On the Potential of Tolerant Region Reuse for Multimedia Applications

Carlos Álvarez, Jesús Corbal, Esther Salamí, Mateo Valero
Departament d'Arquitectura de Computadors,
UPC. Universitat Politècnica de Catalunya–Barcelona, Spain
e-mail: {calvarez,jcorbal,esalami,mateo}@ac.upc.es

ABSTRACT

The recent years have shown an interesting evolution in the mid-end to low-end embedded domain. Portable systems are growing in importance as they improve in storage capacity and in interaction capabilities with general purpose systems. Furthermore, media processing is changing the view embedded processors are designed, keeping in mind the emergence of new application domains such as those for PDA systems or for the third generation of mobile digital phones (UMTS).

The performance requirements of these new kind of devices are not those of the general-purpose domain, where traditionally the premium goal is the highest performance. Embedded systems must face ever increasing real time requirements as well as power consumption constraints. Under this special scenario, instruction/region reuse arises as a promising way of increasing the performance of media embedded processors and, at the same time, reducing the power consumption. Furthermore, media and signal processing applications are a suitable target for instruction/region reuse, given the large amount of redundancy found in media data working sets.

In this paper we propose a novel region reuse mechanism that takes advantage of the tolerance of media algorithms to losses in the precision of computation. By identifying regions of code where an input data set is processed into an output data set, we can reuse computational instances using the result of previous ones with a similar input data set (hence the term tolerant reuse). We will show that conventional region reuse is barely able to provide more than a 8% in reduction of executed instructions (even with significantly big tables) in a typical JPEG encoder application. On the other hand, when applying the concept of tolerance, we are able to provide a reduction of more than 25% of the number of executed instructions with tables smaller than 1KB (with only small degradations in the quality of the output image), and up to a 40% of reduction (and no visually perceptible differences) with bigger tables .

1. INTRODUCTION

During the recent years, the market of mid/low-end portable systems such as PDAs or mobile digital phones have experimented a revolution in both selling volume and features. This bring to an increase in the computational demands of the devices while still having the limitation of power (and energy) consumption.

PDA systems keep including better and larger displays and more media processing capabilities (such as the ability to play MP3 audio files or MPEG-1/AVI video files [1]). On the other hand, third generation of digital mobile phones (UMTS) [2] are expected to deal with new media protocols such as MPEG-4 [3] and thus, will require enough computational raw power to decode different sources of media streams (audio, speech, video and even 3D imaging).

In order to attain the performance levels required for this new applications it is unlikely that the embedded processor domain may adopt the same evolution that characterized general-purpose processor during the last 15 years. Performance improvements in conventional processors were partly achieved by increasing the instruction level parallelism exploitation (increasing the number of instructions fetched and executed per cycle, introducing speculative execution and providing out-of-order execution capabilities). Unfortunately, these features may provide diminishing returns from the point of view of performance/power, as marginal performance improvements are provided at the cost of complexity and thus, increased power.

1.1 Recent proposals for the embedded domain

Despite several alternatives have been proposed for increasing performance, most of embedded processor designers have opted for the VLIW paradigm as the architecture of choice [4, 5, 6], due to its capability to provide performance at a very low architectural complexity. By putting the responsibility over the compiler, the VLIW paradigm allows attainable wide-issue clustered machines without compromising power consumption.

Another common trend in most common embedded processors [5, 8] is the inclusion of sub-word level parallel instructions, similar to those found in the MMX extension [7]. By operating in parallel small slots inside the same register, sub-word level parallel instructions are able to exploit the *Data Level Parallelism* commonly found in most media kernels at a very low cost.

Given the features introduced by most embedded designers during the last years, we may clearly realize that new architectural innovations in the embedded domain targeted at increasing performance must also take into account the cost in the form of additional power. Traditional features commonly found in any general purpose processor such as branch prediction and speculative execution are most likely not an option for the mobile market, as the increased IPC rate come at the cost of executing more instructions (and thus, increasing the power consumption for the same application). This property applies also to most of the current novel proposals targeted at further reducing execution time. Techniques such as value prediction [9], speculative multithreading [10, 11] or preexecution [12] all involve increasing performance at the cost of executing more instructions.

In contrast with the previous techniques, instruction/region reuse [13, 14, 15] arises as a very promising technique, as it allows to skip the execution of an instruction/group of instructions by simply accessing a memorization table, thus providing the potential to reduce both execution time and power/energy consumption.

1.2 Instruction and region reuse

The concept of *Instruction reuse* was first introduced in [13] and comes from the observation that many instructions can be skipped if another instance has already been executed using the same input values. The paper shows that a significant fraction of the overall number of dynamic instructions (in some cases, up to a 50 %) can be reused, by using, typically, *reuse tables* that keep track of the inputs and outputs of the different instructions.

Unfortunately, as already shown in [15], computation reuse is able to pay off only when multiple instructions are reused simultaneously, otherwise, we just keep reducing the latency of the reused instruction (provided this latency is longer than that required to access the reuse table). As a result, performance may only be expected when using *instruction reuse* from long latency operations such as multiplications or divisions.

Region reuse stands for exploiting the value locality exhibited by sets of instructions inside a program. These sets of instruction may have different granularity: basic blocks, traces or even whole functions can be selected as candidates for computation reuse.

There are several papers that deal with coarse-grained reuse exploitation. In [13], an additional instruction reuse mechanism is proposed based on linking data-dependent instructions onto the reuse table (trying to exploit reuse at the dependence-chain level). In [16] region reuse is exploited at the level of basic blocks (hence called block reuse) and in [17] the reuse is exploited at the granularity of traces. Connors et. al. [14, 18] propose an hybrid mechanism where the compiler is responsible of identifying regions of instructions where significant value locality is found (thanks to the use of either software [14] or hardware [18] profiling) so that special instructions may use hardware tables in order to reuse different instances of every region so identified. Finally, in [15] an interesting theoretical study is done identifying potential coarse-level computational reuse for different benchmarks in the context of a dynamic optimization setting.

In the context of instruction reuse for multimedia, Ranganathan et. al. [19] evaluated the impact of applying reconfigurable caches for instruction reuse in several media benchmarks, providing gains ranging from 4% up to a 20% of improvement. Another related work can be found in [20], where memorization is used in order to reduce the latency of multiplication and division units for media applications.

In this paper we propose a novel region reuse technique that takes advantage of the implicit characteristics of media applications. We exploit the robustness of typical media types (such as image pixels or audio samples) to losses in the precision of computation, in order to significantly increase the value locality of instruction regions with modest degradations in the quality of the outputs.

The rest of this paper is organized as follows: section 2 presents a rationale for region reuse in multimedia and introduces the concept of tolerant reuse; section 3 shows the potential benefit of region reuse, proposes a realistic hardware implementation and shows reuse gains with bounded tables; our proposed implementation of a tolerant region reuse mechanism is shown and evaluated in section 4, while final performance gains (including image degradations measurements) are presented in section 5; finally, section 6 presents a summary of this work.

2. RATIONALE FOR TOLERANT REGION REUSE

In this section we will discuss why we believe that instruction region reuse is a promising technique for the embedded media domain and we will introduce the concept of tolerant reuse.

2.1 Suitability of region reuse in multimedia applications

As already shown in related works [14, 15, 18], the main problem we encounter when facing reuse of groups of instructions (regions), is, actually, the identification of the regions themselves. The identification mechanism must face the trade-off between the length of the region and its value locality. Identification methods using both software and hardware approaches are a field of current research [14, 15, 17, 18].

Embedded media applications, on the other hand, may provide a significantly different view regarding *region identification*. Most media applications (specially those involving image, video and audio processing) are characterized for using a set of widely known algorithms (frequency transformations, convolutions, bi-linear transformations, and so on) in a pipeline fashion (see figure 2 for an example). Any given media application can be typically described as a sequence of high-level order stages (or algorithms) where the data is processed in chunks.

Our claim is that each of these high-level pipeline stages is a very good target for exploiting region reuse. These

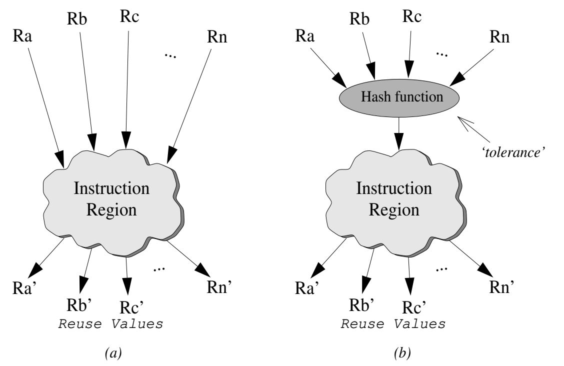

Figure 1: Basis for (a) conventional region reuse, and (b) tolerant region reuse.

algorithms are characterized for being very regular (they perform the same job over each chunk of data) and for processing recognizable media data structures (such as pixels, voxels or audio samples) that are characterized for having a high value locality. DSP code developers can easily identify regions with reuse potential, and end up taking great advantage of any region reuse facility included in a general DSP architecture (with or without compiler assistance). Compilers have already shown the capability to detect such regions (as the DCT transform illustrated in [18]).

In this paper, we will support this claim by showing that most of the different stages of a representative example of a embedded media application (a JPEG encoder) are suitable targets for region reuse.

2.2 Tolerant region reuse

In this paper, we will show that while the potential reuse of the different instruction regions is very high (due to the intrinsic redundancy in the input data sets, such as images or audio records), the real reusability drops significantly when we consider realistic table structure sizes to perform the memorization.

The main drawback of the straight-forward reuse approach is that it does not take benefit from the most relevant characteristic of the media output data sets. In sharp contrast with any other kind of application (such as any given SPECint or SPECfp benchmark), media applications exhibit a high robustness to the accuracy of their outputs. The results of 3D, video, image or audio applications may present differences that are not visually (or audibly) perceptible (property that is, actually, exploited by most of the audio and image encoders).

Our proposal is based on exploiting this property to significantly increase the reuse efficiency by relaxing the precision assumptions used for determining whether we have a hit or not in the memorization table. If we look at figure 1, we can observe the differences between the conventional reuse approach and our tolerant (or 'fuzzy') approach. As we will see in section 4, instead of using the values of a set of (live-in) registers to address a reuse table (and determine a hit with the appropriate tag that keeps record of these input values, as well as the corresponding live-out register values), we perform a hash (understood as non-injective) function over them that ends up reducing their precision. Therefore, we are increasing the likelihood of finding a previous instance that already performed the computation. For example, we might have already processed a pixel very similar to the present one. As a result, we are approximating the result of the current computation instance, introducing a small error in the calculation. As we will show, the introduced error have a very small impact on the overall quality of the application output.

We believe that the potential of the tolerant (or 'fuzzy') reuse may bring us to consider up to three dimensions of design when developing media applications: (a) the efficiency of the application (for instance, the compression rate), (b) the quality of the output (for instance, the image degradation), and (c) the performance/power ratio (increased by the tolerant reuse). Standard algorithms as JPEG already present the trade-off between quality of compression and time required to perform it.

3. EFFECTIVENESS OF REGION REUSE

In this section we will describe our methodology, we will present a case of study for region reuse and will evaluate the potential reuse with both unbounded and realistic table sizes.

3.1 Methodology and Evaluation Background

This paper is still an initial evaluation on the potential of the tolerant reuse concept. Our first approach has been to select a representative media application: a JPEG encoder, from the mediabench suite [21]. This is a very suitable benchmark to support our claims due to two main reasons:

- JPEG encoder uses several algorithms commonly found in a significant number of media applications
- JPEG is a fully detailed ISO standard, that bring us tools to evaluate image quality degradations

Since we are in the first stages of the evaluation, we do not have developed explicit ISA extensions to exploit region reuse. Our approach consists on using stylized subroutine calls to model pseudo-instructions that allows us to approximate the behavior of our proposed reuse mechanism. We have used *ATOM* [22] to perform accurate instruction count based on the detection of these stylized emulation routines. We have been simplistic in modeling the cost in instruction overhead of the reuse mechanism, but the impact (given the size of the regions and the reuse hit rates) is actually very small and do not hide not even remotely the potential gains here presented.

We believe that the reduction of the number of instructions is a powerful metric for an initial evaluation of the potential of the tolerant reuse. In the embedded domain, the number of instructions uses to be strongly correlated with the power consumption and the execution time of the application.

Along this paper, we will use two additional metrics to quantify the image degradation associated to the tolerant reuse:

the *mean square error* (MSE), and the *peak square error* (PSE).

$$MSE = \sum_{1}^{row}\sum_{1}^{col}\sum_{r,g,b}(x-y)^2$$
$$PSE = Max_{\forall_{row,col}}(\sum_{r,g,b}(x-y)^2)$$

The MSE is the summatory of the square of the difference between the original image and the output of JPEG, for all the pixels of the whole image and for each color component (red, green and blue). This metric is associated to the general degradation of the image.

The PSE is the maximum partial square error associated to a single pixel (with its three color components) along the whole image. This metric allows detecting artifacts in the output (in this case, pixels in the output that are extremely different from the original).

Due to the input-dependent nature of the mechanisms proposed in this paper, we will provide data for three different representative images, extracted from the `ijpeg` training data sets (SPECint): `penguin`, `specmun` and `vigo`.

3.2 A case of study : JPEG reuse analysis

We focused on the JPEG encoder and tried to apply region reuse to different parts of the program. JPEG is a compression standard for either grayscale and color digital images based on the DCT-method. The encoding is performed in four main stages: *color space conversion*, *forward DCT*, *quantization* and *entropy coding*.

Figure 2 depicts the whole process and the percentage of the overall execution time consumed by each part of the program (as profiled in an *Alpha* 21264 machine).

The color conversion performs the transformation from the RGB color space to the YCbCr one. The following operations are implemented for each pixel:

$$Y = C1*R + C2*G + C3*B$$
$$Cb = C4*R + C5*G + C6*B + 128$$
$$Cr = C6*R + C7*G + C8*B + 128$$

where $C1$ to $C8$ are constants and R, G and B are the pixel color components (red, green and blue).

As it can be observed, two identical RGB input sets will produce always the same YCrCb output set. Taking into account that R, G and B are 8-bit each, we just need to check these three bytes in the table to know if this result has been previously computed.

After output from the color conversion, image samples are grouped into 8x8 blocks and input to the DCT transform. The resultant coefficient values are representative of the spatial frequencies contained in the input block.

Due to the high computational cost of this bidimensional operation, it is usually broken up into sixteen unidimensional 8-length DCTs (that are further simplified using Lee's decomposition). We could think of reusing every 8-length transform, but it seems unlikely to find a coincidence in the eight input values, as there are 2^{64} possibilities. However, if we go deep into the algorithm, we find that data is partitioned in two independent groups, each one is four 8-bit values input and four 16-bit values output. Obviously, it will be easier to find a coincidence in a four data set than in an eight one, so the reuse is implemented at this point.

Note that the complete transform implies eight horizontal unidimensional transforms (rows) and then eight vertical new transforms (columns) over the previously transformed data. Therefore, the input to the second group of transforms does not have the same properties of the original data, and they must be studied separately.

Each of the 64 DCT output coefficients is then quantified with the help of a a 64 element quantization matrix. Quantization is defined as division of each coefficient by its corresponding quantizer step size, followed by rounding to the nearest integer. In this case, the result is unique for every given pair coefficient-quantizer. Thus, the reuse region is 32-bit input, and includes both division and rounding operations.

Finally, quantized coefficients are ordered in zig-zag and represented in an intermediate symbol sequence, which is then encoded with a variable-length code from a Huffman table set. In the intermediate symbol sequence, each non-zero coefficient is represented in combination with the consecutive number of zero-valued coefficients which precede it. We have considered the instructions that count the number of consecutive zeros to be the reuse region. Instead of comparing each coefficient, we examine groups of four elements, so that these four bytes are the entry to the table.

To take benefit from region reuse, the region must be long enough in terms of number of instructions, but also exhibit enough value locality. Table 1 presents the reuse region average length in every stage for the three different input images. It is seen that region length ranges from ten to fifty-five instructions (about thirty instructions per region in average). Note that region length keeps almost constant for all the tested images.

To measure the existent redundancy, we have used unbounded reuse tables, that is, all regions whose inputs are equal to those from a previous instance are reused. Table 2 shows the percentage of coincidences in the input values sets and the reduction in the overall number of dynamic instructions that they suppose for each of the regions identified in every stage. Quantification and Huffman regions exhibit very high value locality, with up to 98% of coincidences (hits). The nearly zero hit rate for the DCTcol is explained by the fact that, as mentioned above, it takes as input the output from DCTrow. On the one hand, this means that DCTcol inputs are 16-bit in front of the 8-bit DCTrow inputs; on the other

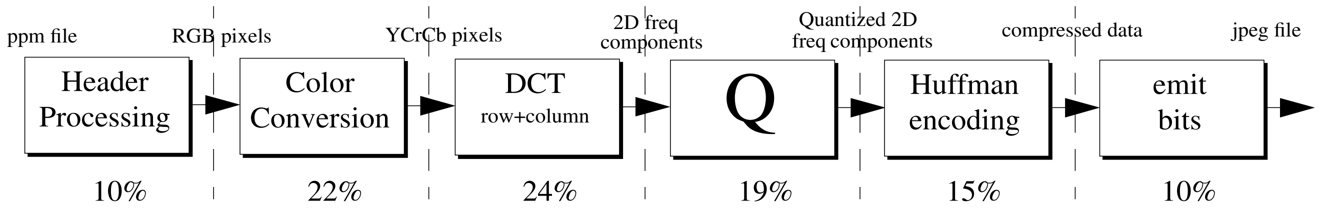

Figure 2: The different stages of a JPEG encoder application.

	average # of instructions per region		
	penguin	specmun	vigo
CColor	41	41	41
DCTrow	48	48	48
DCTcol	56	55	55
Q	16	15	15
Huffman	10	10	10

Table 1: Average number of instructions per reuse region.

	Instructions	% hits	Ins after reuse	% reduction
	penguin			
CColor	39371703	60.3	20649832	47.6
DCTrow	21907200	28.8	17944122	18.1
DCTcol	21907200	0.7	21797163	0.5
Q	34884383	98.0	17430613	50.0
Huffman	26612849	95.7	23919783	10.1
Rest	37177959	0	37177959	0.0
Total	181861294	–	138919474	23.6
	specmun			
CColor	36654576	57.2	20124852	45.1
DCTrow	20185920	30.2	16363634	18.9
DCTcol	20185920	0.4	20134218	0.3
Q	32073913	97.9	16114210	49.8
Huffman	23894314	96.1	21389550	10.5
Rest	27077991	0	27077991	0.0
Total	160072634	–	121204456	24.3
	vigo			
CColor	40916736	94.2	10557917	74.2
DCTrow	22533120	27.6	18620814	17.4
DCTcol	22533120	0.8	22408349	0.6
Q	35930219	97.2	18164836	49.4
Huffman	27706338	94.5	25026505	9.7
Rest	33759170	0	33759170	0.0
Total	183378703	–	128537593	29.9

Table 2: Potential reuse using unbounded tables across the different stages of the JPEG applications

hand, value entropy is reduced by the first DCT processing. Thus, DCTcol is not a good candidate for region reuse. Surprisingly, results do not seem to be extremely dependent on the inputs, as they remain quite similar among the three images except for the color conversion, which varies from 57% for the `specmun` image up to 94% in `vigo`. For the entire application, region reuse provides up to a 30% of reduction in the overall number of dynamic instructions.

3.3 Hardware Implementation

In the previous section we have shown that region reuse has good potential to significantly reduce the amount of executed instructions. In this section, we are going to show how this reuse approach can be implemented and evaluate the consequences of this real implementation. We will show that a realistic (say, with affordable table sizes) implementation decreases drastically the reuse potential.

The regions we are trying to reuse are characterized for having a large set of different input and output values but with few live-in registers (two, three or four for most of the cases). Therefore, we are going to study the impact of a memorization table with a limited width per entry (so that only a limited number of live-in and live-out values can be stored). As seen in figure 3, our proposed memorization table is a direct-mapped structure addressed with the lower bits of up to four registers. Every table entry contains a tag (32 bits overall shared by the up to four live-in registers), a result field (64 bits related to up to four live-out registers) and a flag (V) that indicates whether the entry is valid or not. The number of live-in registers is always the same as the number of live-out registers for every context. The architecture should provide an internal register to keep track of the number of live-in/live-out registers (up to four), that would also determine the size of each sub-tag and another internal register to store the index of the table to be updated.

To handle the mentioned table it will be necessary to have at least four new special instructions. Before the program arrives at any new reuse region (or context), the processor must execute a *Reset Instruction* that would reset the whole table by changing every *valid* flag. This avoids errors that might be produced because of reusing instances corresponding to a different context. Note that it is not unusual that two different contexts alternate in a typical media application (due to blocking transformations). A possible solution to overcome this problem could be to have different independent tables per coexistent context (as in [14]). Our first approach has been to perform some minimum changes to the JPEG application to avoid this effect, but, as we will show later, the previous alternative could be implemented easily for the tolerant versions of the hardware (due to the reduction in the table size requirements).

The second instruction, *Set Context*, would initialize the internal register that determines the number of live-in/live-out registers involved in the instruction region.

A third instruction, *Table Access* would access the table with a set of up to four registers. Note that the architecture could force those logical registers to be always located in

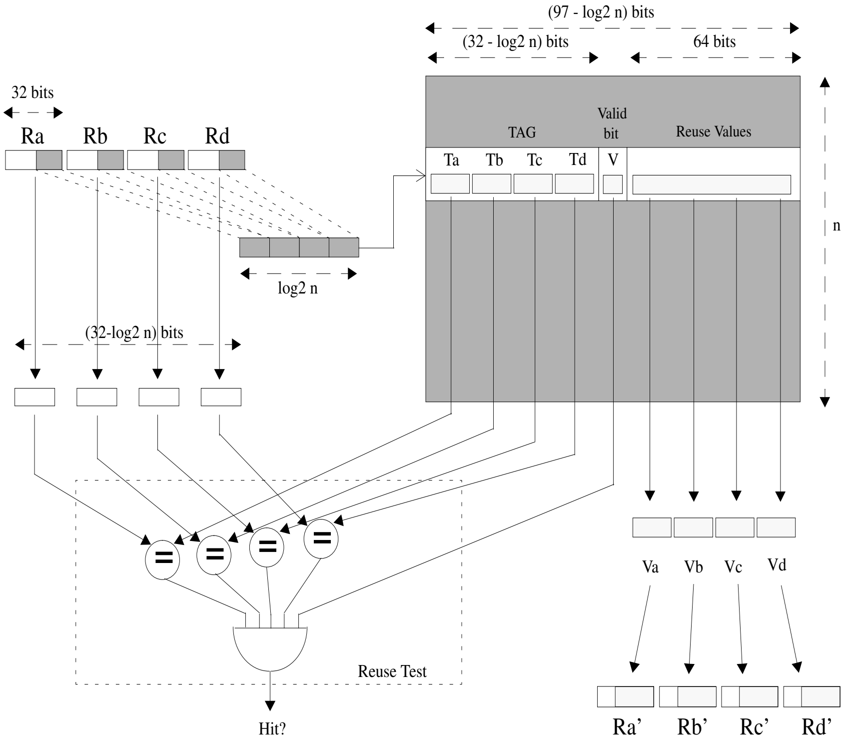

Figure 3: The conventional region reuse mechanism proposed.

the same position, so that clusterized architectures could be implemented to provide the required register bandwidth. Alternatively, given the length of the regions, multi-cycle implementations could be perfectly affordable from the point of view of performance. This instructions would check if there is a hit and would replace, in that case, the contents of the live-in registers with the result values of the table entry, jumping after that to the end of the region. The behavior of this instruction would be that of a branch, and the embedded architecture would deal with it in the same way. In case of miss, a new tag would be inserted based on the value of the live-in registers and the index of the table would be stored in the second internal register to allow the process of updating.

Finally, in case of a miss, we would require a fourth instruction (*Update Entry*) that would finalize the update process based on the most simple heuristic (that is, the entry contents are updated on every failure to perform region reuse). This instruction would execute at the end of every region and would use the index internal register to update the *Reuse Values* field with the contents of the (now) live-out registers.

3.4 Results

To evaluate the performance of the previously described hardware we have evaluated three different table sizes: 256, 1024 and 4096 entries (corresponding to 2.8, 10.9 and 42.5 Kbytes of storage). For these cases, each image and each region, we have calculated hit rates and the associated impact in the reduction of overall number of instructions (as seen in figure 4).

As it can be inferred from figure 4, effective hit rates for realistic table sizes drop significantly. There are only two main exceptions to this trend. The first one is the Huffman encoding, in which the high number of four consecutive zeros allow to high reuse percentages with short tables. Unfortunately, the Huffman reuse region is the shortest one, so, this high reuse percentage does not translate in a high reduction in the total instruction number. The second case is the Color Conversion algorithm for `vigo`, image that is characterized for having large regions with pixels of the same color. All other cases show drastic reductions in the hit rate when compared with unbounded tables, fact that brings us to believe that the efficiency of the reuse mechanism becomes severely compromised.

Indeed, the overall percentages of reduction of executed instructions is fairly modest with a realistic implementation. Let aside `vigo`, which leverages a 17 % with the 4K entries table but drops to a 8% with 256 entries, there is no other configuration that outperforms a 4% of overall reduction. There exist, however, techniques that could allow us to increase marginally these reduction rates: we have observed that loop-tiling improves the value locality in the Color Conversion algorithm and set-associativity might help alleviate the conflict problems detected in the Quantization reuse.

4. TOLERANT REGION REUSE

The previous section has shown that while region reuse might appear as a promising technique to reduce the number of overall instructions to execute, reality shows that affordable memorization table sizes are not enough to grant an efficient exploitation of the value locality.

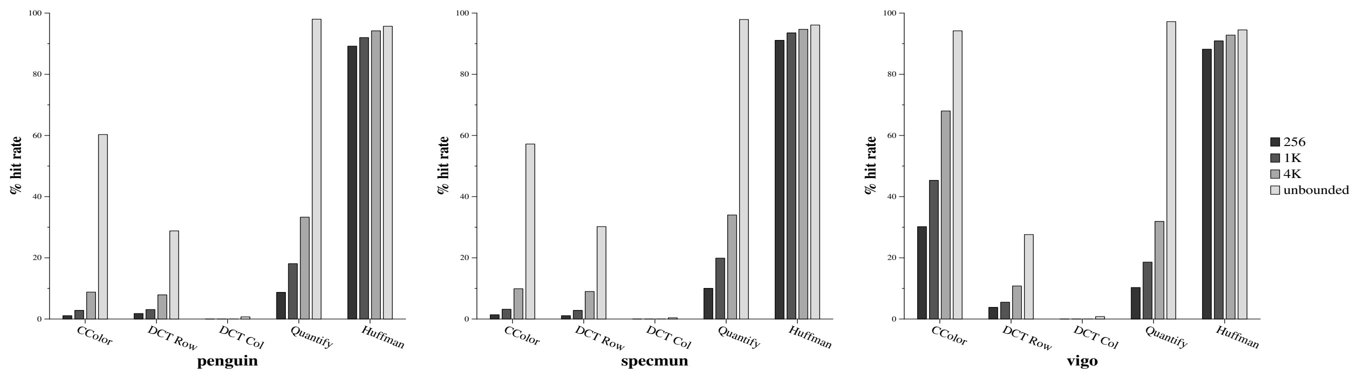

Figure 4: Reuse hit rate with bounded tables vs unbounded table.

In this section we will introduce a simple application of the tolerance reuse concept. We will propose minor hardware changes to the previous reuse mechanism that will demonstrate extremely useful to attain efficient levels of coarse-grain reuse with modest table sizes.

4.1 Introduction to the tolerant region reuse

JPEG, like other several media standards (such as MPEG), is designed to exchange accuracy by size (compression rate) but also by speed (which is a fact generally overlooked). Most of the software implementations of the standard, however, are targeted mostly to perform the accuracy/size trade-off, while in the world of embedded media processing there may be a lot of situations where speed (or power) is a factor almost as important as compression rate (for low/medium quality screens, in the presence of real time constraints, or even due to battery-life concerns). DSP designers would typically desire a way to choose the best trade-off between the two factors.

In order to trade accuracy by speed we propose a tolerant reuse model. The basic idea is that limiting the range of the input data used to access the memorization table will increase drastically the reuse. While this process will obviously have a certain impact over the quality of the output, we will show that this effect is much smaller than the potential gains in instruction reduction with affordable table sizes.

In the classical region reuse model presented above, when a new reuse instance starts, the reuse table is accessed with the live-in registers of the region. In order to assert a hit in the table, the tag and the input set must be identical. The basic concept of the tolerant reuse is to relax the hit assertion conditions, by allowing a hit not only when the input values and the tag are identical, but also when they are close enough.

To implement this distance function we introduce a hash mechanism that modifies the input values before they are used to access the table. Therefore, two different (although similar) input values can obtain the same result from the hash function and produce a hit at the same table position. The result value obtained from the table may not be the correct one but a very close approximation.

There are several possible alternative hash functions. Any distance measurement is a good candidate. For the first implementation we have chosen the simplest one, based on cutting off a certain amount of the less significant bits of every input live-in register. Therefore, two instances that only differ in the lowest-order bits will be considered to produce the same outcome. Of course, in case of miss, the computation will take place with the full precision and so will be stored the *Reuse values* field.

4.2 Hardware Modifications for Tolerant Reuse

To implement the tolerant reuse mechanism, only subtle modifications need to be done on the previously proposed reuse hardware mechanism. These modifications (as it can be observed in figure 5) are related with the inclusion of extra logic to perform the hash function over the set of live-in registers (called the *Tolerance Function Box*). Before the set of input registers is used, some of their less significant bits (up to 4) are shifted, reducing the size of the field to be compared with the tag of the associated entry, as well as changing the bits that are now used to address he table. To keep track of the number of bits to tolerate (t), we would include a third internal register (together with those used to record the number of live-in registers and the update index).

In order to fully exploit the reuse potential of Huffman encoding, we have included an alternative hash function based on performing an OR of every bit for each live-in register. This is a suitable tolerant function due to the characteristics of the Huffman encoding computation, since, for zero run-length calculation we are only interested in asserting whether the value of every byte is zero or not. This additional hash function is not reflected in figure 5 for clearness.

4.3 Tolerant Region Reuse benefits

To measure the effect of tolerance in region reuse we used the same table sizes presented in section 3.4. For every table (256, 1024 and 4096 entries) and every algorithm, tolerances of 0, 1, 2 and 3 bits have been evaluated. Figure 6 shows the hit rates achieved.

The first remarkable effect of the tolerance is the fact that the potential reuse attainable with unbounded tables increases to values over 90% for 3 tolerated bits and near 90% for 2 tolerated bits. This is due to the fact that we are approximating the computation of data structures that

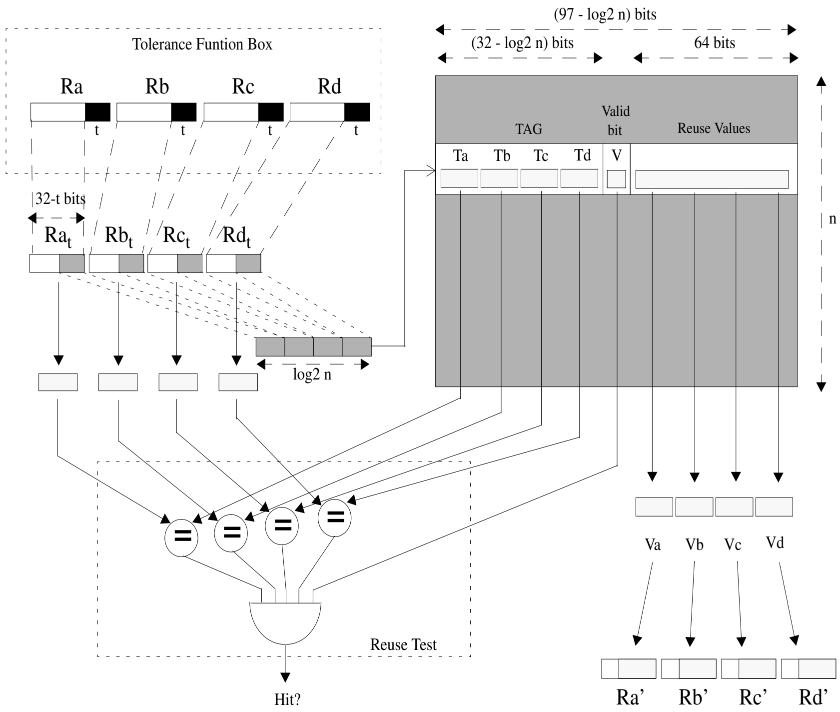

Figure 5: Tolerant region reuse hardware table.

	penguin	specmun	vigo
JPEG 100 0 0 0	17385332	14046844	20676074
JPEG 75 0 0 0	48398003	35819962	60855150
JPEG 50 0 0 0	64558080	48203396	84647203
JPEG 25 0 0 0	96170606	73996242	125839613

Table 3: MSE of the different quality conversions for the JPEG standard

are not identical but are very similar (and not distinguishable, actually, by the human eye). The only clear exception is the DCT Column algorithm, which only raises to acceptable levels (40%) when having high levels of tolerance. Furthermore, we have seen that this algorithm is actually marginally robust to losses in precision and may produce severe degradation in the output quality. That is why our final implementation will not consider *tolerance* at this particular stage.

For all the other algorithms results are clearly positive. For zero run-length (Huffman encoding) toleration raises the percentage of hits to a 99.9% (without any loss in the output) in every table size. For the other three algorithms, the tolerance application raises significantly the amount of hits in the tables, even for the smallest ones (where an average hit rate of a 40% is leveraged), becoming a factor almost more significant than the size of the table itself.

5. FINAL PERFORMANCE RESULTS

Up to this point, we have presented data regarding potential reuse gains by showing the overall hit rate in the memorization tables. We have seen that while conventional region reuse is not able to pay-off for affordable table sizes, the tolerant region reuse exhibits the potential to exploit very high amount of value locality with very modest hardware investments.

Nevertheless, all these gains are not obviously for free. This technique is based on a unique property of media applications, where we can exchange quality for speed, based on the robustness of media data types to 'approximations' in their calculations. Now it is time to evaluate the performance trade-off by presenting data on both instruction reduction and output image degradation. As we will show, considerable instruction savings come at the cost of from nearly unperceptible to very small image losses.

In order for the readers to have a way of establishing scales of quality degradation, we may look at table 3, where the MSE is calculated for different configurations of the JPEG encoder. The JPEG ISO standard introduces different parameters that are designed taking in mind the desire from application developers to control the balance between image quality and compression rate. JPEG 100 stands for the original image (that is, no losses in quality) while JPEG 75 stands for *high quality* (that is, medium compression with no perceptible losses of quality). JPEG 50 and 25 are further versions where the compression rate is increased at the cost of the image quality. We will use JPEG 75 as our baseline. Note that (a) MSE for JPEG 25 may increase up to a 100% (relative to JPEG 75), and (b) reuse percentages are not affected by the JPEG quality except in the Quantization stage, since JPEG output losses are only dependent on the different quantization tables used.

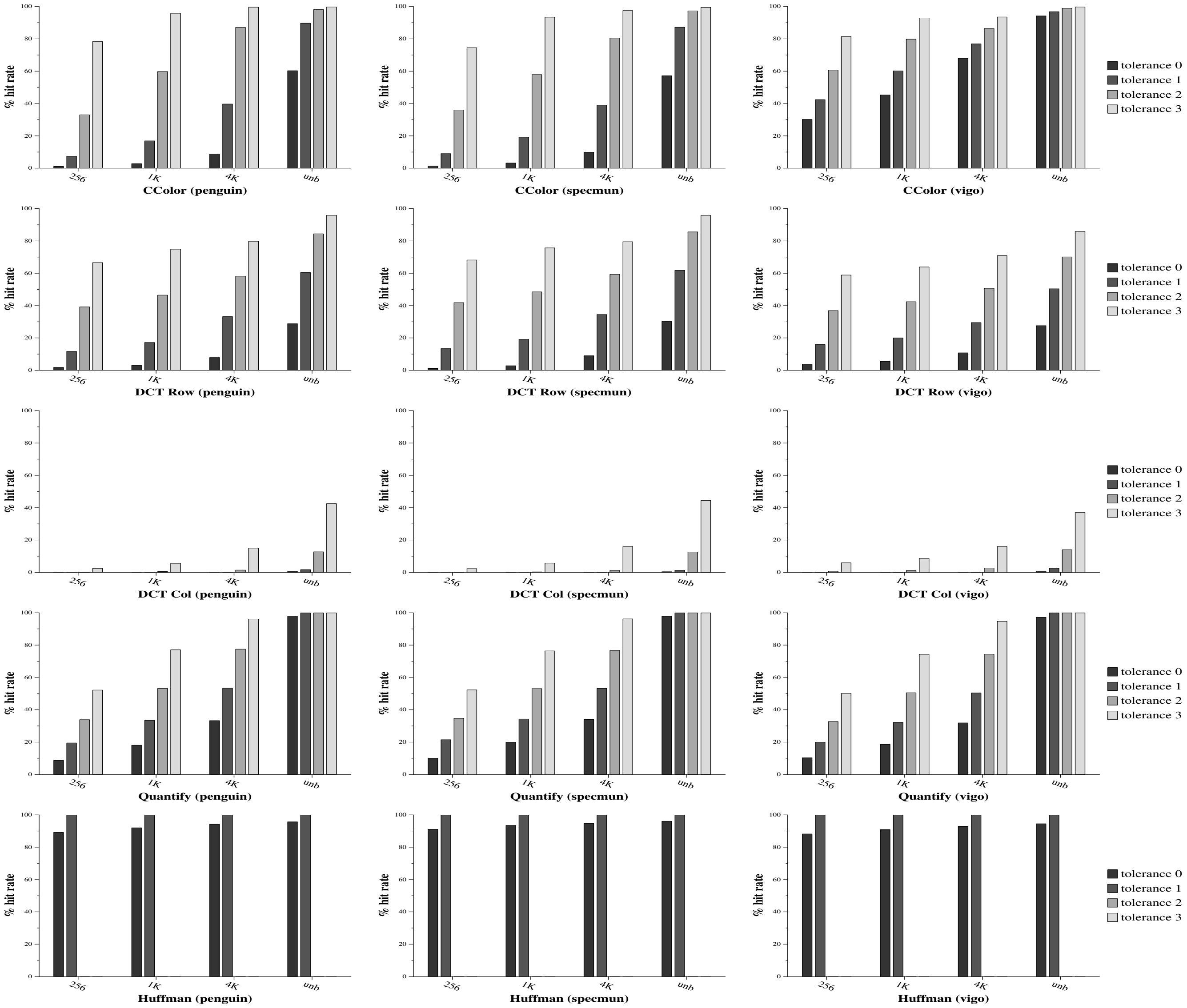

Figure 6: Hit rate percentage when increasing the tolerance degree of the reuse.

From the point of view of visual perception, less than a 10% of increase in the MSE stands for no visually perceptible differences between images. Up to a 30% stands for subtle visually perceptible differences (but still no way to distinguish the degradation in the quality of the image). Up to a 50-60% of increased MSE, the image has a comparable quality to JPEG 50 while up to a 100% of increased MSE, the quality is similar to JPEG 25.

We have evaluated the percentage of reduction in the number of executed instructions, the *Mean Square Error* (MSE) and the *Peak Square Error* for two different number of entries (64 and 4K) and for different tolerant configurations. The bigger table may arise as an interesting alternative for medium/high-end media processors where image degradation is a concern. The smaller table (0.7 Kbytes) is extremely attractive for the low-end embedded domain where a certain image degradation is reasonable. Results may be observed at tables 4 and 5.

Table 4 shows the results for the 4K entries table. Results with no tolerance at all (that is, JPEG 75 quality) are showed for means of comparison. The first five columns show the tolerated bits for every studied algorithm. Note that the DCT Column stage is not evaluated with tolerance, while for Huffman, the special OR hash function has been used for all the cases with tolerance. For every case, three measures are presented: first, the total instruction reduction percentage, second, the percentage of increase in the Mean Square Error and third, the Peak Square Error.

From results in table 4, we can observe that reductions as high as 30-35% of the overall number of executed instructions may be leveraged with nearly non-existent image quality degradations (less than a 5% of increase in the MSE). As a matter of fact, with reasonably big enough table sizes, tolerance provides a big boost in reusability (from a 8% to a 30%) without having to compromise the output. Furthermore, if image degradation is not that of a concern, our

					penguin			specmun			vigo		
CColor	DCT row	DCT col	Quantify	Huffman	% red	% mse	pse	% red	% mse	pse	% red	% mse	pse
					no quality degradation (no tolerance)								
0	0	0	0	0	7.5	0.0	1.9	8.3	0.0	2.9	18.3	0.0	4.2
					visually unperceptible differences								
0	0	0	2	1	11.8	0.0	1.9	12.6	0.0	2.9	22.6	0.0	4.2
1	0	0	2	1	17.1	0.1	1.9	17.9	0.1	2.9	24.1	0.1	4.2
1	1	0	2	1	20.9	0.3	1.9	21.9	0.4	2.9	27.0	0.2	4.2
1	1	0	3	1	22.8	0.5	1.9	23.9	0.9	2.9	29.1	0.4	4.6
2	1	0	2	1	28.6	1.6	1.8	29.2	2.0	2.9	28.6	1.0	4.2
2	1	0	3	1	30.5	2.1	1.8	31.2	2.3	2.9	30.7	1.2	4.6
2	2	0	3	1	34.5	4.0	1.8	35.2	7.5	3.0	34.0	3.8	4.8
					subtle visually perceptible differences								
3	2	0	3	1	36.3	14.7	1.7	38.0	20.6	3.1	35.2	9.4	3.8
2	3	0	3	1	37.8	32.4	2.1	38.5	39.9	2.8	37.1	28.4	4.4
					low image degradation (similar to JPEG 50)								
3	3	0	3	1	39.8	43.0	2.3	41.4	60.7	3.0	38.3	31.0	3.8

Table 4: Final results for low image degradations and high reuse (4K entries).

					penguin			specmun			vigo		
CColor	DCT row	DCT col	Quantify	Huffman	% red	% mse	pse	% red	% mse	pse	% red	% mse	pse
					no quality degradation (no tolerance)								
0	0	0	0	0	2.3	0.0	1.9	2.4	0.0	2.9	6.3	0.0	4.2
					visually unperceptible differences								
2	2	0	4	1	14.2	2.5	1.8	16.5	3.7	2.9	18.5	2.7	4.8
3	1	0	3	1	16.3	5.6	1.8	17.4	7.1	2.9	18.5	4.6	4.2
3	1	0	4	1	17.9	6.2	1.8	19.0	8.0	2.9	19.9	5.1	4.2
					subtle visually perceptible differences								
3	2	0	4	1	20.3	7.7	1.8	22.1	10.3	2.9	22.5	6.6	4.2
					low image degradation (similar to JPEG 50)								
3	3	0	4	1	24.8	33.0	1.8	26.8	47.9	2.9	26.1	28.9	4.5
					moderated image degradation (similar to JPEG 25)								
4	3	0	4	1	28.8	78.8	2.0	30.2	98.6	3.0	28.5	52.6	4.1
					unacceptable degradation								
4	4	0	4	1	33.3	270.3	2.5	34.7	356.8	3.6	31.9	185.8	4.6

Table 5: Final results with 64 entries for moderated image degradations (suitable for small screens, fast previews, fast compression mode).

tolerant reuse mechanism is able to provide up to a 40% of instruction reduction (which would translate into very high performance and/or low power consumption) with reasonable quality (comparable to JPEG 50).

As seen in the table, those quality degradations depend not only on the degree of tolerance but also on the robustness of the algorithm itself. The most tolerant algorithm is Quantization, followed by Color Conversion and finally DCT row transformation. Moreover, degradation is not a linear function of the tolerance degree, but an exponential one. Therefore, low tolerances applied to all the algorithms is the more suitable configuration to achieve the best performance/accuracy trade-off, rather than having high tolerance degrees in only a single stage of the application.

Another very significant effect that can be seen is that our tolerance method does not raise the Peak Square Error (on the contrary, in some cases the PSE decreases). As a matter of fact, our tolerant reuse paradigm has a side-effect that translates into a low-pass filtering of the output data (that is, pixel components are smoothed, avoiding artifacts from the JPEG original algorithm). This provides a certain benefit in the form of reducing image degradation, as the human eye is certainly much more sensible to big but localized errors than more abundant but sparse small ones.

Finally, table 5 shows the same results for the 64 entries table (that is, a suitable hardware configuration for a low-end embedded media processor). Results show that up to a 20% of instruction reduction is obtained with still extremely acceptable image qualities (around 10% of MSE increase). If image quality is not that of a concern, the hardly larger than 1KB table is able to provide around a 25% of reduction with quality similar to JPEG 50 and up to a 28-30% with quality similar to JPEG 25.

6. SUMMARY

In this paper we have shown that coarse-grain computation reuse is a promising technique for the embedded media domain. By significantly reducing the overall number of required instructions to execute, we are targeting the two premium goals of any low-end media processor: reducing the execution time (without having to increase the complexity of the architecture to exploit higher parallelism) and reducing the overall energy consumption (and possibly power).

We have studied one representative media benchmark and we have shown that there exist many opportunities for region reuse along the different stages of the application. This is due to the fact that media algorithms are extremely regular and deal with media data types (such as pixels or audio samples) that are characterized for a high value locality. Unfortunately, reasonable table sizes are not enough to provide significant reuse gains.

The main contribution of this paper is the introduction of the concept of tolerance in region reuse. We exploit a characteristic that is unique to media applications: the output data sets are highly robust (tolerant) to inaccuracies in the computation. We have proposed a novel reuse mechanism that performs a hash over the input data sets of a computational instance so that we can reduce their precision. In this way, we are increasing the likelihood of finding a previous instance already stored in the memorization table. We have shown that the associated image degradation is extremely small compared with the improvements from the point of view of percentage of reused instructions.

We have provided data that show that while conventional region reuse may be able to provide a potential reuse of up to a 30% of reduction in the number of overall instructions to execute, these performance results drop significantly when we consider realistic memorization table sizes. In sharp contrast, when using tolerant reuse, we are able to effectively exploit a higher reuse locality (up to a 40%) with significantly smaller tables and at the cost of very modest image quality degradations.

We envision two main design targets using the potential of the tolerant reuse: one for the high-end embedded domain, where we can use relatively bigger tables (around 16-48 KB), in order to provide very high performance (from 30% to 40% of overall instruction reduction in the JPEG encoder) with near inexistent losses in the quality of the image, and one for the low-end (portable devices with small, low-quality screens such as PDAs), where we can achieve very good performance (up to a 25%) with extremely small tables (hardly larger than 1KB).

6.1 Future work

While we have presented a novel mechanism with great potential, this paper is still an initial evaluation. As future work, we are planning to design a consistent set of ISA extensions to exploit tolerant reuse on regular data structures (vectors typically). Many trade-off designs still remain to be done, specially involving an evaluation of execution time and power consumption.

Our claim is that this kind of region reuse can be applied to most of image, video, audio and 3D graphics applications. As a matter of fact, most of the relevant stages of the JPEG application showed potential for region reuse. Our intention is to apply the same strategy to a wider set of representative benchmarks.

7. ACKNOWLEDGEMENTS

This work was supported by the Ministry of Education of Spain under contracts CICYT98-0511 and the CEPBA.

8. REFERENCES

[1] *Cassiopea* home page. http://www.casio.com/personalpcs/product.cfm. Casio, 2001.

[2] http://www.umts-forum.org/information.html. Home page, UMTS Forum, 2001.

[3] Rob Koenen. Mpeg-4, multimedia for our time. *IEEE Spectrum*, pages 26–34, February 1999.

[4] TI. TMS320C62XX family. Technical Report http://www.ti.com/sc/docs/products/dsp/tms320c6201.html, Texas Instruments, 1999.

[5] Philips Semiconductors. Trimedia tm-1300. http://www-us3.semiconductors.com/trimedia/, 1999.

[6] Analog Devices. Introducing tigersharc. *Whitepaper*, http://www.analog.com/new/ads/html/SHARC2.

[7] Mmx technology programmers reference manual. Technical Report http://developer.intel.com/drg/mmx/manuals, INTEL corporation, 1998.

[8] Semiconductor Group Siemens. Tricore. 32-bit single-chip uc-dsp. *Technical Summary*, http://www.tri-core.com/, 1998.

[9] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit. pages 226–237, December 1996.

[10] J.G. Steffan and T.C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. *Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[11] H. Akkary and M. Driscoll. A dynamic multithreaded processor. *International Symposium on Microarchitecture*, 1998.

[12] A. Roth and G.S. Sohi. Speculative data-driven multithreading. *Seventh International Symposium on High-Perfomance Computer Architecture*, 2001.

[13] A. Sodani and G.S. Sohi. Dynamic instruction reuse. *ISCA 24*, 1997.

[14] D.A. Connors and W.M. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. *MICRO 32*, 1999.

[15] S.S. Sastry, R. Bodik, and J.E. Smith. Characterizing coarse-grained reuse of computation. *3rd ACM Workshop on Feedback-Directed and Dynamic Optmization, in conjuction with MICRO 33*, 2000.

[16] J. Huang and D.J. Lilja. Exploiting basic block calue locality with block reuse. *HPCA 5*, 1999.

[17] A. Gonzalez, Jordi Tubella, and Carlos Molina. Trace-level reuse. *ICPP*, 1999.

[18] D.A. Connors and W.M. Hwu. Hardware support for dynamic activation of compiler-directed computation reuse. *ASPLOS 9*, 2000.

[19] P. Ranganathan, S. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. *ISCA 2000*, Vancouver.

[20] D. Citron, D. Feitelson, and L. Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and division units. *ASPLOS VIII*, 1998.

[21] C. Lee, M. Potkonjak, and W.H. Magione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communication systems. *MICRO 30*, 1997.

[22] A. Srivastava and A. Eulace. Atom: A system for building customized program analysis tools. *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation.*