A Case Study of Algorithm Implementation in Reconfigurable Hardware and Software

Mark Shand

Digital Equipment Corporation, Systems Research Center
130 Lytton Ave, Palo Alto, CA 94301
shand@acm.org

Abstract. We present a case study of implementation of a combinatorial search problem in both reconfigurable hardware and software. The particular problem is the search for approximate solutions of overconstrained systems of equations over GF(2). The problem is of practical interest in cryptanalysis. We consider the efficient implementation of exhaustive search techniques to find the best solutions of sets of up to 1000 equations over 30 variables. *Best* is defined to be those variable assignments that leave the minimum number of equations unsatisfied.
As we apply various techniques to speed up this computation, we find that the techniques, whether inspired by software or reconfigurable hardware, are applicable to both implementation domains. While reconfigurable hardware offers greater raw compute power than software, new microprocessor with wide datapaths and far higher clock speeds do not lag far behind. Software also benefits from faster compilation times which prove important for some optimizations.

1 Introduction

We present a case study of implementing a combinatorial search problem in both reconfigurable hardware and software. Although we did not produce running hardware, the hardware design was taken to a point where both the circuit size and time required for the search execution could be accurately predicted. The software implementation was carried through to optimized C code running on a Digital 64-bit Alpha processor. The particular problem is the search for approximate solutions of overconstrained systems of equations over GF(2). We consider only brute force approaches to the search. To the best of our knowledge the problem is not amenable to non-brute force approaches. The problem size is held at 1000 equations in 30 variables. Thus the problem state space is of size 2^{30}.

The initial goal of this work was to implement this problem on FPGA-based coprocessors. However, we have found that careful coding on a modern microprocessor with good compiler support can also yield very respectable performance, such that an FPGA-based solution, albeit faster, may not be worth pursuing.

FPGA-based reconfigurable computing machines are often seen as having major advantages over traditional computer architectures on problems involving repetitive computation on small or variable-sized integer data because they can tailor their datapaths to the problem at hand. However, for many years now, techniques have been known that use conventional ALU datapaths to process several small integer variables in parallel, in a SIMD manner ([1], [2]). Certainly these techniques entail some overhead but the speedups gained more than compensate, particularly on modern CPUs with 64-bit wide integer ALUs [3].

We will show that it is useful to consider implementations in both SIMD-style software and reconfigurable hardware in parallel. Techniques that emerge more naturally in one implementation domain can often be fruitfully applied to the other. We will also point out some drawbacks in current reconfigurable hardware systems that limit their performance in this and similar problems.

The paper is organized as follows. Section 2 presents the search problem in mathematical terms. Section 3 describes a basic hardware search engine. Section 4 discusses opportunities for parallelism that speed-up the search with modest amounts of extra circuitry. Section 5 covers the software implementation. Section 6 compares the hardware and software solutions. Section 7 concludes the paper.

2 Linear Systems over GF(2)

Consider a system of M formulae over N variables:

$$r_j(X) = \left(\bigoplus_{i=0}^{N-1} w_{ij} x_i \right) \oplus v_j, \quad 0 \le j < M \tag{1}$$

Arithmetic is performed over the Galois Field GF(2). In less mathematical terms this means values are represented in one bit and all computations are modulo 2. Addition and multiplication are the familiar Boolean operations of *exclusive-or* and *and* respectively.

For a given set of w_{ij}'s and v_j's, if $M > N$ we cannot guarantee that an assignment to each x_i can be found such that each formula in (1) is equal to zero $\forall j : 0 \le j < M$. Nevertheless, we can define a figure of merit, R, which can be used to rank the different possible assignments.

$$R(X) = \sum_{j=0}^{M-1} r_j(X) \tag{2}$$

$R(X)$ counts the number of formulae that do not evaluate to zero for a given assignment to X. We seek those assignments of X that minimize $R(X)$. Note that the arithmetic used to compute $r_j(X)$ is over GF(2) whereas that used to combine the single bit $r_j(X)$'s to form $R(X)$ is over the integers.

3 Basic Hardware Search Engine

In this section, we consider the hardware implementation of search engines that can handle up to 1000 equations ($M \le 1000$) in 30 variables ($N \le 30$). This implies 2^{30} possible assignments of X that must be evaluated. We will consider techniques to find the B best ranked X, where B is a small integer, say 16.

For quantitative evaluation of our proposed implementation, we assume the use of configurable hardware based on Xilinx 3000 series FPGAs [4]. These contain a regular two-dimensional mesh of programmable cells called configurable logic blocks (CLBs) in Xilinx terminology. Each CLB contains two registers and two lookup tables (LUTs) that allow it to implement either two 4-input combinatorial logic functions or one 5-input combinatorial logic functions.

3.1 Principal Constraints

In this section we consider the search order. We find that the enumeration of X should be confined to the outer loop to prevent an excess of intermediate results, and the inner loop should be at least parallelized to the level of the M equations to allow the equation parameters to be wired into the processing elements.

The rank of a particular X, $R(X)$, is the sum of the results of each equation $r_j(X)$. We assume that the computation proceeds by enumerating each possible X and maintaining a sorted list of the best $R(X)$ so far encountered. It is impractical to proceed in a different order, for instance evaluating for successive j, $r_j(X)$ for all possible X since this would require the partially accumulated ranks to be stored (and repeatedly updated) of which there are one billion.

The computation can be performed most efficiently if we have enough hardware to hold locally to each processing element in a distributed store, the w_{ij} coefficients needed for computation of each $r_j(X)$. If enough local store is not available we are likely to be required to reload these coefficients from some central store. The bandwidth required for such reloading would create a bottleneck.

3.2 Gray Codes

In principle, a simple counter could be used to enumerate X. Considerable advantages accrue if instead we use a *Gray code.* A Gray code ensures that only one bit in the representation of X changes for each new X. A Gray-coded counter, G, may be derived from a conventional counter, C, by setting $g_i = c_i \oplus c_{i+1}$.

Suppose that the ith bit of X changes, $x'_i = \neg x_i$. There are two cases to consider:

$$\begin{aligned}
\text{if } w_{ij} = 0 \ \text{ then } & w_{ij}x'_i = w_{ij}x_i = 0 \\
& \Longrightarrow r_j(X') = r_j(X) \\
\text{if } w_{ij} = 1 \ \text{ then } & w_{ij}x'_i = x'_i = \neg x_i = \neg w_{ij}x_i \\
& \Longrightarrow r_j(X') = \neg r_j(X)
\end{aligned}$$

Note that the value of $r_j(X')$ is independent of the value of x_i and depends solely on the value of $r_j(X)$ and of w_{ij}. Thus the processing element that computes the value of $r_j(X')$ needs to know only the value of $r_j(X)$ and which bit i changes in going from X to X'.

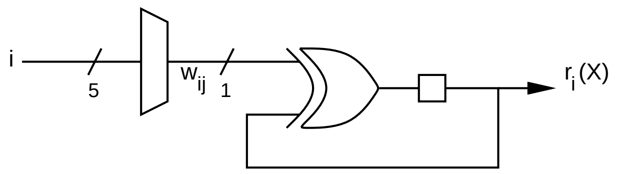

Fig. 1. Formula Evaluation Cell

3.3 Formula Evaluation Cell

The Figure 1 shows the hardware for evaluating successive values for one $r_j(X)$.

The five-bit index i is broadcast from a central Gray-coded enumerator which is shared amongst the evaluation circuits for all formulae. The Xilinx implementation of this circuit requires one and a half CLBs; one for the LUT that decodes i into w_{ij} and a half for an exclusive-or and a register that holds the current value of $r_j(X)$. Parallel evaluation of the 1000 formulae requires 1500 CLBs.

3.4 Population Count

The formula evaluations produce 1000 one-bit values which are added together to yield $R(X)$. The binary representation of this value needs 10 bits. The operation to produce it is a population count. We consider an implementation based on a tree of full-adders. A full-adder takes three values a, b and c of weight 2^k and produces outputs r and s of weights 2^{k+1} and 2^k respectively, where $a+b+c = 2r+s$. A portion of the required full-adder tree is shown in Figure 2.

The population count circuit requires approximately 1000 full-adders. The actual figure will be slightly higher than this due to boundary cases which cause some full-adder inputs to be tied to zero. In a Xilinx implementation this will consume one CLB per full-adder.

3.5 Implementation on DECPeRLe-1

DECPeRLe-1 [5] is an FPGA-based coprocessor built at Digital's Paris Research Laboratory (PRL) in 1992. Its central processing resource is a 4×4 matrix of Xilinx 3090-100 FPGAs containing a total of over 5000 CLBs. The formula evaluation and population count circuitry consumes roughly half of these. The remaining CLBs provide plenty of room to implement the comparators and registers to track the current B best ranked assignments of X.

Given the regularity and the ample opportunities for pipelining, a clock speed of 40MHz is not unreasonable. The proposed design evaluates one assignment of X per cycle and at 40MHz requires 25 seconds to enumerate the 2^{30} states.

We also considered implementation of this computation on the newer FPGA-based coprocessors (TURBOchannel Pamette and PCI Pamette) built by Digital and described in detail in [6] and [7] . These boards have a 2×2 matrix of

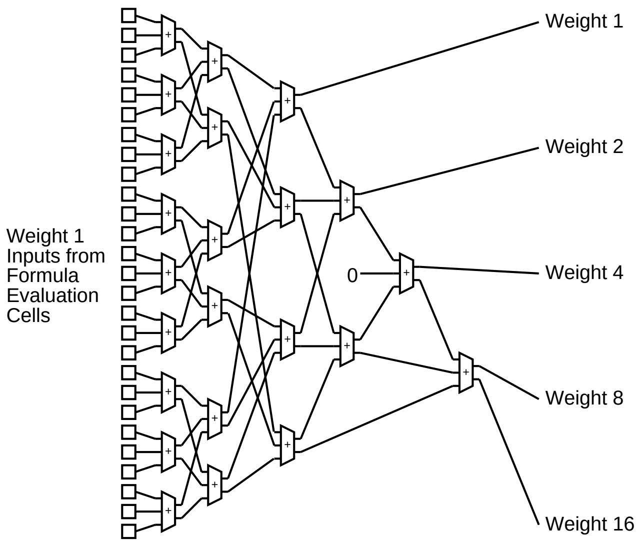

Fig. 2. Population Count Using a Tree of Full-Adders

Xilinx 4010 FPGAs. With only 1600 CLBs, these boards are not large enough to contain the formula evaluation and population count circuitry for the problem size considered.

Surprisingly, we could find no way to use these smaller boards to accelerate the computation over the pure software implementations discussed in section 5. Although the boards provide a powerful computational resource, they can only implement part of the computation and the cost of communicating between them and other parts of the system outweighs any processing advantages they can give.

4 Opportunities for Parallelism

We will consider two classes of methods to parallelize the enumeration of X: methods that make no particular assumptions about the formulae being treated; and methods that depend on w_{ij} and require preprocessing of the formulae. The distinction is important because in case two preprocessing may dominate the time taken for the search, particularly if it changes the wiring to be downloaded into the FPGA-based coprocessor. In the Xilinx technology, logic changes in LUTs can be trivially merged into a precompiled design, whereas wiring changes involve a circuit compilation process that takes, at best, several minutes to run. A software analogue is changing initialized data versus recompiling modified source code. Note also that some of our methods make assumptions about the statistical distribution of w_{ij} coefficients that may not hold in all contexts where solutions to these types of equations are sought.

4.1 Sharing Formula Evaluation Cells

As a first step, let us remove x_0 from the enumeration by computing in parallel the two cases $x_0 = 0$ and $x_0 = 1$. This effectively doubles the number of formulae because we must now evaluate formulae (3) and (4) in each cycle.

$$w_{0j} \cdot 0 \oplus \left(\bigoplus_{i=1}^{N-1} w_{ij} x_i \right) \oplus v_j \tag{3}$$

$$w_{0j} \cdot 1 \oplus \left(\bigoplus_{i=1}^{N-1} w_{ij} x_i \right) \oplus v_j \tag{4}$$

However, observe that (4) is either identical to or the inverse of (3), depending on the value of w_{0j}. This sharing is exploited by the circuit in Figure 3. In the Xilinx technology, the exclusive-or that computes $r_i(X+1)$ may be merged into the logic in the population count for $x_0 = 1$. Thus, this circuit enumerates two states per cycle, double the speed of our previous circuit, at the cost of a second population count circuit, a 1.4 times increase in CLBs.

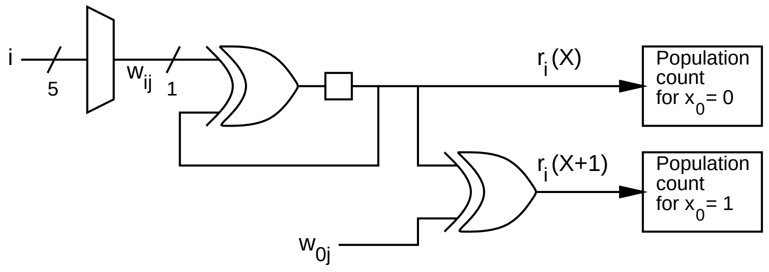

Fig. 3. Sharing Formula Evaluation Cells

We can apply this technique to x_1 and successive variables, but the relative gain decreases since the size of the total circuit quickly becomes dominated by the population count circuits. In any case, evaluating all possible values x_0 and x_1 in a single cycle by this method requires four population count circuits which with the formula evaluation circuit exceeds the CLB resources of DECPeRLe-1.

4.2 Partitioning on w_{ij} Values in Hardware

Much greater savings can be obtained by sharing population count circuits. This is possible if we are permitted to partition the formulae based on the w_{ij} values.

Suppose that in K of the M formulae $w_{0j} = 1$. Without loss of generality, let us sort the formulae so that these correspond to $j : 0 \leq j < K$

$$\begin{aligned} r_j(X \oplus 1) &= w_{0j} \oplus r_j(X) \\ &= \begin{cases} 1 - r_j(X) & \text{if } j < K \\ r_j(X) & \text{if } j \geq K \end{cases} \end{aligned} \tag{5}$$

$$R(X) = \left(\sum_{j=0}^{K-1} r_j(X) \right) + \left(\sum_{j=K}^{M-1} r_j(X) \right) \tag{6}$$

$$R(X \oplus 1) = K - \left(\sum_{j=0}^{K-1} r_j(X) \right) + \left(\sum_{j=K}^{M-1} r_j(X) \right) \tag{7}$$

Equations (6) and (7) show how the bulk of the population count circuitry may be shared. A circuit implementing this sharing appears in Figure 4.

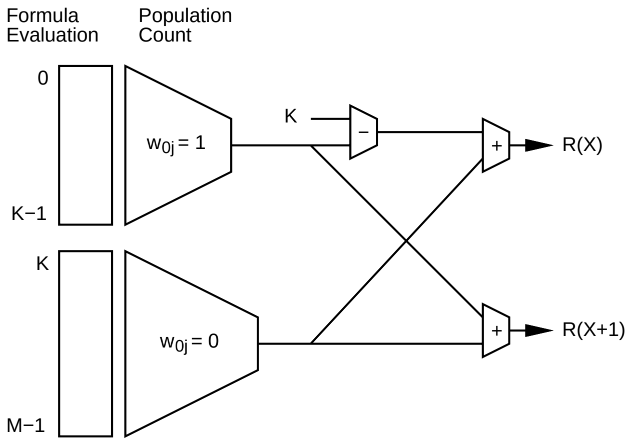

Fig. 4. Sharing Population Count Circuitry

This technique may be applied recursively to w_{1j}, w_{2j}, etc. The overhead of extra adders at the top of the population count tree roughly doubles for each successive application, but remains less than the cost of the population count tree until applied six or seven times. This corresponds to a speed-up of 64 or 128 over our original DECPeRLe-1 execution time estimate of 25 seconds. The chief difficulty posed by the technique is that the preprocessing of the w_{ij} values may require a rewiring of the circuit. The preprocessing overhead is far more manageable in software which is where we have put most implementation effort.

5 Software Implementation

Software implementation has concentrated on Digital's Alpha AXP 21064, with much attention given to maximally exploiting the 64-bit integer datapath of this processor[8]. Measurements on a number of Alpha platforms indicate that the critical parts of the algorithm fit entirely in on-chip caches so the only important parameter in determining speed is CPU clock rate.

5.1 Formula Evaluation

In section 3.1 we argued that in hardware the w_{ij} coefficients needed to be held locally to the processing elements to avoid a performance penalty due to their repeated reloading. Likewise in software these coefficients need to be rapidly loaded to the ALU. The coefficients of 1000 formulae in 30 variables occupy 4 kilobytes and fit comfortably in the 8 kilobyte first level data cache of the 21064.

The coefficients are packed such that adjacent bits share a common i index. A 1000-bit vector represents the current value of the formulae. A Gray-coded counter selects the index of the variable to toggle and the corresponding 1000-bit coefficient vector is exclusive-ored with the current value. This requires 32 loads, 16 xors and 16 stores which can execute in 48 cycles on the 21064.

5.2 Population Count

Population count is designed to exploit the full 64-bit wordlength of the Alpha ALU. Mask and shift operations extract odd and even bits to create 64-bit words containing 32 2-bit fields, the contents of which are either 0 or 1. Three such fields can be added together without danger of overflow into the next field. These adds are performed using the 64-bit register-to-register add, thus 32 fields are processed per instruction. 16 words are required to hold the 1000 1-bit inputs. When these are combined into 2-bit fields, of value 0 to 3, only 11 words are required since only 334 ($\lceil 1000/3 \rceil$) such fields are needed. A new sequence of mask and shift operations extracts the 2-bit fields into 4-bit zero-padded fields. Now 5 such fields can be added without fear of overflow—the maximum value in a 2-bit field is 3 and $5 \times 3 = 15$ which can be represented in 4 bits. There are 66 ($\lceil 1000/15 \rceil$) 4-bit fields, now requiring only 5 words[1]. The process continues until a single field containing the 10-bit population count remains. Portions of the code are reproduced in Figure 5. This code requires approximately 170 cycles on the 21064. On a 200MHz 21064 it takes 25 minutes to enumerate the 2^{30} possible assignments of X and record the 16 best.

5.3 Partioning on w_{ij} Values in Software

Naturally, the ideas of section 4.2 can be applied to software. Indeed refinements are possible. The subtraction from K can be replaced by a subtraction from $2^n - 1$ followed by a post correction. The subtraction from $2^n - 1$ can be implemented as a negation over n bits, and the post correction can be folded into

```
m0 = 0x5555555555555555L;    /* sum single bits to pairs */
d0 = (v[0] & m0) + ((v[0] >> 1) & m0) + (v[1] & m0);
d1 = ((v[1] >> 1) & m0) + (v[2] & m0) + ((v[2] >> 1) & m0);
d2 = (v[3] & m0) + ((v[3] >> 1) & m0) + (v[4] & m0);
...
m1 = 0x3333333333333333L;    /* sum double bits to nibbles */
n0 = (d0 & m1) + ((d0 >> 2) & m1)
   + (d1 & m1) + ((d1 >> 2) & m1) + (d2 & m1);
n1 = ((d2 >> 2) & m1) + (d3 & m1)
   + ((d3 >> 2) & m1) + (d4 & m1) + ((d4 >> 2) & m1);
...
m2 = 0x0f0f0f0f0f0f0f0fL;    /* sum nibbles to bytes */
B0 = (n0 & m2) + ((n0 >> 4) & m2) + (n1 & m2) + ((n1 >> 4) & m2)
   + (n2 & m2) + ((n2 >> 4) & m2) + (n3 & m2) + ((n3 >> 4) & m2)
   + (n4 & m2) + ((n4 >> 4) & m2);
m3 = 0x00ff00ff00ff00ffL;    /* sum bytes to shorts */
...
```

Fig. 5. C Code Fragment of Population Count Algorithm

the comparison against current best ranked assignments. These refinements, invented for the software implementation, can can in turn be applied back to the hardware implementation yielding simplifications in that domain too.

We have implemented six levels of partitioning, which implies that we evaluate 64 states on each iteration. Our current code assumes that w_{ij} coefficients are distributed such that the 64 partitions are of almost equal size[1]. The population count code depends on this. The assumption can be removed either by padding the 1000 formulae with extra dummy formulae to even out the distribution, or by generating and compiling population count code tailored to the particular distribution found in the input formulae.

Extending beyond six levels of partitioning proves difficult to implement and is unlikely to yield an appreciable performance improvement.

5.4 Comparision and Update

With six levels of partitioning we find that the comparison of new ranks against the current best encountered ranks needs attention. Otherwise, time spent in this code can dominate the computation. The 64 new ranks are returned from the population count routine packed into 16-bit fields. Each time the best encountered ranks are updated, we precompute 64 packed 16-bit fields that contain the value of the worst of the current best ranks combined with the post correction for the corresponding new rank. We can now perform 64-bit wordwise subtraction and extract the high bits of the 16-bit fields. Any non-zero high bit represents a borrow in the subtraction which triggers a detailed comparison of each of the 64 new ranks. The detailed comparison is slow but rare.

[1] In terms of Figure 4, $K \approx M/2$, and so on for each subpartition.

5.5 Software Performance

Test were performed under DEC OSF/1 V3.0. The C compiler switches `-O2 -migrate` were used, enabling the GEM compiler [9]. They use six levels of partitioning. Times are for a series of 21064-based Alpha systems. Despite widely differing memory systems the run time is almost a linear function of CPU clock speed, indicating that all critical parts of the algorithm fit in on chip caches. These are uniprocessor times. The computation can be parallelized trivially across multiple CPUs by dividing the search space across the processors.

Model	Clock rate	CPU time
DEC 3000 M400	133MHz	75.64s
DEC 3000 M300	150MHz	69.45s
DEC 2000 300	150MHz	68.52s
DEC 3000 M500	150MHz	67.33s
AlphaStation 200 4/166	166MHz	60.60s
AlphaServer 2100 4/190	190MHz	52.10s
DEC 3000 M800	200MHz	49.91s
DEC 3000 M900	275MHz	36.88s

6 Comparison of Hardware and Software

To make a technologically fair comparison of hardware and software we should concentrate on the 150MHz Alphas which are contempraneous with the DECPeRLe-1 hardware. We acknowledge that the silicon process used in the 150MHz alpha is more advanced than the process used in the FPGAs on DECPeRLe-1, however we argue that FPGA processes have consistently lagged those used on leading edge processors so the comparison is fair in terms of what can be expected to be commercially avaliable at any point in time. Current FPGAs can be clocked at probably two or three times the speed of DECPeRLe-1 and could form the basis of machines with larger logic matrices than DECPeRLe-1 thus allowing greater parallelism. On the other hand CPUs have gotten faster too: software, with six levels of partitioning, takes 17 seconds on a 400MHz Alpha 21164.

Platform	Technique	Running time
Alpha 21064 150MHz	No partitioning	2039s
DECPeRLe-1 (estimate)	No partitioning	25s
Alpha 21064 150MHz	Six levels of partitioning	67.33s

Using our simple algorithm of section 3 the reconfigurable machine is 60-100 times faster than a 150 MHz Alpha. However algorithmic techniques let us speed-up the software to within a factor of two of the initial hardware speed. These same algorithmic techniques can be applied in hardware provided that a data dependent circuit recompilation is not required. If so, the reconfigurable machine can maintain a healthy lead. If however, a data dependent circuit recompilation is required the advantage of the reconfiguarble machine is lost. Whereas software compiles in seconds, traditional FPGAs require tens of minutes if not hours. In the combinatorial problem we are considering here, once a particular input data set is solved, a circuit dedicated to that data set is of no further interest.

7 Conclusions

As we apply various techniques to speed up this computation, we find that the techniques, whether inspired by the SIMD-style software or the reconfigurable hardware, are applicable to both implementation domains. There is important synergy in considering both implementation domains simultaneously.

Many of the constraints of one domain have analogues in the other. However, the analogues are not exact: some constraints that are minor for software become major drawbacks in reconfigurable hardware systems. Reconfigurable hardware offers more raw compute power than current microprocessors but certain operations, such as circuit compilation, are much more expensive than software compilation, limiting the scope of some of our techniques. Both technologies exhibit sharp degradations in performance as certain problem size thresholds are crossed: for instance exceeding the total logic resources in reconfigurable hardware, or the size of the first level cache in software. However, the degradations are more catastrophic in reconfigurable hardware. These drawbacks suggest areas for future work in the design of reconfigurable hardware systems.

Acknowledgements. I thank F. Reblewski of MetaSystèmes for providing this problem and suggesting the use of Gray Codes for state enumeration.

References

1. M. Beeler, R. W. Gosper, R. Schroeppel, *HAKMEM* MIT AI Lab Memo 239, 29 Feb. 1972
2. Leslie Lamport, *Multiple byte Processing with Full-Word Instructions*, Communications of the ACM, August 1975, Volume 18, Number 8.
3. Mark Shand, Wang Wei, Göran B. Scharmer, *A 3.8ms latency correlation tracker for active mirror control based on a reconfigurable interface to a standard workstation*, Photonics East Symposium '95. SPIE, October 1995. SPIE Volume 2607.
4. Xilinx, Inc., *The Programmable Gate Array Data Book*, Xilinx, 2100 Logic Drive, San Jose, CA 95124, USA, 1994.
5. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard, *Programmable Active Memories: Reconfigurable Systems Come of Age*, IEEE Transactions on VLSI Systems, March 1996.
6. M. Shand, *Flexible Image Acquisition using Reconfigurable Hardware*, IEEE Workshop on FPGAs for Custom Computing Machines, April 19-21 1995.
7. http://www.research.digital.com/SRC/pamette
8. Richard L. Sites (editor), *Alpha Architecture Reference Manual*, Digital Press, 1992.
9. David S. Blickstein, Peter W. Craig, Caroline S. Davidson, R. Neil Faiman Jr., Kent D. Glossop, Richard B. Grove, Steven O. Hobbs, William B. Noyce, *The GEM Optimizing Compiler System*, Digital Technical Journal, Volume 4, Number 4, 1992.