Using Reconfigurable Hardware to Customize Memory Hierarchies

Peixin Zhong and Margaret Martonosi
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544-5263
{pzhong, martonosi}@ee.princeton.edu

Abstract

Over the past decade or more, processor speeds have increased much more quickly than memory speeds. As a result, a large, and still increasing, processor-memory performance gap has formed. Many significant applications suffer from substantial memory and communication bottlenecks, and their memory performance problems are often either too unusual or extreme to be mitigated by cache memories alone. Such specialized performance "bugs" require specialized solutions, but it is impossible to provide case-by-case memory hierarchies or caching strategies on general-purpose computers.

We have investigated the potential of implementing mechanisms like victim caches and prefetch buffers in reconfigurable hardware to improve application memory behavior. Based on technology and commercial trends, our simulation-based studies use a forward-looking model in which configurable logic is located on-chip with the CPU. Given such assumptions, our results show that the flexibility of being able to specialize configurable hardware to an application's memory referencing behavior more than balances the slightly slower response times of configurable memory hierarchy structures as compared to custom hardware implementations. For our three applications, small, specialized memory hierarchy additions such as victim caches and prefetch buffers can reduce miss rates substantially and can drop total execution times for these programs to between 60 and 80% of their original execution times. Our results also indicate that different memory specializations may be most effective for each application; this highlights the usefulness of configurable memory hierarchies that can be specialized on a per-application basis.

Keywords: memory latency, configurable computing, victim cache, prefetching.

1 Introduction

Due to the rapid increase in microprocessor speed, the performance gap between processors and main memory is widening. In response to this, cache memories are typically used in computer systems to bridge this performance gap and reduce the average memory access time. Although caches work well in many cases, they may still fail to provide high performance for certain applications.

Several hardware and software techniques have been proposed to improve cache performance in such cases. For example, prefetching techniques aim to hide the large latency out to main memory, and victim caches attempt to reduce conflict misses in low-associativity caches. These hardware techniques have variable results depending on the application's memory referencing behavior, and their disadvantage is that they represent wasted transistor space on the CPU chip for those applications where they are ineffective. On the other hand, the drawback to purely software-based techniques (such as blocked matrix accesses or compiler-inserted prefetching directives) is that it can be difficult to statically analyze a program's memory behavior. For these reasons, we have explored implementing memory hierarchy additions in *programmable* hardware.

Programmable logic, such as field-programmable gate arrays (FPGAs) have gained tremendous popularity in the past decade. Programmable logic is popular because a given chip's behavior can be configured and customized for different functions during different sessions. Customization on a per-application basis is feasible because the reconfiguration process is fast and can be done with the device in the system.

Some FPGAs can even be partially reconfigured when the rest of the device is in use.

While the configurability of FPGAs makes them heavily used for prototyping, they have also been used to build high-performance application specific compute engines [12]. In addition, there has been work (such as PRISC [21] and PRISM [2]) on supplementing conventional processors with configurable coprocessors to accelerate performance for different applications

Thus far, most configurable computing projects have focused heavily on *computation* as opposed to *data access*. In this paper, we explore the potential of using configurable hardware to make application-specific improvements to memory behavior. We envision a library of parameterized memory hierarchy additions that can be invoked to target cases where the cache system does not work well for a particular application.

As fabrication technology improves, more and more transistors fit on a single chip, and it seems likely that we will soon see processor chips that include a region of on-chip configurable logic. This configurable logic can clearly have many uses; our work does not preclude configuring the logic for more traditional compute-oriented uses, but simply attempts to explore an alternative use.

In Section 2, we will first discuss the structure of the configurable memory unit that we envision. Following that, in Section 3, we present case studies of applying the ideas of configurable memory to several applications. In Section 4, we discuss some of the hardware organization and implementation issues inherent in our approach. A brief account of related work is included in Section 5 and Section 6 presents some discussion and our conclusions.

2 Configurable Hardware in Memory Hierarchies

We believe that configurable logic can result in significant performance improvement by improving average memory access latencies. Our overriding goal is to minimize the number of misses that result in accesses to main memory. Researchers have proposed methods that promise performance improvements of up to 3X by reducing cache misses using full-custom hardware [19]; our current research shows the potential of these approaches using flexible, configurable hardware.

Integrated circuits are expected to soon grow to contain over 100 million transistors. As this growth takes place, we must determine ways to best make use of these additional transistors. Rather than simply devoting increased chip areas to increased cache sizes, our research explores other methods for using the transistors to reduce (or better tolerate) memory access latencies.

In current research projects, configurable logic is typically incorporated into the architecture using an attached processor array model. As shown on the left hand side of Figure 1, an accelerator based on FPGAs and dedicated static RAM (SRAM), is attached to the I/O bus of a conventional host processor. The conventional processor and configurable logic array operate asynchronously. The host supplies control signals and monitors results while the logic array processes data obtained from an external source such as a frame-buffer. The major problem with this model is the high communication latency between processor and configurable logic, due to their physical and logical separation.

As shown on the right hand side of Figure 1, the expected integration of configurable logic on-chip gives us more flexibility not only in its logical placement within the architecture, but also in its expected uses. In addition, on-chip configurable logic has more flexibility in its connectivity to processors and caches. We can now begin considering uses for configurable logic that are infeasible (because of latency and connectivity constraints) with the attached processor model.

2.1 Logical Placement, Connectivity

The logical placement of the configurable logic is driven by its intended memory optimization function. The right hand side of Figure 1 shows two distinct possibilities, and in fact, each of these configurable blocks may expend their transistor budgets on some combination of configurable gates and SRAM. (In this figure, the logical positions for configurable logic are indicated by diamonds labelled "C1" and "C2".)

The configurable logic closest to the processors can detect L1 cache misses and manage prefetch buffers, stream buffers, or a victim cache. Similar functions can be performed by the second block of configurable logic, for the L2 cache. In addition, this logic could observe memory accesses between nodes of a dis-

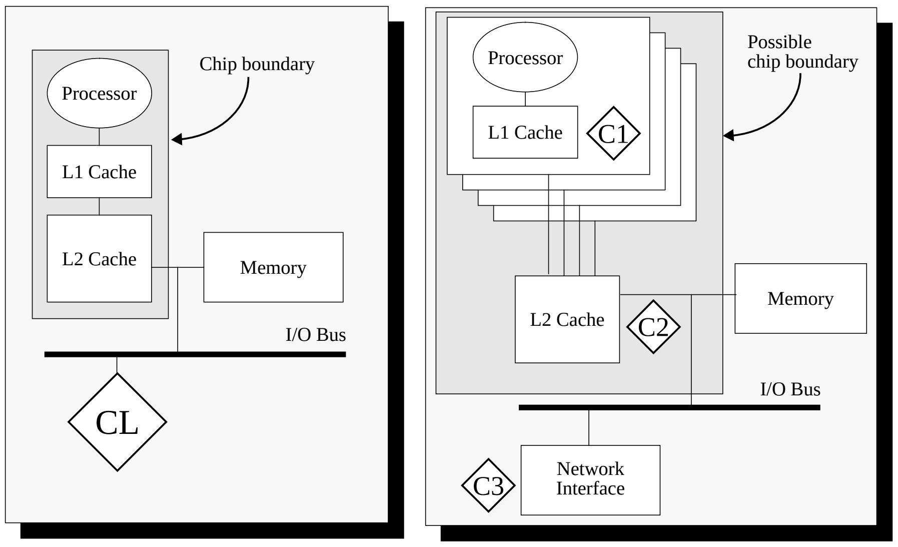

Figure 1: Traditional (left) and more forward-looking (right) uses of configurable logic.

tributed memory multiprocessor, or hold specialized multiprocessor coherence protocols.

In this paper, we focus primarily on two particular optimizations: victim caches and prefetching engines, implemented in configurable hardware. Victim caches are small, fully-associative caches that lie between the cache and main memory. Victim caches primarily aim to reduce conflict misses by caching data that has been evicted from the cache as a result of a miss on another location. Even small (four-entry) victim caches can reduce conflict misses by about 50% and reduce total cache misses by 5-30% [19].

Prefetching engines are hardware structures intended to fetch pieces of data from main memory before references to them occur, in order to hide the large latencies out to main memory. They have been studied in several contexts before [6, 7, 20]. Although such hardware mechanisms have been evaluated via simulation studies, widespread adoption in custom hardware is unlikely, since their benefits vary from application to application. Assuming that future processors will include a block of configurable logic on-chip, we have used simulations to re-evaluate these mechanisms within the context of a dynamically customizable memory hierarchy.

For prefetch or stream buffers, the configurable logic must be able to detect a miss from the L1 (or L2) cache, recognize that the address matches the address of data in the prefetch buffer, abort the memory request, and supply the data to the CPU. Simultaneous to servicing the access request, the prefetch controller may initiate a memory request for the next location from which to prefetch. For a victim cache, the logic must once again be able to detect the primary cache miss, recognize the presence of the data in its SRAM, abort the memory request, and supply the data to the CPU. These functions can be easily performed using a small state machine controller.

2.2 Logical vs. Physical Placement

It is important to make a distinction between the logical and the physical placement of the configurable logic. The discussion above described the logical placement, i.e. where the configurable logic needs to be positioned in terms of the signals it needs access to. Physically, we anticipate future generations of microprocessors to have a fixed amount of configurable logic which may be used for a variety of tasks including but not limited to accelerating computation and memory accesses. Since each application will have different requirements on where the configurable logic should be placed, it is not reasonable to expect it to be positioned in exactly the positions marked in Figure 1. The key demand that our *logical* placement makes on configurable logic's *physical* placement is that of connectivity; there must be signal paths that provide access to the signals required by the logical usage. The physical placement impacts performance in two ways. First, the interconnect delay may be significant and second, the interconnect topology may lead to contention of shared resources such as buses. This contention has not been modelled in the initial study presented here.

2.3 Programming Model

In addition to hardware concerns, our configurable memory hierarchies raise software issues as well. Figure 2 gives a flowchart that represents the path a program would go through in order to make use of configurable memory hierarchy additions.

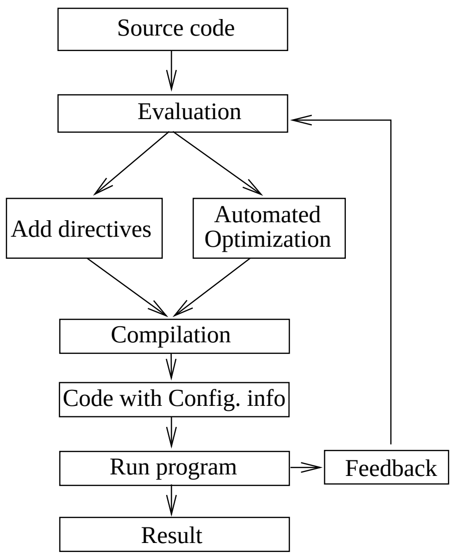

Figure 2: Flowchart of programming model with configurable memory hierarchy additions.

Starting from conventional source code, some performance analysis is needed to determine what types of performance bottlenecks are present. Based on this analysis, configuration directives may be added into the code in order to use particular configurable memory hierarchy additions. The configuration directives may either make use of hardware from a parameterized library of pre-made designs, or they may specify custom-made configurations. Our flowchart also allows for the possibility that these configuration directives will either be hand-inserted into the code by the programmer, or will be automatically inserted as part of the compilation process. At this point, the code is executed, and the memory hierarchy additions are configured as part of the program run. If the code is to be run repeatedly, then the performance statistics about this program run can be used as feedback to modify the configuration for future runs. Clearly, the widespread use of this sort of hardware relies on automating the analysis and choice of configurations as much as possible. In Section 3's preliminary case studies, however, we set the configuration directives manually.

3 Application Case Studies

In order to quantitatively evaluate our idea, we present results on three case study applications.

3.1 Configurable Memory Hardware

In the case studies presented here, we assume the configurable memory hardware sits on the same chip as the processor and first-level (L1) cache. This corresponds to location C1 in Figure 1. For these studies, we examine two configurable hierarchy additions separately: a victim cache and a prefetch buffer.

3.1.1 Victim Cache

A victim cache is a fully-associative cache with typically no more than 16 entries. When an L1 cache miss occurs, the referenced line is pulled into the L1 cache and another line currently stored in the same cache line is evicted. The victim cache is a small fully-associative buffer that holds these evicted lines. Using victim caches can have the effect of increasing the set associativity at low hardware cost.

Figure 3 shows a diagram of the victim cache we consider. The address and data lines go to both the L1 cache and the victim cache. On an L1 cache hit, the data is provided directly from the L1 cache, and the configurable hardware is not involved. On an L1 cache miss, the victim cache is probed in parallel with the request to the next level of memory. If the reference hits in the victim cache, the victim cache data provides to the processor. When the data is pulled into the L1 cache, another line must be evicted from the cache. The victim cache intercepts this line and stores it.

3.1.2 Prefetching Buffer

The goal of a prefetch buffer is to initiate main memory accesses in advance, so that the data will be closer

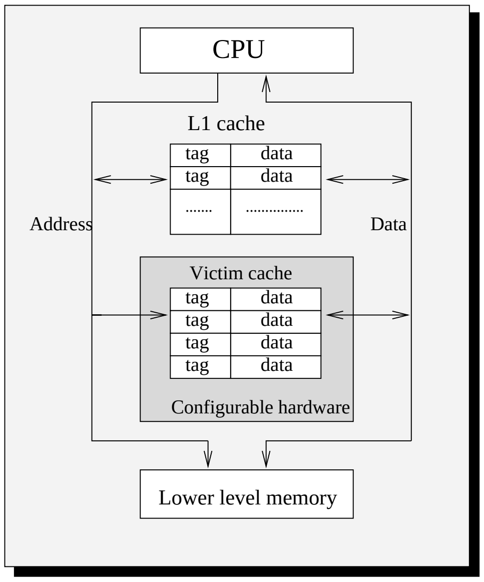

Figure 3: Block diagram of victim cache.

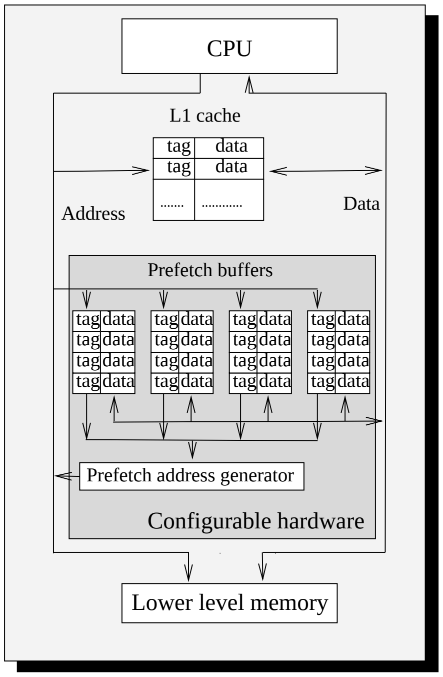

Figure 4: Block diagram of prefetch buffers.

to the processor when referenced. Prefetching can be especially beneficial when the main memory bandwidth is large, the access latency is also quite high. The prefetch buffer in our model has several independent "slots". Each slot holds several cache lines and works like a FIFO. It can issue prefetching commands to the main memory. We need several slots because in a typical program, there will be several concurrent data access streams. For example, when a program is performing matrix addition, there will be three access streams for the two source arrays and the destination array. Figure 4 is the schematic of prefetch buffer with four slots and each slots holds four cache lines of data.

If there is an L1 cache hit, it does not trigger any operation in the prefetch buffer. If there is an L1 miss, the request is sent to the prefetch buffer. If the referenced item is available in one of the slots, this line is pulled into the CPU as well as the L1 cache. The lines below this line are filled into the top lines. The vacancy is filled with data prefetched from main memory. The prefetching engine must determine which line to prefetch next. In the simple case, the memory address for the subsequent line is calculated by incrementing the current address by the cache line size.

If the L1 cache miss also misses in the prefetch buffer, the least recently used slot is designated to begin prefetching the subsequent data following this referenced address. By doing this, we can avoid compulsory cache misses if the initial reference to a memory line is already covered by the prefetching.

To make better use of the slots, we may program them with address ranges and data access strides. For example, when we have a lot of accesses on one array, we can set one slot with the address range of the data array. The slot will only respond to references in this range. If the access pattern has a address increment stride larger than the cache line size, we can assign the stride so we only prefetch the data that is useful.

3.2 Evaluation Methodology

In this study, we use software simulations to estimate the performance for our applications. In particular, we used the TangoLite memory reference generator to perform high-level simulations. TangoLite uses a direct-execution simulation approach. The original program written in C or FORTRAN is compiled and instrumented with calls out to routines that simulate memory behavior and keep statistics.

In our model, we have single CPU that issues no more than one instruction per cycle. The direct-mapped L1 data cache has a capacity of 8 KB and a line size of 32 bytes. In order to focus on data references, we have assumed ideal memory performance in the instruction stream.

Our simulations assume a hit in the L1 cache costs one cycle, the same as a simple ALU instruction. If a reference misses in the L1 cache, additional cycles are needed depending on where the data is found. We assume that data found in the configurable cache takes 2 extra cycles and a miss in both L1 and configurable cache will cost 10 extra cycles to go to main memory.

The victim cache we simulate is fully-associative with 4 cache-line-sized entries, update according to an LRU policy. The prefetch buffer has four slots with 4 cache lines each. Our simulator ignores any additional contention that prefetching may introduce.

3.3 Per-Application Results

With that background, we will now look in more detail at three applications. Our goal is to see the potential performance improvement available by implementing memory hierarchy additions in configurable hardware. The three applications studied include two kernel applications known for their poor cache performance (Matrix Multiplication and Fast Fourier Transform) as well as a more substantial program: the Tomcatv benchmark from the SPEC92 suite [9].

3.3.1 Matrix Multiplication

In the first example, we study a matrix multiplication program multiplying two 100 by 100 matrices. The elements are double-precision floating point. This means that a total of 80000 bytes are needed for each matrix. This greatly exceeds the size of the L1 cache. Clearly the calculation involves two source matrices and one destination matrix. As shown in Figure 5 one of the source matrices is accessed in sequential column order while the other is accessed in sequential row order.

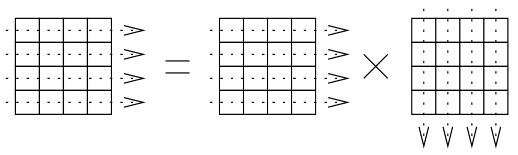

Figure 5: Memory access pattern for arrays in matrix multiplication.

Results The simulated performance of 100x100 matrix multiplication is summarized in Table 1. The standard cache does not perform well both because of the size of the data set and also because one of the arrays is accessed down the columns. Overall, the data miss rate is 21%.

The second column of the table shows the new results obtained when a configurable victim cache is used. In this case, some of the conflict misses can be avoided. The overall performance is only slightly better than in the original case though. The third column of the table shows the results when a configurable prefetch engine is used. In this "stride-one prefetching" each prefetch slot always attempts to prefetch the next cache line. This prefetching allows the program to avoid some compulsory misses, but does not dramatically improve overall performance. The difficulty here is that one of the arrays has a columnar memory access pattern that does not benefit from the stride-one prefetching.

Since simple stride-one prefetching is not fully effective, we also investigated allowing the prefetching engine adjust its prefetch stride on a per-slot basis. The fourth column shows the results of this experiment. Here, we have allocated one slot to each matrix. For the source matrix that accesses elements across a row, the prefetch stride is still one cache line. For the source matrix whose elements are accessed down the columns, we set the stride to be the size of one row in the matrix. Every reference to an address within the matrix will only update prefetch buffer for its matrix. This technique yields much better performance. The miss rate is reduced by 15% and (as shown in Figure 6 the total program execution time is reduced to 61% of the original.

3.3.2 Fast Fourier Transformation

The next example we examine is Fast Fourier Transformation (FFT), a commonly used algorithm in signal processing. FFT's access pattern is not as regular as other matrix computations; the butterfly memory access pattern is shown in Figure 7 for an 8-point radix-2 FFT. In this figure, the data is prepared in bit-reversed order and the result is in normal order. The calculation is done in-place, i.e. only one data array is used for source, intermediate and destination. The computation is done in dual pairs as follows:

$$X_{m+1}(p) = X_m(p) + W_N^r X_m(q)$$

	Orig. Cache	Victim Cache	Stride-One Prefetch	Column Prefetch
L1 cache hit rate	79%	79%	79%	79%
Config. cache hit rate	–	2%	2%	15%
Miss rate	21%	19%	19%	7%
Memory Lat. (10^6 cycles)	4.2	3.9	3.9	1.7
Exec. Time (10^6 cycles)	6.2	5.9	5.9	3.8

Table 1: Performance results for matrix multiplication.

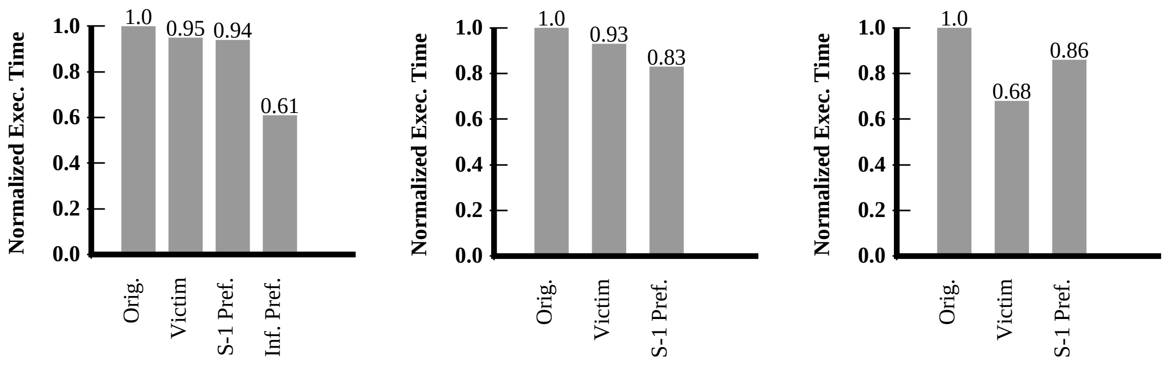

Figure 6: Execution times with different configurable hierarchy additions, normalized to execution time with original cache hierarchy.

$$X_{m+1}(q) = X_m(p) - W_N^r X_m(q)$$

where

$$W_n = e^{-j(2\pi/N)}$$

is the coefficient which is precomputed in our program. The subscript of X stands for the stage of computation and the number in bracket stands for the position in the data array.

For this paper, we examine a 1024-point complex FFT. Since there are 1024 points and each point needs 2 doubles, a data array of 16 KB is involved. The corresponding coefficients are precalculated and stored as an 8KB array. The calculation requires 10 stages. The whole data array is referenced in each stage and the memory access is in dual pairs as shown in 7. For an 8KB direct-mapped cache, the misses come from conflicts between the data and coefficient

	Orig. Cache	Victim Cache	Stride-One Prefetch
L1 cache hit rate	78%	78%	78%
Config. cache hit rate	–	6%	14%
Miss rate	22%	16%	8%
Memory Lat. (10^3 cycles)	96	76	47
Exec. time (10^3 cycles)	293	273	244

Table 2: Performance results for FFT.

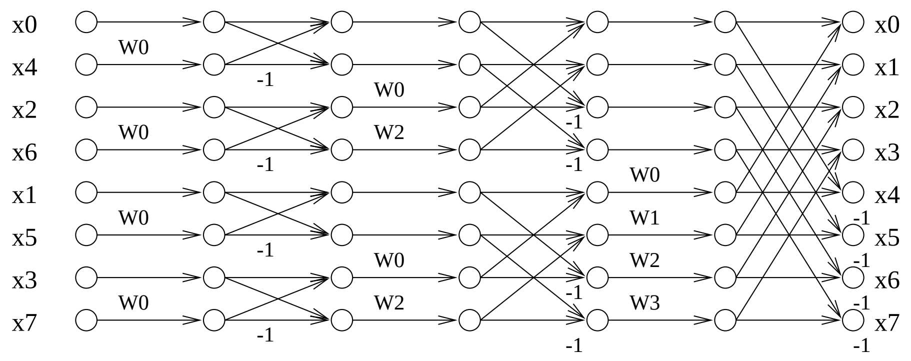

Figure 7: Butterfly access pattern in FFT.

arrays.

Results Table 2 shows the simulation results for FFT. As before, the first column shows the results with the original memory hierarchy. Here, the data miss rate is 22%. Unlike in matrix multiply, the victim cache is fairly effective here in improving FFT's memory performance. The program's miss rate drops to 16%. This is due to the significant data reuse in loops and the reduction of conflict misses.

With stride-one prefetching the program's performance is improved even more. This is because of the good spatial locality and the sequential access pattern of computing the dual pairs. The miss rate in this case is reduced by 14% and the total execution time is reduced to 83% of the original cache. In each stage, there is one reference sequence (i.e. one slot is used) if the dual pair stride is less than a cache line. In later stages, the dual-pair stride will be more than four cache lines (the size of one prefetch slot), and in those cases, two slots are used. In intermediate stages (i.e. the stride falls between one line and four lines) the access to the dual pair may cause the prefetch buffer to be updated prematurely and useful data already in the prefetch buffer is lost. More sophisticated prefetching may avoid this problem but may be too complex to be implemented by reconfigurable hardware.

3.3.3 Tomcatv

Tomcatv, a SPECfp92 benchmark, is a vectorized mesh generation program written in Fortran. There are 7 data arrays used in the program. Each array is a 257 by 257 matrix of double precision floating point. Thus, the data arrays take about 3.7MB each, and are far larger than the cache size. The arrays are mainly accessed sequentially in row order and data elements are reused several times in each loop. Due to the large active data set, however, the miss rate is still quite high.

	Orig. Cache	Victim Cache	Stride-One Prefetch
L1 cache hit rate	67%	67%	67%
Config. cache hit rate	–	19%	8.3%
Miss rate	33%	14%	25%
Memory Lat. (10^6 cycles)	912	7497	739
Exec. Time (10^6 cycles)	1313	898	11306

Table 3: Performance results for tomcatv.

Results Table 3 shows the results for tomcatv. Although the active data set is very large, there is a good deal of spatial and temporal locality in the memory accesses. Thus, it may be somewhat surprising that the original miss rate is as high as 33miss rate is due to a large number of conflict misses in the direct-mapped cache. As shown in the second column of the table, the victim cache gives very good results on this application by greatly reducing the conflict misses. The total miss rate drops to 14% and execution time is reduced to 69% of its original value.

The prefetch buffer, however, does not do well in this case because the repeated access of nearby data causes items to be prematurely updated in the prefetch buffer. Prefetching still reduces the miss rate to 26% though, and reduces execution time to 86% of its original value.

3.3.4 Summary

Overall, this section has shown that relatively simple per-application additions to the memory hierarchy can improve performance significantly. For the three applications studied, total execution time was reduced by 17% to 39%. In the most "realistic" application of the three, tomcatv, execution time is reduced by 31% by adding a small victim cache. It is interesting to note that different applications got their best performance from different hierarchy additions; this helps confirm our belief that *application-specific* memory hierarchy additions can be useful in customizing hardware to improve performance.

Although our case studies were relatively limited in which hardware options were explored, other mechanisms are possible as well. For example, if we start with a prefetch buffer but configure it to keep the line that has just been accessed in the prefetch buffer (as well as forwarding it to the L1 cache), then the prefetch buffer also serves as a miss cache [19]. Jouppi has shown that miss caches can be moderately useful in reducing misses due to cache conflicts. Another possible embellishment on the features evaluated here would be to add complexity to the prefetching mechanism to allow for more elaborate address calculations. Evaluating these additional features is a topic for future work.

4 Hardware Implications

Having shown potential performance improvements from configurable memory hierarchy customizations, this section will delve further into some of the related hardware and implementation details.

First, due to our connectivity requirements, our idea clearly assumes that configurable logic will reside on the CPU chip itself. This assumption is borne out by commercial trends that indicate that combinations of CPU and configurable logic should be widely available in the next decade.

In order to make the remaining hardware discussion more concrete, we will couch them in terms of technology similar to that found in a Xilinx XC4000-series FPGA. [24] This class of FPGAs uses 4-input lookup tables to implement logic functions. In 4000-series FPGAs, a configurable logic block (CLB) includes two 4-input LUTs. When these two are combined, any 5-input function and some 9-input functions can be implemented. Each LUT can be used as a 16x1 bit SRAM as well. In addition to the lookup tables, each CLB has two registers that can used independently of the LUTs. Based on this information, Table 4 shows the number of CLBs required to implement certain basic functions. Although our discussion gives calculations on a per-hardware-structure basis, these estimates compare well with CLB estimates given by automated synthesis from VHDL down to Xilinx parts.

Function	Number of CLBs
4:1 multiplexer	1
16:1 multiplexer	5
16x2 bit memory	1
4 bit comparator	1

Table 4: CLB counts for common logic functions.

Using that basic information as a starting point, Table 5 gives a breakdown of how much hardware is required for the victim cache and prefetch buffer described in the previous section. First, consider a victim cache with four 32-byte cache lines. To implement it, we will need SRAM to store 4x256 bits of data. Assuming a 27 bit tag and 1 valid bit per-entry, we need 112 bits to store the four tags and valid bits. To minimize the latency of the cache lookup, we want to be able to access all four victim cache entries simultaneously. Thus, we need four comparators to decide whether one of the lines holds the requested data. The output of the comparators is encoded into multiplexer control signals, and the multiplexer then selects the requested cache line. In our design, the tag bits use only registers, not lookup tables, so they need not contribute to the CLB count. (We can make use of unused registers from other functions that required only LUTs.) All told, this design requires slightly over 400 CLBs. Current 4000-series parts are already more than large enough to hold this sort of design. (For example, a Xilinx XC4025 has 1024 CLBs).

Based on Figure 4, it might appear as though the 4-slot/4-entry-per-slot prefetch buffer would require significantly more hardware than the victim cache. Because the prefetch buffer can use the CLBs more

Function unit	Num. units	Num. CLBs
Data memory	256x4 bits	128
Tag memory	112 bits	56 (reg. only)
Comparator	4x28 bits	33
4:1 Mux.	256 bits	256
Total		417

Table 5: Hardware breakdown and CLB count for victim cache.

efficiently, however, we find that its CLB count is only slightly higher than that of the victim cache. Although we are storing four times as much data in the prefetch buffer, the prefetch buffer uses the same number of CLBs for data storage, because it uses the LUTs more efficiently. 448 bits of tag are used to identify the 16 entries in the prefetch buffer and 16 comparators are required to identify the correct entry. For the stride-one prefetching, a 27-bit adder is needed to determine the next cache line to prefetch. Overall, the prefetch buffer needs slightly over 550 CLBs.

Function unit	Num. units	Num. CLBs
Data memory	256x16 bits	128
Tag memory	448 bits	224 (reg. only)
Comparator	16x28 bits	129
Adder	27 bits	14
4:1 Mux.	283 bits	283
Total		554

Table 6: Hardware breakdown and CLB count for prefetch buffer.

The next hardware issue to consider is connectivity. Typical FPGAs have more than enough routing resources for functions like our victim cache or prefetch buffer. The real issue concerns connectivity between the CPU and the configurable logic. Since we expect that the configurable logic will also be expected to sometimes serve as a processing unit, it is likely to have a data path that allows for 2 operands to arrive from the CPU simultaneously. For a 32-bit processor, this means a pathwidth of 64 bits. If the unit is used for configurable memory, then this connection would be wide enough for both a 32-bit address and 32-bit data operand to be sent between the CPU and the configurable caching structures. Therefore, adding cache functionality to a configurable processing unit does not necessarily increase its connectivity requirements with the CPU core. In the case of prefetching however, configurable prefetch buffer does, however, require additional connections to lower-level caches or memory in order to prefetch data.

Overall, the point of this discussion is to show that configurable memory hierarchies place relative modest hardware requirements on the configurable hardware and its connection to the CPU. Given that configurable hardware is likely to be co-located on-chip with CPUs in the near future, both our performance results and our hardware results indicate that configurable memory hierarchies are likely to be a useful area for further study.

5 Related Work

Our work represents a convergence of two areas of ongoing research. This section first describes related configurable computing research. We then discuss memory hierarchy research that guided our choice of applications and hierarchy customizations to be incorporated in configurable logic.

5.1 Configurable Computing

To date, many research efforts that use FPGAs as configurable computing resources have focused on system architectures, tools, and algorithms to accelerate end user applications. In most cases, researchers have identified particular types of compute-intensive applications, and have used configurable hardware to accelerate computation for those application domains. Examples of this include systolic computations [12, 15, 13], image processing [1, 5, 10], and video compression [18].

Other researchers have focused more broadly on the acceleration of C code [2]. In this work, Athanas *et al.* worked to *automate* the mapping to configurable hardware using a compiler. Finally, another category of research projects have used FPGAs to emulate conventional microprocessors, as in the Spyder project [17].

In contrast to these compute-oriented approaches, we have investigated accelerating *memory access* in microprocessors. To our knowledge, no other projects have directly attacked this problem, but a handful of projects are relevant to this work. Shirazi has studied the emulation of floating point instructions on FPGAs [22]. Although not focused on memory system behavior, this work had a "data-oriented" angle, since it aimed to specialize data types (such as shortening floats) to the operations the program performed. The ArMen group [4] used programmable logic to control concurrency in a Transputer-based multiprocessor, by implementing the network interface in configurable logic. Finally, DeHon *et al.* have explored mechanisms for integrating configurable logic with microprocessors [8].

5.2 Reducing and Tolerating Memory Latency

In addition to building on research from the configurable computing domain, our work also draws from extensive prior research in memory latency reduction and tolerance techniques. While some previously proposed structures may not have offered the across-the-board performance improvements necessary to warrant inclusion in custom hardware, building structures in configurable logic allows us to draw from this pool of proposals and specialize our selections for particular applications.

Prefetching, for example, is an area that has undergone extensive prior research. Some work has suggested particular hardware structures to aid in prefetching [19, 6, 7, 3]. Hardware structures for prefetching the instruction stream have shown broad enough performance improvements to warrant inclusion on current commercial processors [11]. For the data stream, however, hardware prefetching has shown mixed results across applications. Thus, our work suggests that prefetching structures could be implemented in configurable hardware only when useful.

Other work has also examined issues in prefetching under software (compiler) control [20, 23]. As with hardware prefetching, performance improvements due to compiler-directed prefetching can be significant, but approaches are very application-dependent [14]. Compiler-directed prefetching often has difficulty performing the pointer analysis required to identify which program references are likely to miss, in order to be selective in flagging references to prefetch. In other cases, such decisions of which references to prefetch are highly data-dependent, and can only be effectively identified at runtime [16]. Our work demonstrates the promise of combining runtime monitoring and prefetching support in order to selectively and effectively prefetch data references for these difficult-to-analyze programs.

6 Conclusions

This paper has described a preliminary exploration of the potential of using configurable logic on the CPU chip to improve average memory access times. We feel that it is very likely that configurable logic will be integrated onto some CPU chips in the near future. While there has been extensive research on compute-oriented uses of this configurable logic, there has been little evaluation of more memory-oriented applications. Our work represents a first step towards such evaluations.

Prior work has proposed a wide array of memory hierarchy additions that offer performance improvements on some applications. While many of these special hierarchy features are fairly small and easy to build, they are often not implemented in commercial processors because they do not promise improved performance to a wide-enough array of applications. Configurable memory hierarchy additions are promising because we can use the configurable logic for different hierarchy additions on a per-application basis.

Our simulation-based work has shown that these additions can improve application performance by 20 to 30% even after accounting for the fact that the configurable logic will be slower than custom-made counterparts. Their hardware requirements are fairly modest; the victim caches and prefetch buffers we evaluated required roughly 417 and 554 CLBs respectively.

Overall, our study provides a first look at the issues and promise inherent in memory-oriented uses of configurable logic for general-purpose computing. We feel that a broader application set and the availability of configurable logic on CPUs for experimentation will spur more innovations along these lines.

References

[1] P. Athanas and L. Abbott. Real-Time Image Processing on a Custom Computing Platform. *IEEE Computer*, 28(2):16–24, Feb. 1995.

[2] P. M. Athanas and H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, Mar. 1993.

[3] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. Supercomputing '91*, pages 176–186, Nov. 1991.

[4] C. Beaumont. Using FPGAs as control support in MIMD executions, Field Programmable Logic and Applications. In *FPL '95*, pages 94–103, Aug. 1995.

[5] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to Programmable Active Memories. Technical Report 3, DEC Paris Research Lab, June 1989.

[6] P.-Y. Chang and D. R. Kaeli. Branch-directed Data Cache Prefetching. 4th ISCA Workshop on Scalable Shared-Memory Multiprocessors, Apr. 1994.

[7] F. Dahlgren and P. Stensrom. Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. 4th ISCA Workshop on Scalable Shared-Memory Multiprocessors, Apr. 1994.

[8] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *FCCM '94*, Apr. 1994.

[9] K. M. Dixit. New CPU Benchmark Suites from SPEC. In *Proc. COMPCON*, Spring 1992.

[10] P. Dunn. A Configurable Logic Processor for Machine Vision. In *Field Programmable Logic and Applications, FPL '95*, pages 68–77, Aug. 1995.

[11] J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, et al. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Tech. Journal*, 7(1):119–135, 1995.

[12] M. Gokhale, W. Holmes, A. Kopser, et al. Building and Using a Highly Parallel Programmable Logic Array. *IEEE Computer*, 24(1):81–89, Jan. 1991.

[13] M. Gokhale and B. Schott. Data Parallel C on a Reconfigurable Logic Array. *Journal of Supercomputing*, 9(3):291–313, 1995.

[14] A. Gupta, J. Hennessy, K. Gharachorloo, et al. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. 18th Int'l Symp. on Computer Architecture*, May 1991.

[15] D. T. Hoang. Searching Genetic Databases on Splash 2. In *FCCM '93*, pages 185–191, Apr. 1993.

[16] M. Horowitz, M. Martonosi, T. Mowry, and M. D. Smith. Informing Memory Operations: Providing Performance Feedback in Modern Processors. In *Proc. 23rd Int'l Symp. on Computer Architecture*, May 1996.

[17] C. Iseli and E. Sanchez. Spyder: A Reconfigurable VLIW Processor using FPGAs. In *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 17–24, Apr. 1993.

[18] C. Jones, J. Oswald, B. Schoner, and J. Villasenor. Issues in Wireless Video Coding using Run-time-reconfigurable FPGAs. In *FCCM '95*, Apr. 1995.

[19] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. 17th Int'l Symp. on Computer Architecture*, May 1990.

[20] T. C. Mowry. *Tolerating Latency Through Software-Controlled Prefetching.* PhD thesis, Stanford University, Dec. 1993.

[21] R. Razdan, K. Brace, and M. D. Smith. Prisc software acceleration techniques. In *Proc. Int'l Conf. on Computer Design*, pages 145–149, Oct. 1994.

[22] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. In *FCCM '95*, Apr. 1995.

[23] D. M. Tullsen and S. J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proc. 20th Int'l Symp. on Computer Architecture*, May 1993.

[24] I. Xilinx. *The Programmable Logic Data Book.* Xilinx, Inc, San Jose, CA, 1994.