A Flexible Multi-port Caching Scheme for Reconfigurable Platforms

Su-Shin Ang[1], George Constantinides[1], Peter Cheung[1], and Wayne Luk[2]

[1] Dept. of Electrical and Electronics Engineering, Imperial College, London
[2] Dept. of Computing, Imperial College, London
sa4@imperial.ac.uk

Abstract. Memory accesses contribute sunstantially to aggregate system delays. It is critical for designers to ensure that the memory subsystem is designed efficiently, and much work has been done on the exploitation of data re-use for algorithms that exhibit static memory access patterns in FPGAs. The proposed scheme enables the exploitation of data re-use for both static and non-static parallel memory access patterns through the use of a multi-port cache, where parameters can be determined at compile time and matched to the statistical properties of the application, and where sub-cache contentions are arbitrated with a semaphore-based system. A complete hardware implementation demonstrates that, for a motion vector estimation benchmark, the proposed caching scheme results in a cycle count reduction of 51% and execution time reduction of up to 24%, using a Xilinx XC2V6000 FPGA on a Celoxica RC300 board. Hardware resource usage and clock frequency penalties are analyzed while varying the number of ports and cache size. Consequently, it is demonstrated how the optimum cache size and number of ports may be established for a given datapath.

1 Introduction

FPGAs have become natural platforms for design implementation or prototyping due to their re-programmability and comparatively short design cycle. One of the main advantages that FPGAs have over traditional processors is the massive amount of available parallelism. External memory bandwidth available for reconfigurable logic, however, has not developed at the same rate, limiting the effective amount of achievable parallelism. Hence, it is critical to account for the memory subsystem during the design process.

Much work has been done in the development of scratchpad memories (SPM) [1,2,3] for algorithms with static memory access patterns. However, algorithms such as the Huffman decoder and some motion vector estimation approaches [4] exhibit data dependent memory access patterns, and as a result, the memory accesses cannot be predicted at compile time.

In this work, a flexible multi-port caching scheme is presented. Besides the exploitation of data re-use inherent in an algorithm, this scheme allows accesses for an arbitrarily parallelized data path and so may be transparently used alongside an existing hardware design. Parallel cache-system accesses are detected and

K. Bertels, J.M.P. Cardoso, and S. Vassiliadis (Eds.): ARC 2006, LNCS 3985, pp. 205–216, 2006.

arbitrated if they are contending for the same sub-cache. A significant speed-up of up to 24% in execution time and a cycle count reduction of up to 51% is observed for a cache size that is approximately 3% of image size for a benchmark application involving motion vector estimation. The contributions of this work are as follows:

1. A novel parameterisable cache design, based on a semaphore-style arbitration scheme, is developed to allow user transparency and parallel accesses to multiple sub-caches.
2. A complete implementation of the caching scheme, including the quantification of clock period degradation and area overhead.
3. FPGA-based *in situ* hardware profiling to determine the trade-off between resource usage and performance benchmark algorithm.

This paper is organized as follows: in Section 2, work related to this paper will be discussed and an overview of the multi-port caching system is given in Section 3. The architecture of the caching system will be discussed in Section 4. In Section 5, implementation details and experimental results for a motion vector estimation algorithm are presented and analyzed and finally, the paper is concluded in Section 6.

2 Related Work

Caches are widely used to exploit data re-use within algorithms. A large volume of work has been done on the improvement of cache performance for software applications [5]. These include techniques to optimize data placement and reduce cache misses [6,7], as well as to reduce the number of tag and way accesses [8].

In [9], a dynamic scheme is used for the allocation of variables to scratchpad memory (SPM) which is implemented using block RAMs. Profiling and loop transformation are carried out by the compiler. Based on this profile, the variables are allocated to the SPM for the exploitation of data re-use. However, this approach only considers static memory access patterns. Another compiler that is capable of detecting data re-use is [10]. Smart buffers are inserted at the input and output of the datapath and these in turn interface with external memory. These buffers store windows of data that are re-used within the loop body such that external memory accesses are reduced. Similarly, this technique only accounts for static memory access patterns.

Some papers have been published on multi-port caches: in [11], a multi-port cache is implemented using interleaved cache banks targeting the MIPS 2000 instruction set. This work targets superscalar processors, enabling multiple instructions to be carried out in parallel. The cache bandwidth, however, is limited by the maximum number of instructions that can be issued, restricting the design space that can be explored. In [12], a multi-port cache is implemented by cache duplication. This requires the updating of multiple cache locations in the event of cache misses. The number of ports is restricted to two on the particular platform so the trade off between resources and parallelism is not explored.

This work is targeted at FPGAs. Consequently, cache parameters have to be chosen to match well with the underlying device granularity. The user can determine the number of ports to access cache contents, providing greater leverage over total execution time and resource usage. By taking advantage of the reconfigurability of FPGAs, profiling is carried out *in situ*, on a hardware platform. This allows a wide range of designs to be explored quickly and accurately compared to software modelling. Most importantly, this scheme allows the exploitation of data re-use for non-static memory access patterns.

3 Overview of Multi-port Caching System

Memory accesses can be categorized into different types. During compile time, it might be impossible to determine the exact cycle that main memory is accessed due to data-dependent control. This type of access has *dynamic* timing. Accesses with non-dynamic timing are referred to as *static*. Statically timed memory accesses can have either static or dynamic addresses sequences (dynamic address sequences occur as a result of data dependency). Three major points distinguish this work from others:

1. Previous schemes [9,10] for FPGAs are capable of handling accesses with static timing and address sequence. The proposed caching scheme on the other hand is able to handle dynamic accesses. Therefore, it is potentially more effective for data dependent algorithms.
2. Memory-based optimizations often involve substantial changes to the code [10]. The proposed caching scheme optimizes memory accesses with minimal changes to the high-level code. Further, it does not require the user to sequentialize external memory accesses manually.
3. Multi-ported caches [11] have been explored before. However, our work targets FPGAs where the design space is often larger but permits more rapid and accurate exploration.

In Figure 1(a), the datapath and the proposed caching system are illustrated. Data items are retrieved from external main memory through the cache. N sub-caches are used to provide the parallel accesses required by the datapath, and each of the sub-caches is a variant of a direct-mapped cache.

There are two levels of connectivity in this system. The first level connects the datapath to the cache. M ports allow communication between the caching system and the datapath. Specifically, the datapath can access any of the N sub-caches using any of the given ports. A crossbar switch is therefore necessary to realize this functionality. Given that addresses presented at these ports could contend for the same sub-cache, there is a need for an arbiter to sequentialize accesses should this situation occur. The second level connects the sub-caches to external main memory, which it is assumed has only one port. Since more than one sub-cache might wish to access main memory, the interface to main memory again needs to be able to sequentialize accesses in that situation.

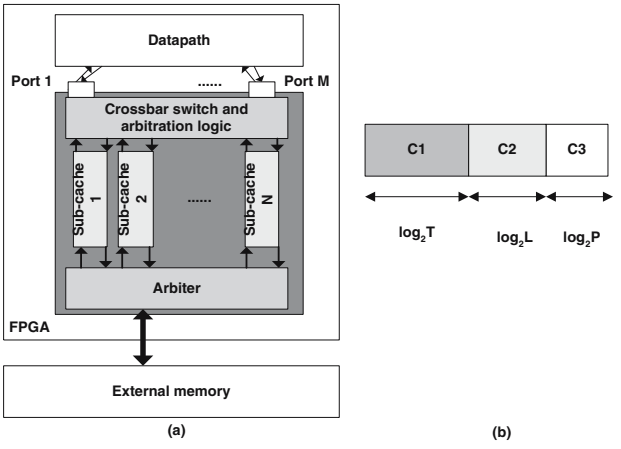

Fig. 1. Proposed multi-port cache: (a) Top-level diagram of caching system. (b) Address mapping scheme.

The address mapping scheme has a transparent address interface; this is shown in Figure 1(b). The address is split into three components: the most significant $\log_2 T$ bits make up the tag of the address, the middle $\log_2 L$ bits are used to determine the correct line within the cache, and the least significant $\log_2 P$ bits are used to determine the sub-cache that is currently targeted. The components are arranged in this order to allow spatial locality of memory accesses to be exploited. Indeed, consecutive sub-caches will store items from consecutive addresses of main memory because the address bits that determine the target sub-cache are the least significant bits.

4 Usage and Arbitration Scheme

This caching scheme is designed in a completely user-transparent way, using a semaphore-based system. An example usage of the cache is shown in Figure 2. Figure 2(a) shows the original source code containing a function stub *func*. The input parameters of the function, *address0* and *address1*, which may not be known at compile time, are used in the retrieval of data items *data0* and *data1* from a common external memory. The result of the computation is then returned to register *O*. To make use of the cache, the external memory access macros are replaced with cache access macros as shown in Figure 2(b). Parallel cache accesses are made possible through the use of the crossbar switch and arbitration logic. It is important to note that in Figure 2(a), assuming only one port of access, the user has to ensure that multiple external memory accesses have to take place in different cycles or the data retrieved will be incorrect, whereas this is transparently ensured by the cache access macros in Figure 2(b).

In Figure 2(b), sub-cache contention may occur. This type of access has static timing but dynamic addressing since the addresses are data-dependent, whereas

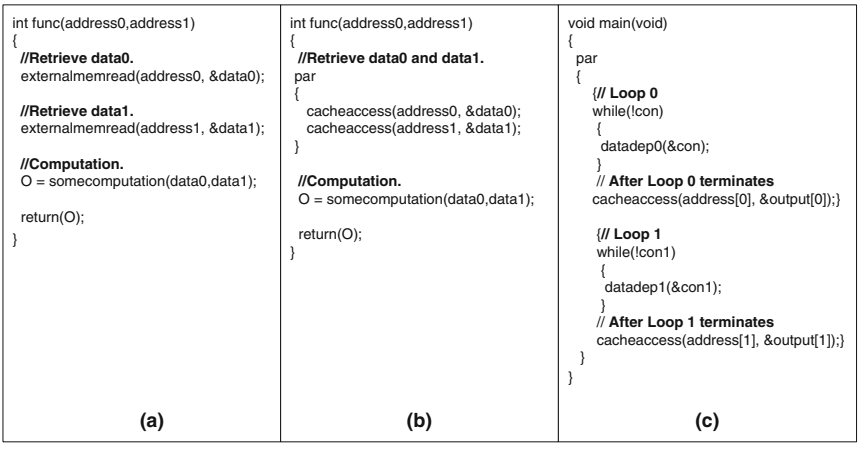

Fig. 2. Cache usage: (a) Original code. (b) Cache substitution. (c) Indeterminate memory access. Note: construct *par* indicates that statements encapsulated within its braces are carried out in parallel.

in the latter example, an example of a memory access with dynamic timing is seen in Figure 2(c). Two concurrent loops are running in parallel and two data dependent functions: *datadep0* and *datadep1* determine when the loops terminate; cache access takes place after loop termination. If *con* and *con1* are asserted in the same cycle, then concurrent cache accesses will take place. If the two accesses are targeting different sub-caches, accesses will take place concurrently. However, under the proposed scheme, these accesses will be sequentialized if the same sub-cache is targeted.

In the proposed scheme, semaphores are used for the architecture of the arbiters at both levels of connectivity to automatically ensure sequential access to the sub-caches as well as external memory when there are multiple requests, facilitating user transparency. The architecture of the arbitration scheme is detailed in the rest of this section.

In [13], algorithms described in a high-level language are translated into hardware by complementing the data path with a token-based control path: a statement is executed when it captures a token; the statement releases this token only upon completion of the task specified by the statement. The token may be duplicated and passed to mutiple statements meant to be carried out in parallel. Upon completion of the task, the token belonging to the statement that consumes the largest number of cycles will be transferred to the next statement in sequence. The proposed arbitration architecture uses such a token-based control scheme. Figure 3(a) shows the block diagram of the semaphore-based system. Token I_x, $1 \leq x \leq N$, is captured by the request block when an assertion is detected. Subsequently, a request for the semaphore guarding the resource is submited using a *trysema* statement; up to N *trysema* statements potentially compete for the semaphore but only one is allowed access to the resource.

Equivalently, only one token, S_x may be granted such that only one statement, x is allowed access to that resource at a time. The semaphore is released when token R_y, $1 \leq y \leq M$, is captured by the Sema state block, which in turn activates the *releasesema* statement, making the semaphore available to other requests. Signal *State* is asserted if the semaphore is captured.

Specifically, the function of individual blocks is described by Boolean equations in Figure 3(b). If I_x is asserted, the corresponding Request block is used to check if the resource is currently occupied. If the resource is free, Q_x is asserted. Otherwise, Q_x is not asserted, but the request is remembered by asserting input of register, P_x^+ for consecutive cycles until the resource is eventually free as shown in line 2 of Figure 3(b). P_x^+ will also be asserted if the semaphore is free but the request is over-ridden by other statements, such that $S_x = 0$. If the semaphore is free, the Priority encoder block is used to determine the statement that is allowed access to this resource. Among the asserted Q_x values, it chooses one with the smallest value of x resulting in lines 3 to 4. If any $S_x = 1$, then $Captured = 1$. The state of the semaphore in the next cycle, $State^+$, determined by the Sema State block will be asserted if $Captured = 1$, or if the resource is currently busy and none of the *releasesema* statements have been asserted as shown in line 5. For this system, the area and delay growth are $O(N^2)$ and $O(\log N + \log M)$ respectively.

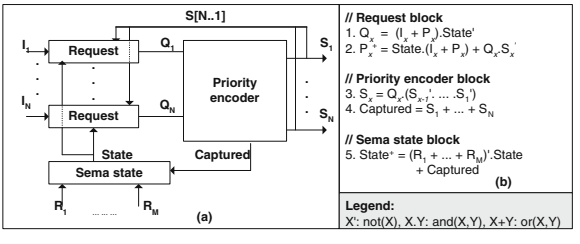

Fig. 3. Architecture of a semaphore-based system: (a) Block diagram. (b) Boolean equations for individual blocks.

5 Implementation and Results

The effectiveness of the caching system is shown in the following sections. The cache is expected to reduce the cycle count. However, degradation in clock speed as well as greater resource utilization will also occur. The experimental setup used to investigate this caching scheme and the performance-resource usage trade-offs in practice will be presented in the following sections.

5.1 Experimental Setup

A memory intensive variant of motion vector estimation [14] is used as a benchmark circuit to test the effectiveness of the caching system. This algorithm and

the proposed multi-port cache are implemented using the Handel-C [15] language, which includes semaphores as a built-in construct.

The RC300 board [16] from Celoxica containing a Xilinx Virtex XC2V6000 FPGA is used for this experiment. The FPGA contains 33792 slices and 144 block RAMs [17]. Two external synchronous SRAMs (SSRAM) are used to store image frames. Only one port of access exists for each SSRAM and each access requires two cycles [18]. On-chip block RAMs are used for the implementation of the cache. The access time for block RAM access is one cycle, but logic overheads prolong access time to two cycles for the semaphore-based system which is the same as external memory access time. Therefore, a reduction in overall cycle count comes only by parallelizing accesses to the sub-caches.

Two experiments were conducted. For both experiments, each design is indicated by *S_X_Y_Z* in Sections 5.2 and 5.3, where *X* indicates the number of ports, *Y* indicates the logarithm of the number of cache lines (base 2) within 1 sub-cache, and *Z* represents the search window size. Two motion vector search window sizes, 7 and 15 pels, are used where a pel indicates a block region in an image frame of size of 16 by 16 pixels. The number of pels represents the distance of the search center from the boundary of a square search area. In Experiment 1, execution time and resource usage are monitored while the number of ports is varied. The number of data items in the cache is held constant at 2^{11} (approximately 3% of frame size). These designs are compared with a reference design where no cache is included. Intuitively, execution time will fall with the increasing parallelism afforded by the increasing memory bandwidth. At the same time, the extent to which spatial locality is exploited increases under the mapping scheme described in Section 3, implying an increased incidence of cache hits. However, degradation in clock speed and resource usage are expected because of logic resources used in the implementation of increasing numbers of semaphores as well as the size of the crossbar switch. In the experiment, the optimum number of ports is established empirically.

In Experiment 2, for each window size, the execution time and resource usage is monitored while the number of cache lines is varied for a constant number of ports, which are found to give the minimum execution time in Experiment 1. With an increase in the number of cache lines, the number of cache hits should increase resulting in execution time reduction. However, more storage and routing resources are needed to accommodate the extra cache lines, leading to degradation in clock speed. Therefore, an optimum trade-off point is again expected.

5.2 Experiment 1

In Table 1, *Baseline_Z* indicates the design where no cache is added and external memory accesses are sequentialized by hand; *Z* represents the search window size. The performance columns are partitioned into two sub-columns. The left column corresponds to values for a search window size of 7 pels and the right column corresponds to 15 pels. A significant reduction of up to 50.6% in cycle count is seen for both *S_16_7_7* and *S_16_7_15*. However, due to degradation in the clock period, the execution time is reduced by at most 23.6% (*S_4_9_15*) for 15

pels. The maximum reduction in execution time for 7 pels *S_2_10_15* is 14.7%, for design *S_2_10_7*. Given that the number of cycles required to access data items in the cache is the same as the number of cycles used to access external memory, no significant benefit is observed in a cache with a single port. Indeed, designs *S_1_11_7* and *S_1_11_15* have larger cycle counts compared to *Baseline_7* and *Baseline_15* respectively because each cache miss results in an access time of 3 cycles (the additional cycle consumed over normal external memory access is due to the overhead of tag checking). It can be seen that a reduction in execution time can, however be obtained by parallelizing cache accesses. Also, there is an increase of approximately 52.8% in execution time, comparing the lowest execution time of both 7 and 15 pels, with an increase of search area by 76.5% for each reference block. This increase in resource usage and execution time represents a trade-off between motion vector quality and search window size. The resource usage for both window sizes is the same because they have the same data paths.

Table 1. Table of timing and resource usage for a fixed cache size for window sizes of 7 and 15 pels

Design	Period /ns		Cycle Count $/10^8$		Execution time /sec per frame		Slice Count	Block RAMs
	Z/pels		Z/pels		Z/pels			
	7	15	7	15	7	15		
Baseline_Z	29.5	29.7	5.78	13.5	17.0	40.2	1166	4
S_1_11_Z	32.8	35.8	5.93	13.9	19.4	49.7	1224	7
S_2_10_Z	34.1	30.4	4.31	10.1	14.7	30.7	1539	8
S_4_9_Z	41.9	43.9	3.47	8.13	14.5	35.7	2363	12
S_8_8_Z	51.4	52.1	3.06	7.15	15.7	37.3	4650	20
S_16_7_Z	62.5	62.9	2.85	6.67	17.8	41.9	10927	36

For the cache design, the number of *trysema* statements, N is equal to the number of *releasesema* statements, M. The slice count increases superlinearly with the number of ports, in line with the $O(N^2)$ prediction of section 4.

A Pareto-optimum trade-off curve between execution time and resource usage is shown in Figure 4. Resource usage is obtained by taking the larger of the proportions of block RAM and slice usage [19] as seen in (1). Note that each point on the graph represents a fully placed and routed design. The leftmost point of the trade-off curve shows the *Baseline* design and the number of ports increase from the left to the right. For 7 pels, beyond a port count of 4, there is an increase in execution time even when more resources are used due to clock period degradation, indicating that the designs are sub-optimal. For 15 pels, *S_4_9_15* does not lie on the Pareto-optimum curve because of the comparatively smaller clock period of *S_2_10_15*.

$$\text{Resource usage} = \max\left(\frac{B}{T_{\text{B}}}, \frac{S}{T_{\text{S}}}\right) \tag{1}$$

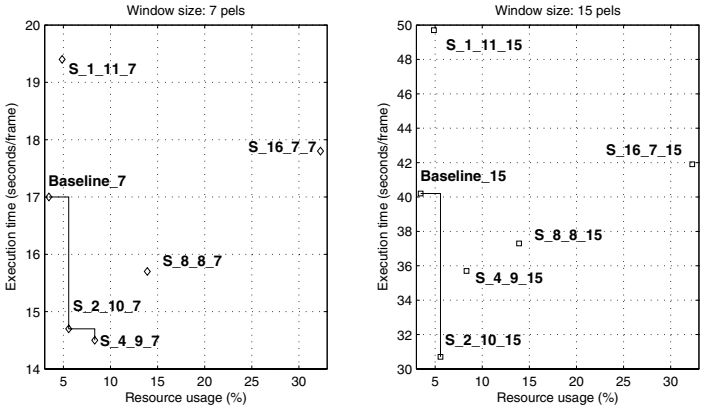

Fig. 4. Graph of execution time versus resource usage for different number of ports on a Xilinx XC2V6000 chip

$$
\begin{aligned}
B &= \text{Number of block RAMs used in the design} \\
T_{\mathrm{B}} &= \text{Total number of block RAMs on-chip} \\
S &= \text{Number of slices used in the design} \\
T_{\mathrm{S}} &= \text{Total number of slices on-chip}
\end{aligned}
$$

5.3 Experiment 2

In Table 2, the timing and resource usage information with varying number of cache lines are shown for a fixed port count of 4 and 2, for window sizes of 7 and 15 pels respectively. The number of cache lines is not extended beyond 2^{14} because the number of items in the cache exceeds the size of the image beyond that point. An optimum point is seen in the execution time where number of cache lines is 2^{10}. A block RAM is able to hold 2^{11} pixels, so no reduction of block RAM usage is seen below 2^{11} cache lines. However, a reduction of slice count still occurs. The number of data block RAMs for 15 pels is the same for 2^9 and 2^{10} cache lines for the same reason, but two additional block RAMs are required for 2^{11} cache lines to hold the tag and valid bits because of the fixed number of wordlength formats allowed in block RAMs.

The Pareto-optimum curve is shown in Figure 5. The number of cache lines increases with resource usage from the left to the right; For 7 pels, aside from *Baseline_7* and *S_4_11_7*, all other designs are clearly sub-optimal. *S_4_9_7* and *S_4_10_7* are sub-optimal because, by employing design *S_4_11_7*, execution time can be reduced without additional resource usage. This behaviour is attributed to the granularity of the FPGA platform; a block RAM has a storage capacity of 2^{11} pixels so that further reductions in the number of cache lines will still employ one block RAM. Further, designs not lying on the Pareto-optimum curve require more resources but require longer execution times because of clock period

Table 2. Table of timing and resource usage for fixed number of ports (Z refers to the window size in pels and X refers to the number of ports)

Design	Period /ns		Cycle Count /10^8		Execution time /sec per frame		Slice Count		Block RAMs	
	X=4, Z=7	X=2, Z=15	X=4, Z=7	X=2, Z=15	X=4, Z=7	X=2, Z=15	X=4, Z=7	X=2, Z=15	X=4, Z=7	X=2, Z=15
S_X_9_Z	41.9	32.3	3.47	10.2	14.5	33.1	2363	1531	12	8
S_X_10_Z	42.3	30.4	3.31	10.1	14.0	30.7	2374	1539	12	8
S_X_11_Z	40.8	33.2	3.28	9.68	13.4	32.1	2370	1544	12	10
S_X_12_Z	44.6	37.6	3.28	9.64	14.6	36.3	2386	1552	20	14
S_X_13_Z	49.0	37.3	3.28	9.62	16.1	35.9	2399	1556	36	18
S_X_14_Z	47.4	43.5	3.28	9.62	15.5	41.9	2404	1536	48	28

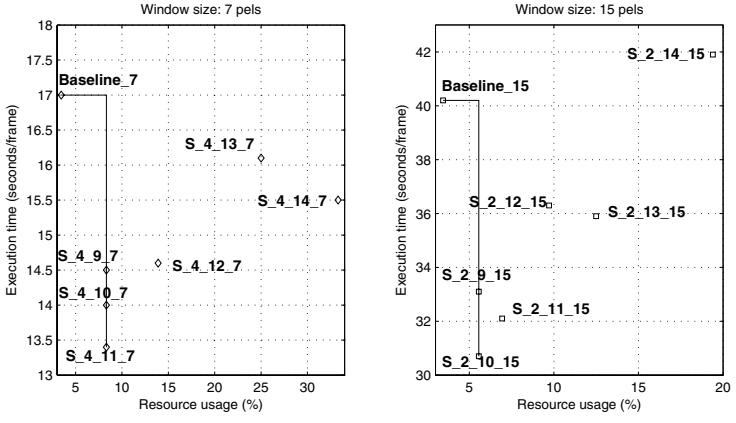

Fig. 5. Graph of execution time versus resource usage for different cache sizes on a Xilinx XC2V6000 chip

degradation. For a window size of 15 pels, the trade-off characteristic is similar. However, *Baseline_15* and *S_2_10_15* are optimal.

6 Conclusion

In this work, a novel multi-port caching scheme for circuits with parallel datapaths has been described. This scheme detects parallel accesses to cache contents dynamically and uses a semaphore-based system to sequentialize these accesses if they are targeted at the same sub-cache. This scheme requires minimal changes to the algorithm description. Significant savings of up to 51% and up to 24% in cycle count and execution time are seen, respectively, for a benchmark application. Further, it was verified in hardware that parallel sub-cache accesses were responsible for the cycle count reduction. However, degradation in clock speed reduces the extent of these gains. Due to the varying degree of clock degradation,

the savings are different for different window sizes. A 24% reduction in execution time is seen for a window size of 15 pels compared to 15% for 7 pels. In addition, beyond a specific number of ports and cache size, this degradation negates further reductions in cycle count, leading to an increase in execution time. Finally, the trade-off between resource usage and execution time were shown via hardware profiling. It has been explicitly shown that in the process of selecting Pareto-optimal designs, it is important to account for clock speed degradation. Indeed, considering cycle count reduction and resource usage alone are insufficent in the selection process.

Current and future work includes the investigation of the trade-off between energy consumption and resource usage. Also, trade-offs between dynamic and static memory accesses will be explored in greater detail. Potentially, more work could be done to tune the cache parameters during run-time to exploit trade-offs between resource usage and execution time to cater to statistical properties of the algorithm. However, re-configuration overheads have to be considered in determining the benefit and timing of re-configuration.

References

1. Issenin, I., Dutt, N.: Automatic generation of affine functions for memory optimizations. In: Proceedings of the conference on Design, Automation and Test in Europe. (2005) 808–813
2. Kandemir, M., Choudhary, A.: Compiler-directed scratch-pad memory hierarchy design and management. In: Proceedings of the Design Automation Conference. (2002) 628–633
3. Udayakumaran, A., Barua, R.: Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems. (2003) 276–279
4. Chalidabhongse, J., Kuo, C.: Fast motion vector estimation using multiresolution-spatio-temporal correlations. IEEE transactions on circuits and systems for video technology **7**(3) (1997) 477–488
5. Patterson, D.A., L.Hennessy, J.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco (1996)
6. Kulkarni, C., Catthoor, F., Man, H.: Data and memory optimization techniques for embedded systems. In: Proceedings of the IPDPS Workshops on Parallel and Distributed Processing. (2000) 186–193
7. Panda, P., Catthoor, F., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., Kjeldsberg, P.: Data and memory optimization techniques for embedded systems. IEEE Transactions on Very Large Scale Integr. Syst. **6**(2) (2001) 149–206
8. Ishihara, T., Fallah, F.: A way memoization technique for reducing power consumption in caches in Application Specific Integrated Procesors. In: Proceedings of the conference on Design, Automation and Test in Europe. (2005) 358–363
9. Nastaran, B., Park, J., Diniz, P.: A compiler analysis and algorithm for exploiting data reuse in configurable architectures with RAM blocks. In: Proceedings of the Field-Programmable Logic and Applications. (2004) 1113–1115
10. Guo, Z., Buyukkurt, B., Najjar, W., Vissers, K.: Optimized generation of datapaths from C codes for FPGAs. In: Proceedings of the conference on Design, Automation and Test in Europe. (2005) 112–118

11. Sohi, G., Franklin, M.: High-bandwidth data memory systems for superscalar processors. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. (1991) 53–62
12. Edmondson, J., Rubinfield, P., Bannon, P., Benschneider, B., Berstein, D., Castelino, R., Cooper, E., Dever, D., Donchin, D., Fischer, T., Jain, A., Mehta, S., Meyer, J., Preston, R., Rajagopalan, V., Somanathan, C., Taylor, S., Wolrich, G.: Internal organization of the Alpha 21164 a 300MHz 64-bit quad-issue CMOS RISC microprocessor. Digital Technical Journal **7**(1) (1995) 119–135
13. Page, I., Luk, W.: Compiling Occam into FPGAs. In: Proceedings of the Field-Programmable Logic and Applications. (1991) 271–283
14. Intel: (Understanding memory access characteristics of motion estimation algorithms) http://www.intel.com/cd/ids/developer/asmona/eng/182345.htm?page=2, accessed 1 October 2005.
15. Celoxica: (DK compiler) http://www.celoxica.com, accessed 1 October 2005.
16. Celoxica: (RC300 board) http://www.celoxica.com/rc300/default.asp, accessed 1 October 2005.
17. Xilinx: (Virtex 2 datasheet) http://www.xilinx.com/bvdocs/publications/ds031.pdf, accessed 1 October 2005.
18. Celoxica: (RC300 manual) http://www.celoxica.com/techlib/CELWO4110816VG-316.pdf, accessed 1 October 2005.
19. Bouganis, C.S., Constantinides, G., Cheung, P.Y.K.: A novel 2-D design methodology for heterogeneous devices. In: Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines. (2005) 1–10