A Two Dimensional Vector Architecture for Multimedia

Ahmed El-Mahdy and Ian Watson

Computer Science Department, University of Manchester
Manchester M13 9PL, UK,
{aelmahdy,watson}@cs.man.ac.uk

Abstract. Vector processing is gaining attention for supporting multimedia workloads, particularly small subword vectors. In this paper we propose a novel vector instruction set combining the benefits of subword parallelism and traditional vector processing. We also develop a simple cache prefetching optimisation that exploits the two dimensional data access pattern of multimedia MPEG2 video applications. The architecture parameter space is explored by a simple analytical study. The analysis is complemented by detailed simulation of the actual system where it is shown that the optimised cache removes 75% of the misses and the instruction set performance is equivalent to a subword instruction with double the word size on the average.

1 Introduction

There is currently a convergence between general-purpose processors and multimedia processors [2]. Major microprocessor manufacturers have extended their instruction sets with multimedia specific instructions. These include Intel x86's MMX, PowerPC's AltiVec, UltraSPARC's VIS, and PA-RISC's MAX-2. However, more radical architecture changes are sought for meeting the increasing high performance requirement of multimedia applications [5,9].

Microprocessor technology is progressing very fast. It is expected that by the year 2010 it will be possible to build billion-transistor microprocessors with clock speeds approaching 10 GHz [1]. However, there are two main issues; memory and global clocking. Memory is not progressing at the same rate and thus the memory processor gap is increasing. With a billion transistors on a chip, wire delays will dominate and the global clocking of large synchronous systems will become a problem. Single-chip multiprocessors may be a way to overcome these problems.

In this paper we develop a *two dimensional* vector architecture for supporting multimedia on the Jamaica processor. Jamaica [12] is a proposal for a single-chip multithreaded multiprocessor targeting Java. The architecture combines the benefits of traditional vector processing and subword parallelism found in current general-purpose processors. The instruction set architecture is described in section 2 together with related work. We have observed 2D spatial locality in MPEG2 video data accesses and developed a simple cache optimisation to exploit that locality. This is described in section 3 together with related work. To explore

© Springer-Verlag Berlin Heidelberg 2001

the wide ranging parameter space, we have developed a simple analytical model. In section 4 we describe the model and use it to analyse MPEG2 encode and decode applications (mpeg2encode and mpeg2decode). Initial simulation results are presented in section 5.

2 The 2d-Vector Instruction Set

Multimedia workloads are inherently data-parallel which makes a vector architecture a suitable candidate. The vector instruction set has the benefit of decreasing address generation and loop control overheads. Our *2d-vector* instruction set combines the benefits of a traditional vector instruction set and the subword parallelism used in current microprocessor multimedia extensions.

We propose the use of 8 vector registers, each containing a maximum of 8 32-bit elements. The vector length is specified in a 'vl' register. Each element can be viewed as a simple vector of 8-bit or 16-bit subword data types. Thus a vector register is a vector of simple subword vectors, effectively a two dimensional vector.

The main memory is addressed in a *submatrix* addressing mode. The vector load instruction '`vld`', is specified as follows:

```
vld vi, Xm, Ystride
```

Where, '`vi`' is the destination vector register, '`Xm`' is the number of stride-1 memory accesses, and '`Ystride`' is a variable stride. The X and Y characters refer to row and column matrix access respectively. '`Xm`' memory words (32-bits) are loaded with stride-1 from a base address (specified in a control register), then '`Ystride`' is added to the base address, and the process is repeated until '`vl`' words have been loaded. Another addressing mode is called the transpose mode. In this mode, '`Xm`' words are accessed with the '`Ystride`', the base address is then incremented by one and the process is repeated as above. A similar instruction is defined for vector store operations.

Loads and stores are word aligned. Two other instructions are specified for unaligned access. An extra memory access is required for every row access even if the address is aligned. This makes the instruction simpler to implement and it is a compiler/programmer decision instead.

The other instructions are similar to the PowerPC's AltiVec multimedia extensions. It is worth noting that pack and unpack instructions are modified so that they take advantage of the variable vector length. Packing and unpacking halves and doubles '`vl`' respectively.

The following is an example code for the metric operation A=B+C. Where A, B, and C are 16×16 byte submatrices. Assuming that each instruction takes one cycle, the scalar code takes $16 \times (16 \times 9 + 1 + 5) = 2400$ cycles while the *2d-vector* takes $8 \times (4 \times 8 + 2) = 279$ cycles (assuming 32-bit word size).

```
Scalar instruction set                2d-vector instruction set
...                                   ...
mov r1, 16                            mov r1, 8
loop1: mov r2, 16                     loop:  !two rows added
loop2:                                       !each loop iteration
ld  r3, 0(r4); add r4, 1, r4          vld v1, 4, r9
ld  r5, 0(r6); add r6, 1, r6          vld v2, 4, r9
add r3, r5, r7                        vadd v1, v2, v3
st  r7, 0(r8); add r8, 1, r8          vst v3, 4, r9
sub r2, 1, r2; jnz r2, loop2;
add r4, r9, r4;
add r6, r9, r6; add r6, r9, r8;
sub r1, 1, r1; jnz r1, loop1          sub r1, r, r1; jnz r1, loop
```

2.1 Related Work

There are two classes for vector addressing modes; regular and sparse addressing [8]. Regular addressing is used for dense data that are organised in regular structures (fixed stride), whereas sparse addressing is used for other sparse structures. The former is more suited to multimedia applications.

Within the regular class, there are three main addressing varieties; the first is *sequential* addressing where data are stored in stride-1 organisation. Current general-purpose processors' multimedia extensions [10] implement this addressing mode. They exploit the wide data paths in microprocessors and small data types in multimedia applications and implement small subword vectors in wide registers (64- and 128-bit). However this implementation restricts matrix access and misalignment is introduced. Most implementations have permutation instructions to overcome the addressing restrictions.

An alternative addressing method is *nonsequential* addressing where data are stored with a stride-n organisation. The MOM instruction set architecture [3] implements this technique and also has a small subword vector in each vector element. This technique subsumes sequential addressing and adds extra flexibility, however, the row size is limited by the word size and misalignment problems still exist.

Our *submatrix* addressing method is closest to the MOM instruction set. The main differences are that row size is not limited by word size and the effect of misalignment is decreased for large row sizes, and column accesses are enhanced.

3 The 2D Cache

Conventional caches exploit spatial and temporal locality. Multimedia applications process massive amounts of data and temporal locality is not very abundant. Spatial locality for conventional caches occurs in one dimension; on a cache miss a line is fetched, effectively fetching nearby data. Multimedia video applications operate on small blocks of data (16×16). The operations range from regular scanning of all the blocks on a frame, to searching for a matching block.

Such searches are used, for example, in motion estimation and data access is unpredictable. However, once a block is determined all other rows are predictable. Data access has thus a two dimensional locality.

Our 2D cache works as follows: on a cache miss, $b + 1$ lines are loaded from memory. The first line fetched is the miss line, and the other b lines are prefetched with a specified stride. We assume that the data being accessed is within a big fixed matrix (e.g. a large image). The stride represents the row size of the matrix. Thus, a submatrix is effectively prefetched on a cache miss. The stride information is conveyed by vector load and store instructions. Also, a control register is used to specify stride information for scalar load and store instructions.

3.1 Related Work

The existence of 2D spatial locality in multimedia video applications was observed by Kuroda and Nishitani [10], though no design was suggested. Cucchiara et al. [4] proposed a 2D cache architecture similar to our technique where lines are prefetched on a cache miss. However, they maintain the stride information for every address referring to a 2D data structure, in a hardware table. Our technique uses instructions instead to control the 2D access; specifying current stride and possibly turning prefetching off.

Another relevant prefetch technique, though not intended to cache 2D locality, was developed by Fu et al. [6]. They developed a vector cache for vector processors. On a cache miss, either consecutive lines or stride-n lines are prefetched depending on the stride of the vector instruction. The prefetch in the latter case is similar to the one we propose. However, in our technique stride-n prefetching can also be initiated on scalar misses. A different technique is software prefetching where a prefetch is initiated by a prefetch instruction. A main drawback is the complexity of scheduling the prefetches especially for motion prediction kernels where data access is not predictable.

4 Analytical Study

The *2d-vector* instruction set has the potential to remove address generation, missalignment, and loop overheads. In addition, the subword parallelism decreases the instruction execution cycles. However the relative memory latency will increase, limiting further improvements. In the next subsection, we develop a simple analytical model that relates the overall speedup to the instruction set and cache components and gives performance bounds.

4.1 Analytical Performance Model

The performance can be decomposed into instruction execution cycles and memory access cycles. Fig. 1-a shows this situation. On average every h execution cycles a cache miss occurs and instruction execution is stalled for $T1$ cycles while memory is accessed.

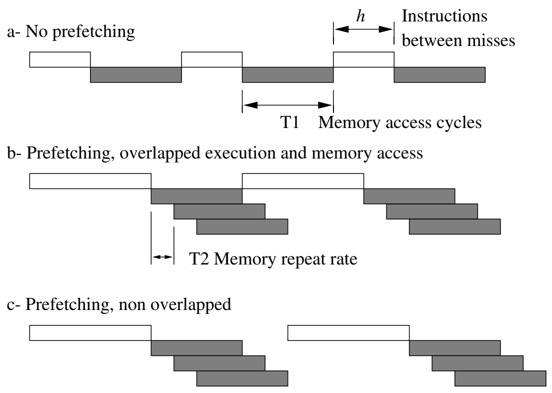
a- No prefetching
h
Instructions between misses
T1 Memory access cycles
b- Prefetching, overlapped execution and memory access
T2 Memory repeat rate
c- Prefetching, non overlapped

Fig. 1. Performance model

The effect of the 2D cache is that $b + 1$ memory access cycles happen on every cache miss but h should be increased (h'). Multiple memory accesses can be pipelined with a new memory access cycle starting every $T2$ cycles. The best case performance occurs with maximum overlap between execution and prefetching. This is depicted in Fig. 1-b. The worst case occurs when there is no overlap, as shown in Fig. 1-c.

The minimum and maximum execution cycles (t_{min}, t_{max} respectively.) are given by:

$$t_{\min} = misses \cdot \max\left(h' + T1, b \cdot T2 + T1\right) \tag{1}$$

$$t_{\max} = misses \cdot \left(h' + T1 + b \cdot T2\right) \tag{2}$$

Where $misses$ is the total number of misses.

4.2 Cache Misses

Data cache misses are relatively independent of the instruction set and machine word size. They are highly dependent on the cache and memory access patterns (application dependent). With prefetching, the number of misses will be affected by the number of lines loaded into the cache on a miss.

We have developed a cache simulation using the shade tools [11] and modelled the 2D cache using a 16Kb, 4-way set associative organisation with 32 byte line size. The benchmark programs, written in C, were run on an UltraSPARC-II processor. Our high level language target is Java, however both C and Java versions have the same data access pattern and we thus opted to use C for this exercise (the same C program is manually converted into Java). This enabled us to achieve much higher simulation speed as the program runs in native mode.

Fig. 2-a and Fig. 2-b show the number of misses for the 2D cache against the number of lines loaded on a cache miss for mpeg2encode and mpeg2decode applications [7] processing two 720×480 frames. 'Actual' is the misses obtained

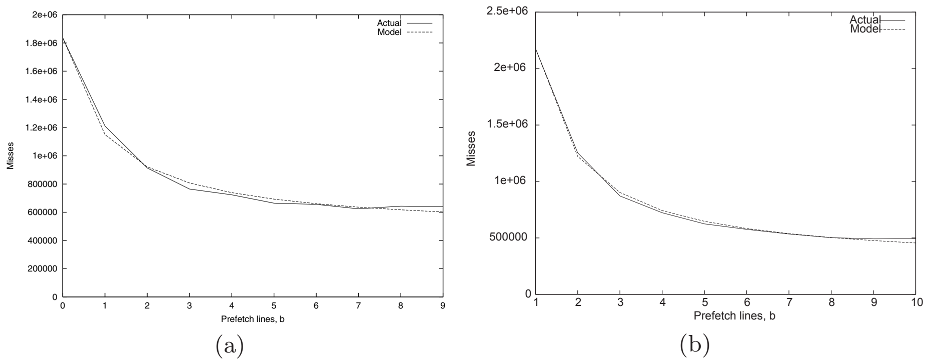

Fig. 2. Cache misses for mpeg2encode (a), and mpeg2decode (b)

by cache simulation and show that, as the number of lines is increased, up to 75% and 88% of the misses can be removed for the mpeg2encode and mpeg2decode respectively.

'Model' is the misses obtained by formulating the behaviour of the cache by the following equation:

$$misses(b) = misses(0)/(b + 1) \cdot \alpha + misses(0) \cdot (1 - \alpha) \tag{3}$$

Where $misses(0)$ is the initial misses (without prefetching), and α is the ratio of the initial misses that have 2D spatial locality. Since on a cache miss, $b+1$ lines are loaded, b misses can be removed and thus misses are decreased by $b+1$. Fitting Equation 3 to the asymptotic simulation results, we get α equal to 0.75 and 0.88 for mpeg2encode and mpeg2decode respectively. A close fit can then be observed over the range of b.

We have carried out experiments with different cache sizes. However, the working data set (3 frames, each of 338Kb) is sufficiently large that it will not be contained in realistic first level cache sizes and the effects are therefore small. It is worth noting that prefetching did not increase the number of misses in any kernel for mpeg2encode. For mpeg2decode some kernels had their misses doubled. However, these misses are less than 0.5% of the initial misses.

4.3 Instruction Execution Cycles

Fig. 3 shows the speedup of the instruction execution cycles for the vectorised kernels of the mpeg2encode and mpeg2decode benchmarks for various multimedia instruction set proposals relative to a scalar one. The programs have been manually translated into Java then compiled and assembled using the Jamaica tools. The *2d-vector* instructions can be generated directly, the others have been hand generated. The cycles are calculated assuming zero memory latency and no register spills for the scalar model (to isolate the submatrix addressing benefit). The instruction types (address generation, loop control, memory, ALU)

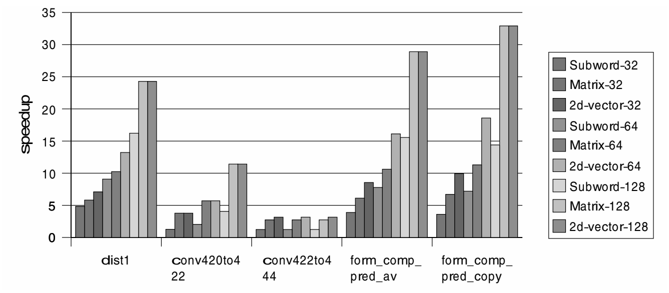

Fig. 3. Instruction execution cycles for various instruction set models

are scaled to reflect a subword based instruction set (*subword*), a subword non-sequential access instruction set (*matrix*) which is similar to MOM, and our proposed instruction set (*2d-vector*).

The *2d-vector* outperforms *subword* for all the kernels, with double the speedup on average. This is mainly due to the removal of address generation, missalignment and loop overhead. For smaller word sizes, *2d-vector* is generally faster than *matrix* (average of 25% for 32-bit word size). We have not used *subword* permute instructions nor matrix transpose instruction for the *subword* and *matrix* models in order to isolate the effect of the submatrix addressing. Using these instructions would improve the performance of all the models especially for the 'conv422to444' kernel (data parallelism is in the vertical direction).

4.4 Overall Performance

Fig. 4-a and Fig. 4-b show the speedup bounds (speedup over the scalar case with no prefetching) for mpeg2encode and mpeg2decode respectively against the memory latency. The bounds are shown for the *2d-vector* architecture with 32-, 64-, and 128-bit word sizes. Three simulations were done: two to obtain the instruction cycle speedup of the *2d-vector* over the scalar instruction set with zero memory latency (used to adjust the analytical kernel speedups to account of detailed factors and to have a better estimate of actual speedups), and the other to obtain h for the scalar instruction set. These data are used in Equations 1, 2, and 3 together with kernel speedups (section 4.3) to plot the bounds. The number of lines prefetched on a cache miss (b, prefetch count) is calculated so that it gives a near optimal (95%) upper bound. These are shown in Fig. 1-c and Fig. 1-d. Specifying the optimal upper bound will result in large values to b without having a significant improvement on the bound.

The memory repeat rate $T2$ is set to $T1/3.5$ which is a typical value. A memory access time $T1$ of 70 cycles is typical for current technology assuming 1 GHz CPU speed, we might expect $T1$ to be about 220 cycles (10 GHz CPU) in 10 years time.

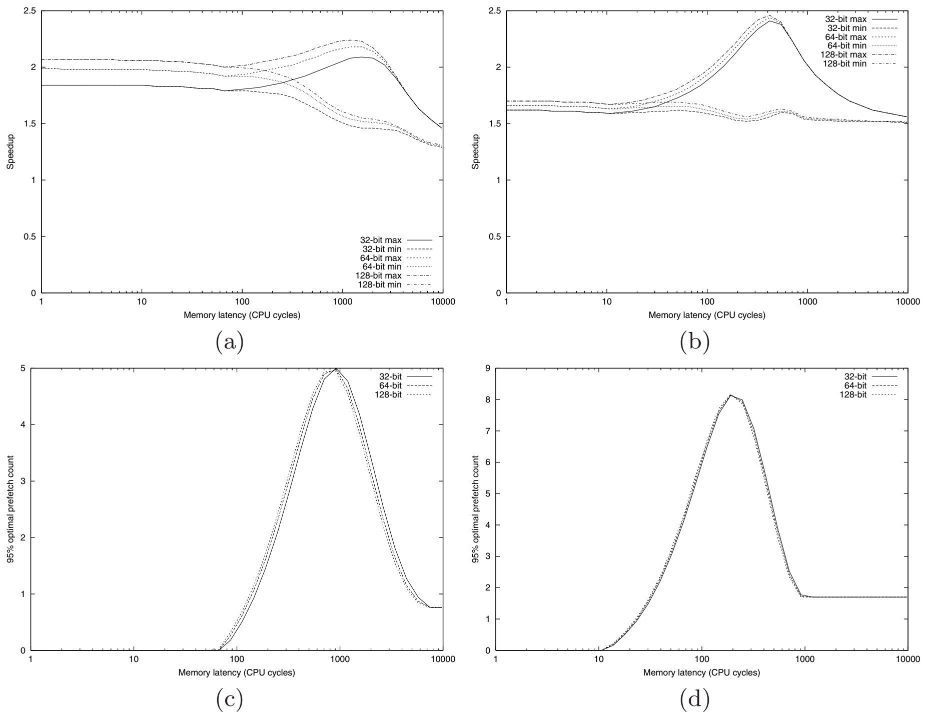

Fig. 4. Performance near optimal (95%) bounds for mpeg2encode (a), and mpeg2decode (b) and corresponding prefetch count curves (c) and (d)

For fast memory, the upper and lower bounds coincide as there is hardly any memory latency to hide and thus no prefetching is necessary. For the mpeg2-encode, as memory latency increases from 70 cycles (Fig. 4-a), the positive effect of prefetching increases, peaking in the region 400-3000 cycle. As memory latency increases further, the optimal b is limited by the relative speeds of memory access and repeat rate and cache efficiency parameter α and thus b stays constant.

The mpeg2decode is more memory bound than mpeg2encode ($h \approx 1980$ for mpeg2encode and $h \approx 335$ for mpeg2decode). The prefetching has a much more significant effect and the bounds are shifted to the right accordingly (peaking in the region 200-700). It is interesting to note that the lower bound does not decrease significantly while the upper bound peaks.

5 Simulation Study

We have carried out an initial simulation study to assess how far real results are from the theoretical bounds presented earlier on. We assume a slow memory configuration to account for the fact that the memory is shared by other processors in our multiprocessor configuration and thus is likely to emphasise the effect of the 2D cache.

The base architecture is the Jamaica processor. Jamaica is a single-chip multiprocessor with multithreading. We have extended the Jamaica simulator [12] with the multimedia extensions. The architecture, modelled by simulation, is bus based using shared memory with private 16Kb, 4-way set associative L1 caches with a 32 byte cache line. The processors share a pipelined, split transaction bus. The memory is four channel Rambus with a bus speed of 400 MHz. The processor speed is 10 GHz. We developed a Java translator tool that supports the *2d-vector* instruction set. Mpeg2encode and mpeg2decode applications are simulated, processing 2 video frames (720×480).

Fig. 5 shows the speedup obtained over the scalar case with no prefetching. The x-axis represents the number of lines prefetched on a cache miss. The simulation results are shown for mpeg2encode and mpeg2decode together with the upper bound assuming perfect prefetch ($\alpha = 1$). For the mpeg2encode, 86% of the maximum is achieved. The *2d-vector* contributes 59% speedup and the 2D cache contributes 18%. For the mpeg2decode, about 88% of the maximum is achieved. The *2d-vector* contributes 50% speedup and the 2D cache contributes 25%.

It is also interesting to compare the results with the bounds presented in section 4.4. From the simulation, we can obtain an effective memory access time. Due to the L2 cache and the bus protocol this is application dependent. For the mpeg2encode, we get $T1 \approx 455$ cycles which gives a speedup bound of 1.93 compared to the 1.79 achieved in simulation (within 7%) for 3 prefetched lines. For mpeg2decode, $T1 \approx 108$ cycles, which gives a speedup bound of 1.91 against 1.84 obtained in simulation for 7 prefetched lines (within 4%). The mpeg2decode is closer to the bound as the vectorised 'Conv420to420' kernel has a better spatial locality than the original scalar kernel.

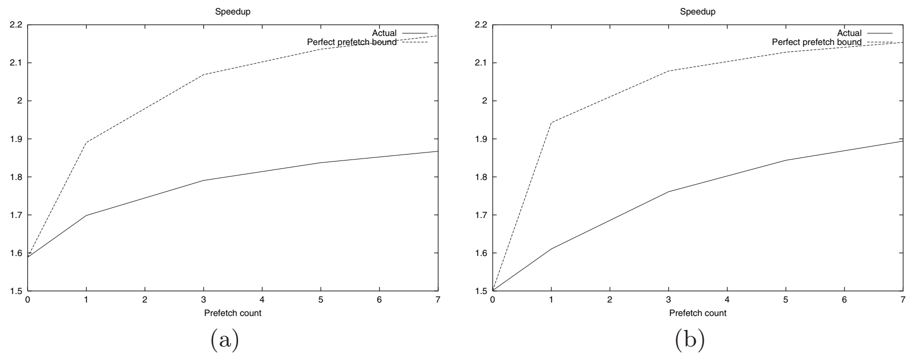

Fig. 5. Simulation results for mpeg2encode (a), and mpeg2decode (b)

6 Conclusions

In this paper we have proposed hardware support mechanisms for multimedia processing in a general purpose processor. The support comprises a vector-based instruction set with a submatrix addressing mode which utilises subword vectors. Examining mpeg2encode and decode kernels, the *2d-vector* shows a distinct performance advantages over other instruction sets.

The other hardware support is a simple cache prefetch technique that exploits the two dimensional data access patterns in MPEG2 encode and decode video applications. The technique gives a significant reduction in memory latency.

An initial detailed simulation of the system is presented demonstrating the benefits of the approach. However, an exhaustive simulation study is a subject for future work, together with examining other multimedia kernels. Future work is also needed to consider the effect of multithreading and real-time scheduling. On the software side, just-in-time vectorisation for Java programs needs to be pursued.

References

1. Doug Burger and James R. Goodman. Billion-transistor architectures. *IEEE Computer*, 30(9):46–49, September 1997. 687
2. Thomas M. Conte, Pradeep K. Dubey, Matthew D. Jennings, Ruby B. Lee, Alex Peleg, Salliah Rathnam, Mike Schlansker, Peter Song, and Andrew Wolfe. Challenges to combining general-purpose and multimedia processors. *IEEE Computer*, 30(12):33–37, December 1997. 687
3. J. Corbal, R. Espasa, and M. Valero. MOM: a matrix SIMD instruction set architecture for multimedia applications. In *SC'99 "Supercomputing Conference"*, Oregon, 1999. 689
4. R. Cucchiara, M. Piccardi, and A. Prati. Exploiting cache in multimedia. In *International Conference on Computing and Systems*, Florance, Italy, 1999. IEEE. 690
5. Keith Diefendorff and Pradeep K. Dubey. How multimedia workload will change processor design. *IEEE Computer*, 30(9):43–45, September 1997. 687
6. John W. C. Fu and James H. Patel. Data prefetching in multiprocessing vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–63. ACM, 1991. 690
7. MPEG Software Simulation Group. *MPEG-2 Encoder/Decoder, version 1.2 July 19, 1996*. http://www.mpeg.org/MSSG. 691
8. P. M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Publishing Corporation, Washington New York London, 1981. 689
9. Christoforos E. Kozyrakis and David A. Patterson. A new direction for computer architecture research. *IEEE Computer*, pages 24–32, November 1998. 687
10. Ichiro Kuroda and Takao Nishitani. Multimedia processors. *Proceedings of the IEEE*, 86(6):1203–1221, June 1998. 689, 690
11. Sun Microsystems Laboratories. *Shade V5.33A*. Mountain View, CA 94043, June 1997. 691
12. G. M. Wright. *A single-chip multiprocessor architecture with hardware thread support*. PhD thesis, Dept. of Computer Science, University of Manchester, January 2001. 687, 695