PAMOS and PAROS
Parallel Addition of Multiple or Redundant Operands in a Single Word

Klaus Echtle
Institute for Computer Science and Business Information Systems
Dependability of Computing Systems, University of Duisburg-Essen
echtle@dc.uni-due.de

Bernhard Fechner, Jörg Keller
FernUniversität in Hagen, Department of Mathematics and Computer Science
Parallel Computing and VLSI Group, 58084 Hagen, Germany
{Bernhard.Fechner, Joerg.Keller}@fernuni-hagen.de

ABSTRACT: We present two novel addition techniques called PAMOS (Parallel addition of multiple operands in a single word) and PAROS (Parallel addition of redundant operands in a single word). PAMOS increases the throughput of a single n-bit adder by carrying out m parallel additions of p bits each. PAROS enhances reliability by using m identical operands for a redundant computation. Both mechanisms are based on the fact that modern microprocessors distinguish different operand bit lengths p (which are typically, but not necessarily powers of two). For $p < n$ the leading $n – p$ bits of an n-bit adder are not used at all. This space can either be used to carry out multiple additions with different operands or multiple additions with the same operands. The difference to common SIMD extensions like MMX or SSE is that resources are allocated dynamically and that no extensions to the instruction set of the processor must be implemented. Furthermore, PAMOS and PAROS can be seen completely decoupled from underlying hardware resources.
The first experimental results reveal that with an operand bit width of $p = 4$, at most 16 additions can be carried out in parallel by a 64-bit adder, and 2.7 in average. Another experiment performed by a static code analysis of SPECint2006_base benchmarks gave an average speedup factor of 3.19, showing the potential of PAMOS and PAROS.

1 INTRODUCTION

Modern microprocessors distinguish different operand bit sizes p as early as in the decode stage. Usually these have sizes in a power of two. Hence $n – p$ bits of an n-bit operation unit are not used at all. This paper shows how to exploit this space to execute additions in parallel or redundantly.
Our first technique, called PAMOS (Parallel addition of multiple operands in a single word), is able to concurrently execute multiple operand streams on a single adder, just like in common MMX or SSE instruction set extensions [23]. The difference is that PAMOS has the potential of dynamic resource allocation. Extensions to the instruction are not needed, thus keeping code compatibility. Typical resource conflicts [12] (e.g. when two or more threads want to access a resource simultaneously) concerning arithmetic units in simultaneous multithreaded (SMT) systems [11][22] can be reduced, since the bandwidth of the concerned resources is increased. Note, that PAMOS refers to the concurrent execution of multiple operand streams on a single addition unit and not the microarchitectural approach in [14], coining the term *multiscalar processors*. The second technique, called PAROS (Parallel addition of redundant operands in a single word) is based on PAMOS. Here redundant operands are scheduled onto an adder. The result is a mixture between temporal and dynamic structural redundancy. By comparison of the results we are able to detect faults. In our fault model we consider both transient bit-flips and permanent stuck-at faults of a limited number of bits.

The paper is organized as follows. Section 2 presents related work. Section 3 depicts our first approach by regarding arbitrary bit sizes. Section 4 describes the second approach by regarding fixed operand bit sizes. Section 5 presents experimental results; Section 6 summarizes and concludes the paper.

2 RELATED WORK

The number of different summation techniques developed over the years is nearly countless. Many of these are based on variations of the parallel prefix principle. Ladner and Fischer [10] showed that the outputs of each finite, determined automaton can be computed simultaneously with techniques based on the solution of the parallel prefix sum problem. In their work, Brent and Kung [2] presented the first representation of a parallel prefix adder working in $O(2\log_2 n-1)$ time that could be visualized by using black and gray processors. The carry-skip principle [3][5][6] can also by applied by propagating a precomputed carry over a number of blocks. By introducing multiple levels as in [8], the Multilevel Carry-Skip scheme can be applied. The prefix scheme from Sklansky [7] works in minimal time O(log n). Han-Carlson [4] presented an almost time-optimal prefix adder, reducing the space consumption of the Sklansky adder. Tyagi [9] proposed a carry-increment adder-architecture which is similar to the carry-select adder of Bedrij [1].
The topic of fault-tolerant arithmetic is as old as fault-tolerance. A general overview over redundant numbering systems as well as error detection and correction is given in [15]. Error detection and correction can be achieved by duplicating or triplicating hardware [17], respectively. With reduced structural hardware overhead, error detection can also be achieved by time redundancy [18] or recomputation by using shifting operands (RESO)[16]. Further techniques based on RESO include the recomputation with swapped operands [19] or the recomputing with duplication and comparison trying to reduce the hardware complexity and delay. The recomputation with shifted or swapped operands covers both transient and permanent faults.
MMX or SSE instruction set extensions [23] in x86 compatible processors are able to execute multiple operand streams on a single command and thus in a SIMD fashion. The instruction itself specifies the size of the operands and the operation.

PAMOS and PAROS can be implemented in different variants, as presented in Sections 3 and 4. In some variants we have extended previous work [24] by introducing control bits for each register, indicating different types of notation, redundant or normal. The different types allow – besides the check after an addition – the checking of registers for equality at any point in time which may be adequate, e.g. at a latency through a memory operation. Furthermore, we regard things from a completely different point of view, mainly the compiler.

3 FIRST APPROACH: ARBITRARY BIT WIDTHS

The key idea in this work is to get the upper (temporarily unused) bits of an adder to work. In this point our effort fundamentally differs from techniques to speed up the addition itself, as we leave the addition principle untouched. The first thoughts in this section handle a general case, where we do not know the actual length of the operands. Thus, we have to determine the last position a carry propagates to.
As our experiments in Section 5 will show the actual number is typically far below the theoretical average value for equiprobable operands. Long chains of leading zeros are frequent!
Remark: When we speak of leading zeroes, we also include leading ones for negative numbers.
As we have the possibility to concurrently execute multiple operand pairs, they must be (optimally) distributed so that we can carry out as many concurrent additions as possible. The space-optimal dynamic scheduling of m operands of different widths onto an n-bit addition unit is a problem similar to the knapsack problem which is NP-complete [13]. To work around this, i.e. to omit the scheduling of operands, we suggest a simple differentiation according to the bit width of the operands, deducted from the fact that e.g. in x86 processors we already have such a distinction starting from the decode stage, since it is inbuilt with the type of register we use (i.e. al, ah, ax, eax, rax). The techniques presented in Section 4 are directly based on this fact. As the possibilities to distribute the workload are limited (as are the operand bit widths), the pre-calculated scheduling of operands onto the adder can be e.g. stored in a ROM. In the static code experiments in Section 5, we only distinguish the bit widths induced from the register choices above.

4 SECOND APPROACH: FIXED OPERAND BIT WIDTHS

4.1 *PAMOS*

When we restrict the first approach to a set of fixed bit widths of the operands, we can use simpler techniques to implement PAMOS. This applies for all of the three major steps:

- The *decision* whether operands can be concatenated, this means whether the sum of the bit widths of the operands "information part" is smaller than the adders bit width n.
- The *concatenation* of the operands to be added simultaneously.
- The *separation* of concatenated operands to prevent undesired flow of carry information between them.

For the description of PAMOS and PAROS we use the following notations (Table 1), if not stated otherwise.

Table 1. Notation.

Variable	Remark
n	Bit width of the adder
m	Number of operands to be processed in parallel
p	Number of bits of an operand
a	First operand
b	Second operand
c	The sum, result of an addition
w	Word, which consists of multiple operands
k	Operand index, typical k = 1, ... , m

Figure 1 shows the hardware of an enhanced n-bit adder for the *concatenation* of m operands. For each adder block of bit width p the preceding multiplexers allow for the selection of operands. Note that the multiplexers can also select longer operands of bit width i·p, to be added by i adder bocks. This includes the case where the n-bit adder just adds two n-bit operands.

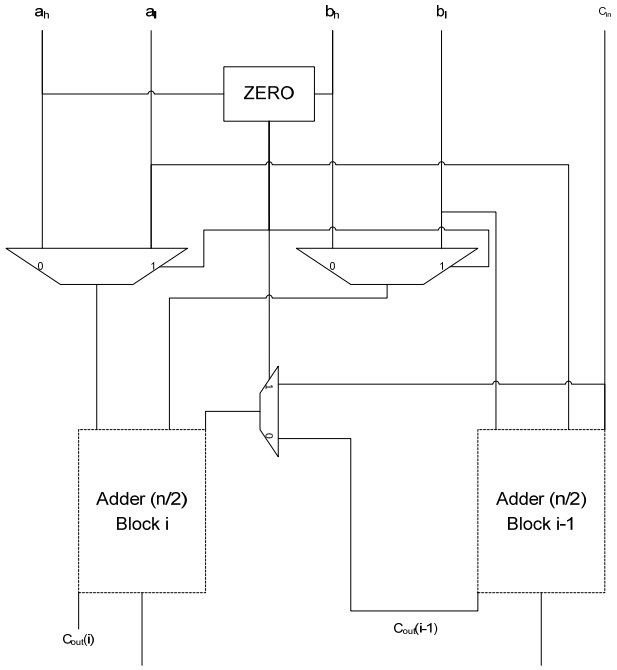

Figure 1. An Enhanced n-Bit Adder.

For the other major steps we present several alternatives.

The *decision* on operand concatenation can be taken as follows.

Alternative 1: (explained for m = 2)

At first, only one pair (a, b) of operands is considered. Their upper halves are called a_h and b_h, the lower halfs a_l and b_l. The ZERO detector (see Figure 1) decides whether the upper halves entirely consist of zeroes. To detect the number of leading zeroes, we can start a binary search from the most significant bit. This work can be done prior to the computation, e.g. in a pipelined manner. In case the upper halves don't contain even a single 1-bit, the respective adder block can be used to add a further pair of operands – which must fit into the remaining n/2 adder bits, of course (to be checked by the same method).

Alternative 2:

The inquiry for leading zeroes is not necessary, when the operand bit width is known in advance (from the operands' register bit width, for example).

Alternative 3:

Control bits can be introduced for each operand, signaling, whether an operand is in normal or redundant representation (see section 4.2). We extend each operand by a control bit $s_a \in \{normal, redundant\}$. For two operands, a, b, we have four cases, one of which ($s_a = s_b = redundant$) is presented in the next Section.

1. If $s_a = s_b = normal$, an n-bit addition is performed.
2. If $s_a = redundant$ and $s_b = normal$, the first operand must be normalized by setting the upper n/2 bits to zero (needs two gate delays). A normal addition will be performed then.
3. The case $s_a = normal$ and $s_b = redundant$ is symmetric to the previous case (exchanged operands).

The mechanism requires that all registers are equipped with a control bit. The control bit will be calculated on a load. If the result from an arithmetic operation is written to the register, we already calculate the control bit. If not, we must advance just like in the case of a load.

The third major step is the *separation* of concatenated operands to prevent undesired flow of carry information. Two alternative approaches can be taken.

Alternative 1:

A multiplexer may either forward or cut all carry information between the adder blocks assigned to different pairs of operands. In case of look-ahead techniques multiple lines are affected.

Alternative 2:

At least one separating 0-bit is inserted between each pair of adjacent operand pairs, to guarantee that carry will not propagate further. Each separating zero means wasting an adder bit, of course.

PAMOS also works with negative numbers in two's complement representation. However, the processor's negative flag is only affected by the most significant bit. For other results either extra flags or shift left operations must be implemented. In case of separating zeroes 1-bits in the gap between adjacent operands must be deleted after an addition.

4.2 *PAROS*

PAMOS contributes to fault detection when identical replicas of the operands are added in parallel. In this case we speak of PAROS, the parallel addition of redundant operands in a single word. This approach requires the following steps (P1 to P6) for processing two integer addends a and b:

P1. It must be checked whether a and b exhibit a sufficient number of leading zeroes, such that m redundant replicas $a_1, \ldots, a_m$ and $b_1, \ldots b_m$, respectively, fit into a single adder-word. For fault detection purposes duplication (m = 2) is mostly sufficient, which means n/2 leading zeroes at least.

P2. The operands a and b have to be replicated into m copies each. The replicas are then concatenated within a machine word $w_a = a_m \cdot 2^{(m-1)\cdot k-1} + \ldots + a_1 \cdot 2^{(1-1)\cdot k}$ as well as in a machine word $w_b = a_m \cdot 2^{(m-1)\cdot k-1} + \ldots + a_1 \cdot 2^{(1-1)\cdot k}$, where $k = \lfloor n/m \rfloor$. The words w_a and w_b consist of n bits each.

P3. In the adder, the flow of carry information between the replica boundaries has to be inhibited.

P4. The addition of all replicas is performed simultaneously by an ordinary n-bit adder. In the result is the sum $w_c = c_m \cdot 2^{(m-1)\cdot k-1} + \ldots + c_1 \cdot 2^{(1-1)\cdot k}$ where $c_1, \ldots, c_m$ are replicas of the sum a + b.

P5. The replicas $c_1, \ldots, c_m$ have to be compared to each other. Duplication (m = 2) allows fault detection by checking if $c_1 = c_2$. Higher degrees of redundancy ($m \geq 3$) allow even fault masking by majority voting.

P6. In case of fault detection appropriate exception handling is required. Otherwise a replica of the sum, say c_1, has to be selected as the final result c.

All of these steps can be performed by hardware or by software. In principle we can think of an implementation (cf. Figure 1) as follows:

I1. Special hardware may count the number of leading zeroes. Instead, software may check $0 \leq a \leq 2^k$ and $0 \leq b \leq 2^k$, where $k = \lfloor n/m \rfloor$.

I2. Multiplexers may perform the replication of the addends a and b. Instead the words w_a and w_b may be formed by software, preferably by m shifts to the left of k bits each, and by or-connecting all the results of the shifts.

I3. If extra hardware has access to the internals of the adder, the flow of carry information can be controlled. Both hardware and software can suppress the flow of carry information by the same approach: Between adjacent redundant replicas

at least one separating bit with the value zero is inserted.

I4. The addition is always done in hardware, of course.

I5. Extra hardware may compare the replicas c_1, ... , c_m of the sum, and also take a majority decision in case $m \geq 3$. Software may perform these operations as well. However, comparisons may use the hardware in a similar way as the addition does. Consequently, undetectable coincident faults are not necessarily excluded.

I6. The selection of a final result is a masking operation, which can be equally done in hardware or software.

If the comparison (see I5) is performed in hardware according to Figure 1 the adder itself does not differ between the parallel addition of different operands (PAMOS) and the parallel addition of redundant replicas of a pair of operands (PAROS). In the presence of control bits it decides as follows.

1. If $s_a = s_b =$ *redundant*, the mux in the middle of the adder will be switched to carry-in (c_{in}) and both additions can be carried out in parallel.
   - If $c_{out} = 0$, the result is in redundant notation. The zero and negative flags are correct, overflow must be set to zero.
   - If $c_{out} = 1$, the bit at position n/2 must be set to one. The zero, negative, overflow and the carry flags as well as the bit positions n/2 + 1 through n - 1 and the s-bit are set to zero, since the result is in normal notation.

   Regarding an implementation within a pipelined processor, we can execute the redundant computation in the following clock cycle (in an extra stage within the arithmetic pipeline) or in the current cycle, if the adder is not in the critical path. The normalizing of the result can be done in the successing cycle. The throughput of the adder is 1, since in each cycle an addition can be started.
2. For the remaining cases we carry out the additions in two consecutive clock cycles. The comparison is done the following cycle. Parallel to the comparison, a test to a possible conversion to redundant notation can be done. The throughput is lowered to 1/2, since the adder is blocked for two clock cycles.

Alternatively, we can frequently check the registers containing redundant notations for equality, and do a test for a possible conversion from normal to redundant notation. The definition of the timing interval is dependent from the performance and energy constraints of the final system.

4.3 *PAROS seen from the Compiler*

In principle, PAROS can be seen from both hardware and software implementations. Obviously the software solution costs a couple of additional instructions to be executed, thus causing a significant slow-down. On the other hand, hardware has its cost when implemented as an add-on to an existing design.

In both cases the overhead (whether runtime or structural) can be reduced, when we are able to eliminate some of the steps (I1-I6) defined just before. In the following we present an approach suitable for both hardware and software implementations. It is based on two ideas:

- Some of the checks can be done before runtime. When an integer variable is known to have a limited range of values, the permanent existence of a number of leading zeroes can be concluded. Such guarantee is even helpful when the limited range lasts only for some short period, like the execution of a loop. In many cases knowledge in this direction is available at compile time (it is also available at program design time, of course, but we do not want to put this burden to the programmer). Whenever, the value of an integer variable is known to be sufficiently small an explicit check on leading zeroes (I1) can be skipped. Similarly, the potential check for a gap for separating zeroes (part of I1 and I3) can be omitted.
- The second idea is restricted to PAROS. When some variable is subject to repeated additions, it need not be replicated before each addition (I2) and mapped to a single variable value thereafter (I6). Instead, replication before the first and selection after the last addition is sufficient. Besides saving cost and / or runtime this approach has a further substantial benefit contributing to fault detection as well: The values are stored in their redundant representation. Moreover, on their transfer to and from the ALU they are redundant as well. Hence, any data corruption limited to a number of bits at any point in time during the complete “redundant interval” can be detected by the very tests already implemented for checking the correctness of the adder (I5).

A practical question remains: How frequent are these "lucky cases"? In the worst case too few leading zeros may diminish the performance advantage of PAROS. In the second worst case, it can only be applied for a single addition. In the best case the "redundant intervals" last for many operations. The distribution among these cases is highly program-dependent. See Section 5 for theoretical and empirical data.

4.4 *Software Examples*

There are many cases where the limited range of values is visible at compile-time, as is illustrated by the following examples:

- One may use a (signed) integer variable (with 32 bits) to encode system states, where the only operations are assignments ("enter some state") and increments ("proceed to next state" or "proceed to next but one state" etc.). The number of states may be small (100, for example, which means always 24 leading zeros).
- Cardinal numbers may be used purely for indexing the elements of an array. If the size of the array is known, a guaranteed number of leading zeroes can be concluded thereof.
- Numbers may represent data from sensors which range of values is known (could be an assertion to the compiler). Again, this may save many I1- and I6-operations.

A very simple example for m = 2 should demonstrate the benefits of PAROS, but also its limitations. Assume an array a, which elements should be sorted to ascending order by a naive approach, written in Java:

```
int a[] = new int[90];
for (int i = 0; i < 89; i++)
  for (int j = i + 1; j < 90; j++)
  if (a[j] > a[i])
  { int h = a[i]; a[i] = a[j]; a[j] = h;
  }
```

When this piece of program is compiled it can be transformed as follows. For better readability we characterize the object code by using special symbols:

x_p	means the p bit representation of value x.
$-x_p$	means the p bit two's complement of value x.
$(x_p:y_q:z_r)$	means the concatenation of x, y and z with total length of p + q + r bits.

The integers i and j in the given sorting program can be kept redundant throughout the execution of the two nested loops.

Outer loop:

initial assignment:	$i = (0_{15} : 0_2 : 0_{15})$
comparison to zero:	$i + (-89_{15} : 0_2 : -89_{15})$
increment:	$i = i + (1_{15} : 0_2 : 1_{15})$

Inner loop:

initial assignment:	$j = i + (1_{15} : 0_2 : 1_{15})$
comparison to zero:	$j + (-90_{15} : 0_2 : -90_{15})$
increment:	$j = j + (1_{15} : 0_2 : 1_{15})$

For the indexing of array elements by i and j, addresses have to be calculated which are likely to exceed 2^{15}. Therefore, additional variables x and y have to be created which take the values of i and j in a non-redundant way on each iteration of the inner loop.

assignment:	$x = i \text{ and } (0_{17} : -1_{15})$
assignment:	$y = j \text{ and } (0_{17} : -1_{15})$
check:	$x == i \text{ shiftright } 17$
check:	$y == j \text{ shiftright } 17$

Note that the operations in outer and the inner loop cost identical runtime whether implemented in hardware or in software. The assignments to x and y as well as the two subsequent checks are faster when implemented in hardware. The temporary variables x and y are used in the address calculation to access a[x] and a[y] for comparing them, and exchange them if necessary.

A variety of strategies can be implemented in a compiler (or pre-compiler) to apply PAROS. At present, we have studied the usage of array-indexing variables, as was depicted in the example above. Before proceeding to further strategies we will evaluate the statistics on the usage of integer variables (see Section 5.3).

4.5 *Combination of short and long operands*

In many cases a "short operand" a (with zeroes only in its upper half a_h) is added to a "long operand" b (with few leading zeroes). Typical examples are the additions of offsets to base addresses (as was the case for the access to array a in subsection 4.4. In this case standard PAROS cannot be applied due to the lack a free adder block.

By a dynamic check for zeroes in the leftmost bit of a_l and the leftmost bit in b_l we can see whether or not

carry information is flowing to the upper half of the long operand. If this is the case, the addition a + b must be executed non-redundantly.
Otherwise, however, we can replace the higher half b_h by the lower half b_l. Accordingly the lower half of operand a is doubled. Thus, the adder forms the sum $(b_l, b_l) + (a_l, a_l) = (c_l, c_l)$. Since the absence of a carry between the halves of c_l is guaranteed, we can simply replace the higher half of the result by b_h to obtain the correct result (b_h, c_l).
The disadvantage of this approach is its limitation to the cases where specific bits are zero. This is the case for approximately half of the operands.

4.6 *Recovery*

What about recovery after error detection? Again, we distinguish some alternatives:

- For operands in normal representation, the addition can be carried out a third time, followed by majority voting over all three sums. In this case, all additions succeeding the fault must be cancelled.
- In case of two redundant operands, either the operation can be immediately repeated after the comparison or, if comparisons are directly done on registers, a successful check will form a checkpoint. In case of a fault-revealing test, a rollback to the last known sane checkpoint must be done. This is far more complex, but offers support for other fault-tolerance mechanisms.

5 EXPERIMENTAL RESULTS

5.1 *Monte-Carlo experiments*

We evaluated the throughput of PAMOS by a Monte-Carlo experiment. $2 \cdot 10^6$ equally distributed random binary numbers of lengths {4, 8, 12, 16, 20, 24, 28, 32} were generated, serving as operands for the addition. We computed the number of leading zeroes and determined how many operands can be scheduled onto a 32 bit addition unit. Figure 2 shows the results for a window size of four 32 bit operand pairs. The window size determines the number of waiting arithmetic operations for each arithmetic unit and thus the maximal number of concurrent summations. Figure 2 shows the average number of additions per clock (APC, left y-axis) and the peak APC (right y-axis), supposing an addition in a single clock cycle.

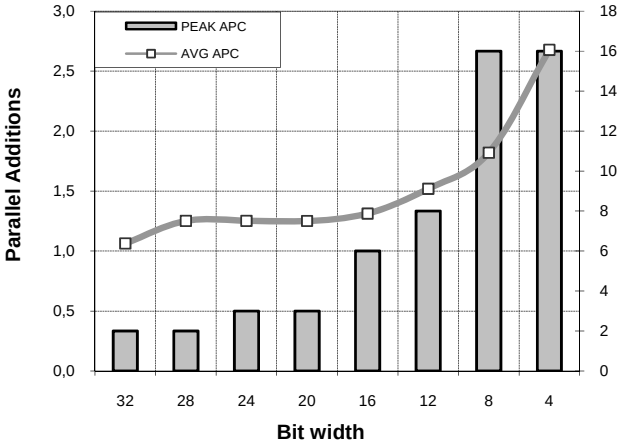

Figure 2. Speedup for binary numbers of length {4, 8, 12, 16, 20, 24, 28, and 32}, window size 4.

With an operand bit width of 4, maximal 16 additions can be carried out in parallel and 2.7 in average.

5.2 *Livermore Loop 1*

To get a first impression on how the PAMOS and PAROS perform under a certain workload, we analyzed a kernel (no. 1) from the Livermore loops, shown in Figure 3.

```
for (k = 1; k <= 990; ++k) {
/* L1: */
      space1_1.x[k - 1] = spaces_1.q +
      space1_1.y[k - 1]*(spaces_1.r *
      space1_1.z[k + 9] + spaces_1.t*
      space1_1.z[k + 10]);
}
```

Figure 3: Kernel #1 (Livermore loops)

Table 2 shows the results of an operand analysis, including operation, frequency and bit width. We only regarded the topmost zeroes within the operands.

Table 2. Operations and bit widths within the Livermore loop kernel no. 1.

Times	Operation	Operand width
991	Comparison through subtraction	First: 12 bit Second: 15*4 bit, 240*8 bit, 736*12 bit
990	Increment	First: 4 bit Second: 15* 4 bit, 240*8 bit, 735*12 bit.
990	Struct-ref., addition between address and value	Spaces1_1: 32 bit and 8 bit, since 256>struct>16 byte. space1_1: 32 bit and 16 bit, since 3 fields have less than 12000 byte and 4th to 6th entry in struct.
990	Array access	Addition between address (32 bit) and index size: **1st and 2nd array:** 16*4 bit, 240*8 bit, 734*12 bit. **3rd array:** 6*4 bit, 240* 8-bit, 744*12 bit. **4th array:** 5*4 bit, 239*8bit, 745*12 bit.
990	2 additions, unknown size.	32 bit.
Weight	4 bit: 5083/33661≈0.15. 8 bit: 5369/33661≈0.16. 12 bit: 8359/33661≈0.25. 16 bit: 3960/33661≈0.12. 32 bit: 10890/33661≈0.32.	

With the frequencies gained from the analysis, we weighted the probability to which different operands of different bit widths occur and conducted the experiments from Figure 2 again to calculate the possible number of parallel additions - this time for adders of widths between 32 and 64 in steps of 4 bits. Figure 4 shows the results. "-" denotes that the mode is deactivated, "T" the additional inclusion of trailing zeroes, "E" the execution of experimentally weighted data and "R" the redundant execution of operands (PAROS).

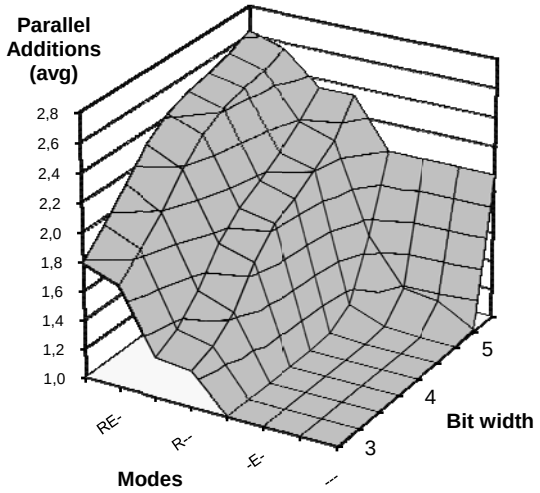

Figure 4. Average speedup for a numeric-intensive application (Livermore loops, kernel #1).

From Figure 4 we see that it is possible to break the boundary of factor 2. How many bits must be added to the underlying adder expecting 32 bit operands to guarantee as many as two redundant additions in average? The question can be is answered by rearranging the results from Figure 4 in a different context. Figure 5 shows the results. Here, only the results for the redundant execution are included.

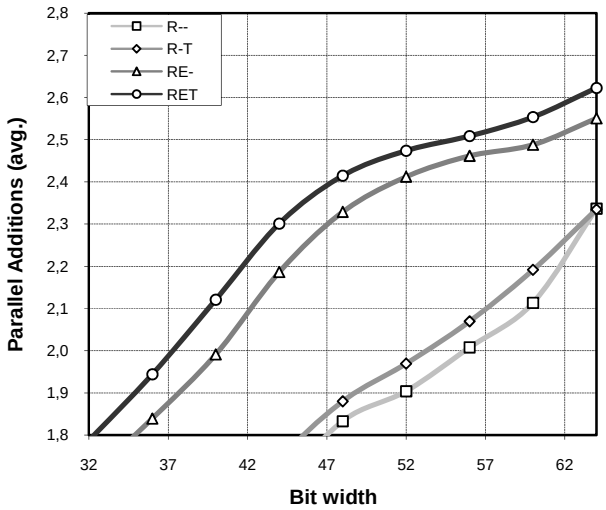

Figure 5. Average parallel additions in redundant modes for a numeric-intensive application (Livermore loops, kernel #1).

From Figure 5 we see that when applying redundant ("R", trailing ("T") and leading zeroes ("-"), non-experimental data "-"), we need 56 bit in average, with experimental data 44 bit (excluding trailing zeroes) and 40 bit including trailing zeroes to carry out two parallel additions in average.

5.3 *Static Code Analysis with valgrind and SPECint2006_base Experiments*

As the results from the Livermore loops were quite promising, we continued experimenting. This time, we ran the SPECint2006_base benchmarks with the simulator valgrind [21], determining the bit width of each operand within the ALU. We distinguish the fixed operand bit widths {1, 8, 16, 32, 64} and ran all benchmarks until they finished.
Figure 6 shows the simulation results (x-axis: Benchmark, y-axis operand bit frequency.

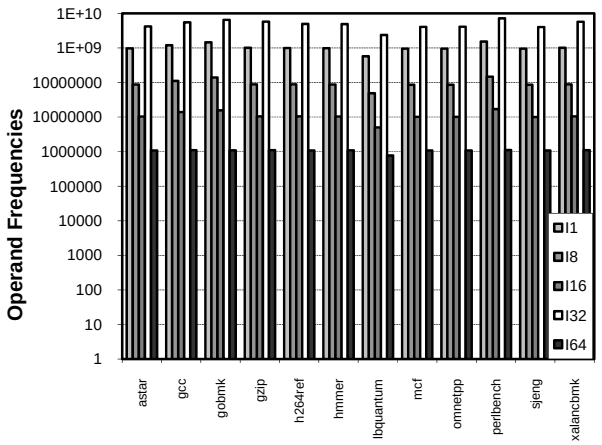

Figure 6. ALU Operand Bit Frequencies.

We counted one-bit operations, since single bit tests (BT) are also part of the instruction stream. We additionally calculated the average percentage of operands in the Load/Store streams (for completeness) and the ALU. The results are shown in Table 3. For the speedup in the last column, we regarded ALU operations only. The speedup is not given as percentage but as factor.

Table 3. Average Percentage of Operands.

Width	Load	Store	ALU	Speedup
1	N/A	N/A	17.2133919	11.0165708
8	10.360407	3.25459459	1.56715165	0.12537213
16	1.36096887	0.35923114	0.1837302	0.00734921
32	88.2777974	96.3823611	81.018251	1.62036502
64	0.00082676	0.00381321	0.01747524	N/A

An average speedup factor of 3.19 is reached. Through reorganization of code, the compiler can take additional care that more cases of additions with two redundant operands occur, e.g. (big + small1) + small2 could be reorganized to (small1 + small2) + big.

6 SUMMARY, CONCLUSION AND OUTLOOK

In this work, a novel design methodology for addition has been presented. The technique enables the execution of multiple or redundant summations (PAMOS or PAROS, respectively) on a single addition unit by leaving the addition principle untouched. PAMOS can be implemented in different variants with higher or lower amounts of extra hardware, or, in an extreme case, by software only. The decisions for the major steps of PAMOS can be taken either during runtime (by inspection of the operands) or statically (by referring to the bit width of the operand's registers.

PAROS performs additions on replicated operands to allow for fault detection. The redundant representation of an operand can be either generated just for a single addition or for a longer path within the program to be executed, thus covering more fault locations. Moreover, when the redundant representation is kept throughout a longer program path, even pure software implementations (or implementation with only a small amount of additional hardware) are efficient.

Monte-Carlo experiments have shown that with an operand bit width of 4, maximal 16 additions can be carried out in parallel and 2.7 in average. By using equally distributed binary numbers, we need 24 additional bits for the adder to carry out two redundant additions in average.

We regarded the results from the analysis once more from a different angle by applying probabilities of bit widths gained from a Livermore benchmark kernel. According to the results, we weighted the distribution and carried out the experiments again. The results show, that in this case we will need only 12 or 8 bit (including trailing zeroes) to carry out two redundant additions in average. Thus, we are able to reduce the additional hardware complexity to only 37.5% (12 additional bits) in comparison to a 100% overhead of structural redundant systems. By a static code analysis of SPECint2006_base benchmarks, we received an average speedup factor of 3.19. PAROS and PAMOS seem to be a promising research area in the future, regarding performance, power consumption, scheduling and reliability aspects. We will continue our work in this direction by doing an implementation in hardware.

REFERENCES

[1] Bedrij, O.J.: *Carry-Select Adder*, IRE Transactions on Electronic Computers, EC-11(3): 340-346, June 1962.

[2] Brent, R. P.; Kung H. T.: *A regular layout for parallel adders*, IEEE Transactions on Computers, C-31(3): 260–264, March 1982.

[3] Chan P. K. et al: *Delay optimization of carry-skip adders and block carry-lookahead adders*, In Proc. 10th Computer Arithmetic Symp., Grenoble, S. 154– 164, June 1991.

[4] Han, T.; Carlson, D. A.: *Fast area-efficient VLSI adders*, In Proc. 8th Computer Arithmetic Symp., Como, S. 49–56, May 1987.

[5] Hobson, R. F.: *Optimal skip-block considerations for regenerative carry- skip adders*, IEEE Journal of Solid-State Circuits, 30(9): 1020–1024, September 1995.

[6] Kantabutra, V.: *Designing optimum one-level carry-skip adders*, IEEE Trans. on Computers, 42(6): 759– 764, June 1993.
[7] Sklansky, J.: *Conditional sum addition logic*, IRE Trans. on Electronic Computers, EC-9(6): 226–231, June 1960.
[8] Turrini, S.: *Optimal Group Distribution in Carry-Skip Adders*, Western Research Laboratory (WRL) Research Report 89/2, 1989.
[9] Tyagi, A.: *A Reduced-Area Scheme for Carry-Select Adders*, IEEE Trans. on Computers, 42(10):1162-1170, October 1993.
[10] Ladner, R. E.; Fischer, M. J.: *Parallel Prefix computation*, Journal of the ACM, 27(4): 831– 838, October 1980.
[11] Joel S. Emer, "Simultaneous Multithreading: Multiplying Alpha Performance," *Microprocessor Forum*, Oct. 1999.
[12] Nathan Tuck and Dean M. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. Proceedings of 12th Intl Conference on Parallel Architectures and Compilation Techniques, September 2003.
[13] Chapman, W. L., Rozenblit, J., and Bahill, A. T. 2001. System design is an NP-complete problem: Correspondence. *Syst. Eng.* 4, 3 (Sep. 2001), 222-229.
[14] Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd Annual international Symposium on Computer Architecture*. ACM Press, New York, NY, 414-425.
[15] B. Parhami. *Computer arithmetic and hardware designs*, Oxford University Press, 2000.
[16] Whitney J. Townsend, Jacob A. Abraham, and Earl E. Swartzlander, Jr. *Quadruple Time Redundancy Adders*, 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 250-256, Cambridge, MA, November 3-5, 2003
[17] D.P. Siewiorek, R.S. Swarz, *Reliable Computer Systems Design and Evaluation*, 3rd ed., A.K. Peters, 1998.
[18] J.H. Patel, L.Y. Fung, Concurrent error detection in ALUs by recomputing with shifted operands, IEEE trans. on Comp., vol. 27, pp. 1093-1098, Dec. 1978.
[19] B.W. Johnson, *Fault-tolerant microprocessor-based systems*, IEEE Micro, vol. 4, pp. 6-21, Dec. 1984.
[20] B.W. Johnson, J.H. Aylor, H.H. Hana, *Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder*, IEEE Journal of Solid-State Circuits, vol. 23, pp. 208-215, Feb. 1988.
[21] Nethercote, N., Seward, J. *Valgrind: a framework for heavyweight dynamic binary instrumentation*. In Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI 2007), S. 89-100, 2007.
[22] Dean Tullsen, Susan Eggers, and Henry Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*, Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995.
[23] T. Podschun. *Das Assembler-Buch . Grundlagen, Einführung und Hochsprachenoptimierung*, ISBN: 978-3827319296, Addison-Wesley, 2002.
[24] B. Fechner, J. Keller. *Efficient Fault-Tolerant Addition by Operand Width Consideration*. In Proc. 6th ARCS Workshop on Dependability and Fault-Tolerance, Delft, NL, March 2009.