] from a static analysis using definition 5 than from another static analysis [5] using definition 3, we draw the conclusion that the difference in savings is not due to the use of differing definitions, but either to a more powerful compile-time analyses preceding the analysis of bit-sets, or to significantly different operand accounting.

To summarize, the number of bits that can be saved other than in the high-order prefix is small, by any definition. Analysis using backward flow over the dynamically reconstructed data-flow graph is not very powerful in inferring that bit-positions are not needed. The constantness information from observed operand values is much more informative.[2]

Acknowledgments. We would like to thank Jonathan Babb for valuable discussions on the design of *Bitwise* as well as for allowing us the use of the Raw benchmark suite. We thank David Brooks, Zhen Luo, and the anonymous reviewers for their comments.

References

[1] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, Cambridge, Massachusetts, 1994.

[2] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Micro-27*, Nov. 1994.

[3] R. Razdan, K. Brace, and M. D. Smith. PRISC software acceleration techniques. In *Proc. Int'l Conf. on Computer Design*, pages 145–149, Oct. 1994.

[4] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *PLDI 2000*, Vancouver, BC, June 2000.

[5] M. Budiu, S. C. Goldstein, M. Sakr, and K. Walker. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *EuroPar 2000*, Munich, Germany, 2000.

[6] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, Dec. 1996.

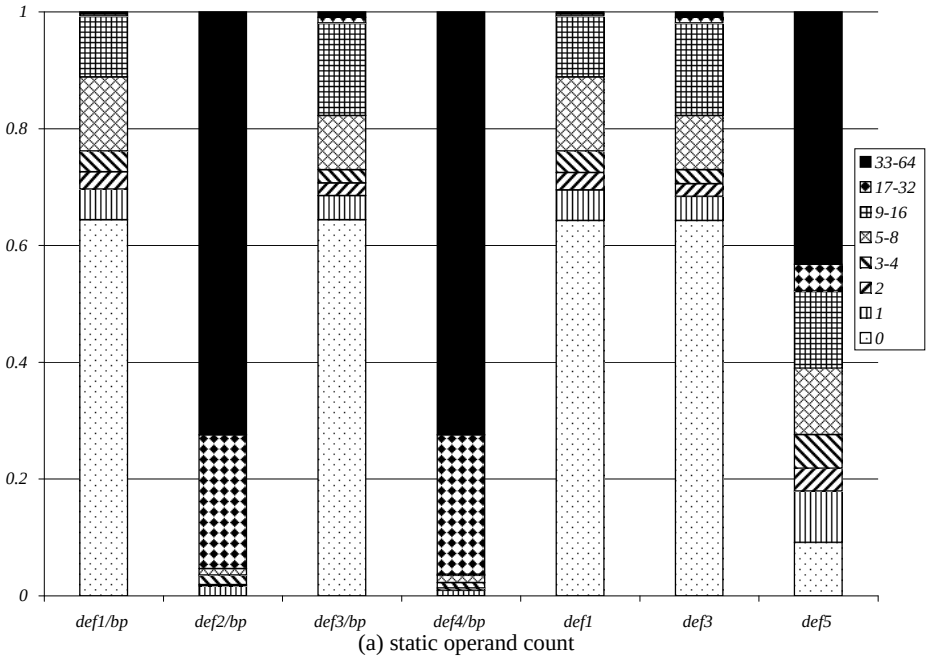
[7] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *FCCM '99*, Napa Valley, CA, Apr. 1999.

[8] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *5th HPCA*, Jan. 1999.

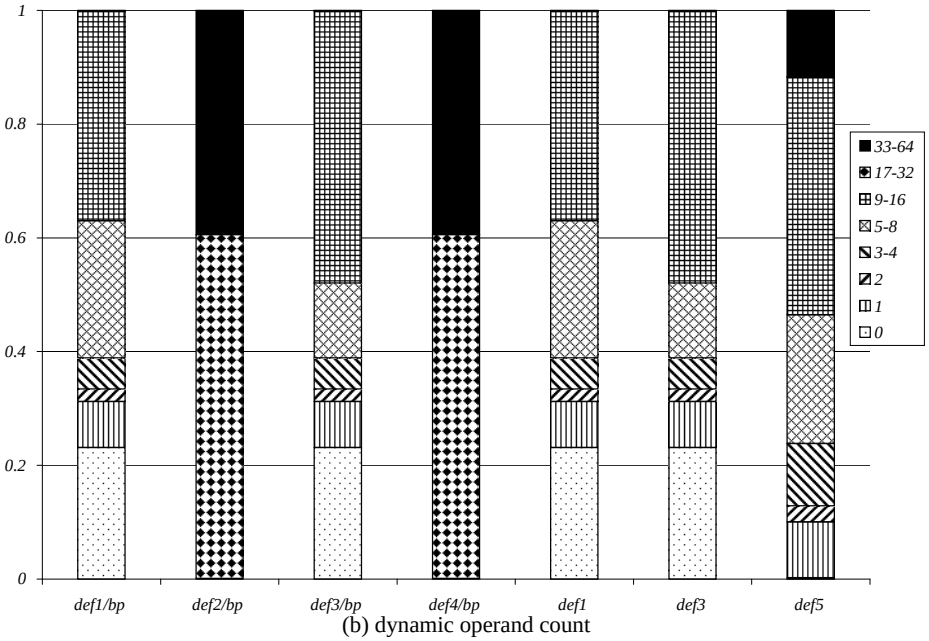
[9] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, pages 13–25, June 1997.

[10] S. Kumar. Benchmarking tools and assessment environment for configurable computing. Submitted by Honeywell Technology Center to USA Intelligence Center and Fort Huachuca under Contract No. DABT63-96-C-0085, Sept. 1999.

[2] The corresponding compile-time backward analysis may be *relatively* more powerful, because the static inference of operand values is weaker.

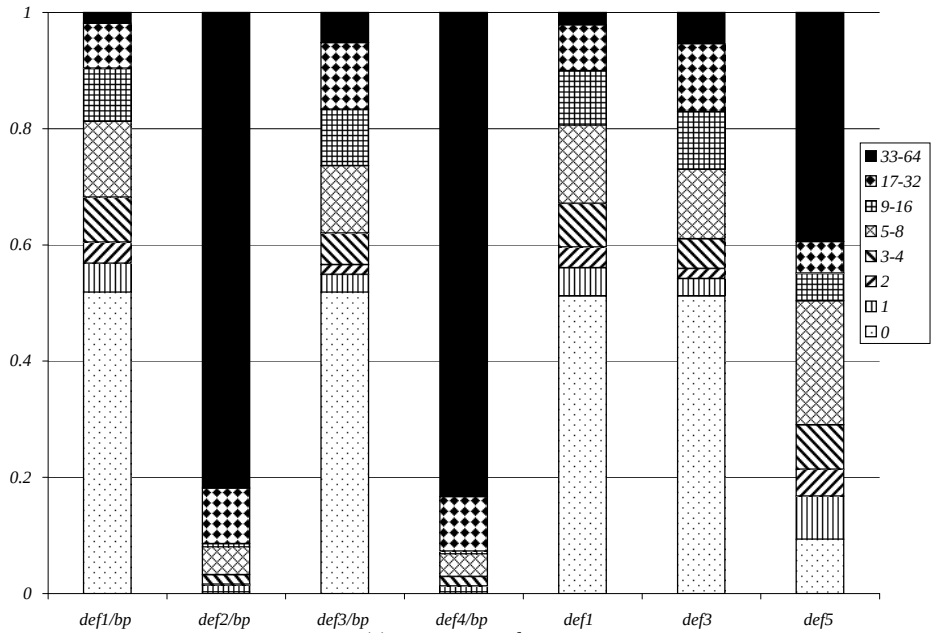


(a) static operand count

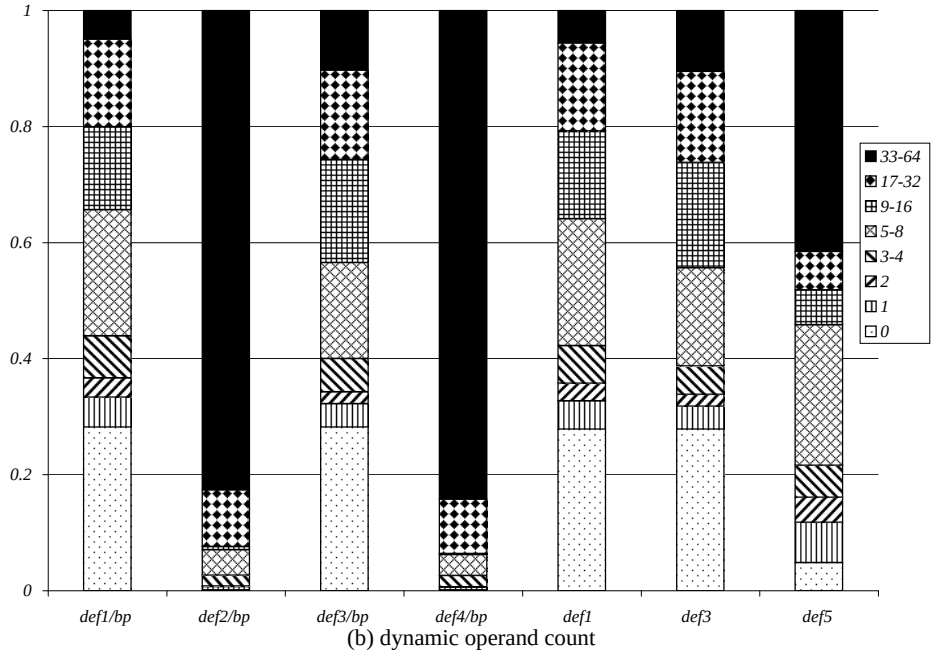


(b) dynamic operand count

Fig. 3. Benchmark *life*: bit requirements distribution



(a) static operand count



(b) dynamic operand count

Fig. 4. Benchmark *126.gcc-jump*: bit requirements distribution