Improved Dynamic Cache Sharing for Communicating Threads on a Runtime-Adaptable Processor

Joost Hoozemans* Arthur Lorenzon† Antonio Carlos Schneider Beck‡ Stephan Wong§
Computer Engineering Lab
Delft University of Technology
Email: {j.j.hoozemans*, j.s.s.m.wong§}@tudelft.nl
Institute of Informatics
Universidade Federal do Rio Grande do Sul
Email: {aflorenzon†, caco‡}@inf.ufrgs.br

***Abstract*—Multi-threaded applications execute their threads on different cores with their own local caches and need to share data among the threads. Shared caches are used to avoid lengthy and costly main memory accesses. The degree of cache sharing is a balance between reducing misses and increased hit latency. Dynamic caches have been proposed to adapt this balance to the workload type. Similarly, dynamic processors aim to execute workloads as efficient as possible to being able to balance between exploiting Instruction-level parallelism (ILP) and Thread-level parallelism (TLP). To support this, they consist of multiple processing components and caches that have adaptable interconnects between them. Depending on the workload characteristics, these can connect them together to form a large core that exploits ILP, or split them up to form multiple cores that can run multiple threads (exploiting TLP). In this paper, we propose a cache system that is able to further exploit this additional connectivity of a dynamic VLIW processor by being able to forward cache accesses to multiple cache blocks while the processor is running in multi-threaded ('split') mode. Additionally, only requests to global data are broadcasted, while accesses to local data are kept private. This will improve the hit rates similar to existing cache sharing schemes, but reduce the penalty due to stalling the other subcores. Local accesses are recognized by distinguishing memory accesses relative to the stack frame pointer. Results show that our cache exhibits similar miss rate reductions as shared caches (up to 90% and on average 26%), and reduces the number of broadcasted accesses by 21%.**

I. Introduction

Modern computing system rely heavily on caches to deliver the bandwidth needed to achieve high performance. As the memory gap has been steadily increasing, the cache hierarchy has become increasingly important. The challenge is that different programs have widely different requirements on the caches. This becomes worse when considering multi-threaded workloads; some of these need large amounts of inter-thread communication, benefiting from large caches that are shared with neighboring processing cores, while others benefit more from small private caches that have lower latency compared to the large shared caches. On a shared memory multicore platform with a single level of cache, running a multi-threaded program requiring a large amount of communication will result in excessive bus bandwidth utilization and main memory accesses. This can impede performance and scalability. Furthermore, in embedded systems, it increases energy utilization considerably, as main memory accesses consume more energy than cache accesses and will typically stall the processor for a large number of cycles. This is one of the reasons why designers have started to add large shared cache levels to the system.

Shared caches require multiple access ports or banking to be able to handle multiple requests simultaneously. This means they will likely require multiple cycles to handle requests and will utilize more area with respect to private caches that can be smaller and only need to service a single request at a time. In other words, the degree of sharing is a design trade-off. This is why a large body of work has been dedicated to studying cache sharing and partitioning. Even in contemporary systems with multiple levels of cache, this is still relevant as contention for the shared Last-Level Cache (LLC) has impact on the total performance [1].

Besides the memory system, workloads also have varying requirements on the processing fabric. Some programs are very thread-parallel by nature, others are more sequential but may have a high degree of ILP (Instruction-Level Parallelism) that can be exploited by wide-issue processors. Ideally, one would like to design or select processors and caches specifically for an application at design-time, to allow for design-time optimization to be performed. However, with general-purpose platforms (and modern, high performance embedded systems such as mobile phone SoCs), this is not possible as they need to be able to perform well on a wide range of programs not known at design time. That is why researchers have introduced processors and caches that can change their characteristics *dynamically*, at run-time [2] [3]. This means that the processor can operate as a single, wide-issue core to exploit ILP or as multiple smaller cores that can run multiple threads in parallel. Dynamic processors require caches that can facilitate all the different possible core configurations. When the processor is running in a single-core configuration, it should still be

Part of this work has been supported by *Omitted for blind review*

able to access all of the available cache blocks. This requires additional connectivity between the processing elements and the cache blocks (see Figure 1).

This additional connectivity leads to the following drawback of both 'classical' dynamic caches (targeting static multiprocessor systems) and dynamic processors: it results in longer cycle times or hit latency [3]. In return this should provide higher hit rates (in case of dynamic caches) and/or a higher degree of flexibility to adapt to the workload (in case of dynamic processors). Additionally, when a dynamic processor is running in multi-core mode, it allows cache requests to be forwarded to multiple blocks (because the connections are already in place to support the dynamic adaptability). In this case, the forwarded request will introduce contention, because the blocks will either be able to service a single request per cycle (stalling the core if it is trying to access memory at the same time) or need more access ports (increasing area usage, cycle time and/or latency). In this paper, we propose a scheme to alleviate this penalty.

As discussed, using shared cache schemes will only increase hit rates when accessing data that may be present in a neighboring cache block. This means that the target application domain is workloads with threads that have a high degree of communication between them. On Symmetric Multi-Processing (SMP) systems, threads usually communicate by simply operating on globally shared data in the same address space (although message passing and other schemes exist). However, a thread does not only use global (shared) data, it also references local (private) data such as the stack. Sending these memory requests to other cache blocks will never result in an advantage, as private data is in principle never needed by other threads.

In this paper, we propose a scheme where a dynamic processor can broadcast only *global* memory requests to its cache blocks, while keeping *local* accesses private to its own first level cache. This means that local accesses do not suffer from the penalty of a shared cache, while global accesses can benefit from being able to access shared data more efficiently. There are several ways the processor can identify the difference between these types of accesses; the most obvious way is by introducing an instruction set extension that makes a distinction between the 2 types in its memory instructions (e.g., Private_Load/Shared_Load). These can be supported by the compiler, as it knows which variables are shared or private. The instruction set architecture of the run-time adaptable processor used in this work, ρ-VEX, uses a specific register as stack pointer (similar to ARM and x86). In this case, the processor inherently uses the stack pointer for local memory accesses (i.e., variables that are on the stack) and an instruction set extension is therefore not necessary.

The contributions of this paper are:

- We propose an improvement to the cache system of a dynamic processor that sends *global* memory accesses to multiple blocks ("broadcasting") while keeping *thread-local* memory accesses private.
- We show that for multi-threaded workloads, broadcasting only global cache requests can significantly decrease the number of accesses to main memory, equivalent to results obtained when broadcasting all cache requests.
- We show that, by not broadcasting local cache requests, the penalty for this dynamic cache block sharing can be reduced.
- We examine and discuss the effects cache sharing has on the power utilization for multi-threaded workloads.

II. Background

The cache sharing concept introduced in this paper is applied to a dynamically reconfigurable VLIW processor introduced by [4]. It is a proof-of-concept that can dynamically balance Instruction-Level Parallelism (ILP) and Thread-Level Parallelism (TLP) by dividing resources between cores as efficiently as possible. This is done by splitting the processor into multiple separate cores when there is a high level of TLP and merging them back into a large (high-issue width) VLIW core when there is only a single program with a high level of ILP. One of the special characteristics of the ρ-VEX compared to related dynamic processors (see Section VII) is that it is a VLIW architecture. Traditional VLIW architectures need binaries that were specifically compiled for their organization (number of datapaths, pipeline organization). The ρ-VEX project has solved this drawback by introducing generic VLIW binaries [5], that allow a single binary to be executed on different issue widths. In this manner, binary compatibility is achieved. In the following sections, the ρ-VEX dynamic processor and cache design will be described briefly.

A. Dynamic Core

This section provides a brief overview of the ρ-VEX processor. For a more in-depth discussion, we refer to [4] and [6]. The ρ-VEX VLIW core is a VHDL implementation of a dynamically reconfigurable VLIW processor. It consist of a number of 2-issue VLIW processors that can be merged if the available ILP is high enough. Merging can be performed in powers of 2 - combinations of 2, 4, and 8-issue cores are possible. A configuration switch has a latency of approximately 9 cycles, which is the time to decode the new setting and flushing the 4-stage pipelines. The total number of cores is design-time reconfigurable. The design has been prototyped on FPGA using up to 4 2-issue cores (that can be merged into a 8-issue VLIW). Larger configurations are supported by the code, but not feasible for prototyping on FPGA. All sub-cores have their own register file, control registers, and a complete set of functional units, so they can operate fully independent. In case the processor is running in single-core mode, it functions as an 8-issue VLIW processor. The principle works by multiplexing the program counter and other architectural control registers to the datapaths depending on the processor configuration. There core will be interfaced with the dynamic cache using 4 program counters and data access ports (address, data, write/read enable signals). Furthermore, there are 4 sets of memory mapped control registers and signals related to

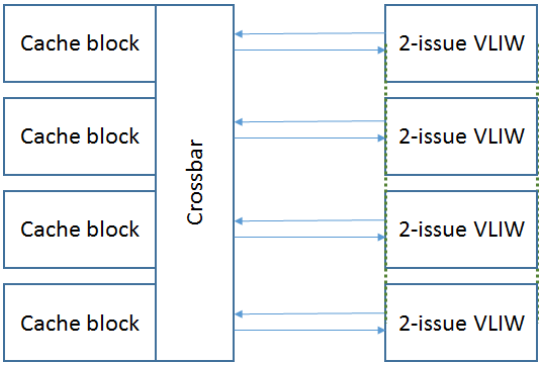

Fig. 1. Cache design that allows the ρ-VEX to run multiple programs as separate cores or one program as a combined core, while always using all available cache storage capacity.

interrupts etcetera that interface with the rest of the SoC. Depending on the configuration, these connections may or may not be active at the same time. This requires a dynamic cache that can support this.

B. Dynamic Cache

To allow the core to run using its different configurations, a cache has been developed that consists of multiple blocks. To increase efficiency of the blocks, they can be combined much like the datapaths of the processor to work together when the configuration allows (see Figure 1). In a more straight-forward design, each core would have a private block of cache directly connected to it. When combining multiple cores, this would mean that the other blocks of cache stay idle. The dynamic cache works by introducing a small series of multiplexers between the cache blocks and the processor cores. The network routes the request from a core to all the blocks that are connected to it in the current configuration. The data from the block that has it is routed to the correct memory unit, similar to how a set associative cache works. We exploit this functionality that is already in place to support our proposed cache request broadcasting scheme, as we will describe in Section IV.

III. Concept

In multi-threaded applications, there are 2 types of memory accesses; (thread-)local and global (also referred to as "private" and "shared", respectively). The first type is to data that is used only by the thread itself, such as variables on the stack. The other type is to data that is global to the application, and that all threads need to access or update. A programmer typically creates stack variables when declaring them inside a function declaration, and global variables by declaring them outside functions. In other words, the distinction between stack variables and other (possibly shared) data on a software level is trivial. The amount of data that is communicated between multiple threads is an inherent characteristic of the algorithm and implementation; some algorithms can run very independently and others need large amounts of communication between the working sets of different threads (see Table I for the communication characteristics of the benchmarks used in our evaluation).

Dynamically shared caches try to increase the hit rate when running applications with large amounts of communication, while minimizing latency for applications with small amounts of communication. Figure 2 shows a simplified overview of how a dynamically shared cache works; it is able to forward memory accesses to multiple cache blocks, depending on a reconfigurable interconnection network (a series of switches that can forward the request or not). The caches blocks need to be able to handle requests from multiple sources, as will be discussed in more detail in Section IV. This results in penalties in the form of added circuit complexity and access latency when considering the hardware design, and increased contention and energy utilization at the respective cache level during run-time. On the other hand, it also results in decreased numbers of cache misses, resulting in *de*creased contention and energy utilization *after* the respective cache level. By making the level of sharing dynamic (as has been done in previous work), this balance can be tuned in favor of the current workload. In existing dynamically shared caches, all memory accesses are forwarded because the hardware does not know whether they are global or local.

We propose to use the distinction between local and global memory accesses to send only the global memory requests to the shared caches and to keep local memory requests private. Because local variables can never hit in other cache blocks, broadcasting these requests can never result in lower miss rates (Figure 2, left side). However, the requests will still increase L2 or main memory contention and energy utilization. By broadcasting only the global memory accesses, miss rates can be reduced with similar ratio's compared to existing shared caches, while causing the penalty (i.e., increased contention and energy utilization) to be reduced.

We propose two methods to implement the concept. The first method preserves binary compatibility and does not require compiler support. Many architectures use a special register to keep track of the stack. All accesses to the stack (that are inherently local) will be performed by addressing memory relative to that register, which is how the hardware can detect whether a memory access is local. Additionally, many architectures (such as ARM and x86) use PUSH and POP instructions to access the stack. The second method entails adding an instruction to the ISA that allows the programmer or compiler to broadcast a memory requests only if it has a certain probability of hitting in a neighboring cache block. This will also allow thread-private malloc'ed memory to be distinguished and will improve the results. This paper only evaluates the first method.

The reduction in total accesses to all shared blocks of cache can be formulated as

$$Red_{L1acc} = \frac{(Acc_{local} + Acc_{global}) * N_{sharedcaches}}{Acc_{local} + (Acc_{global} * N_{sharedcaches})} \quad (1)$$

Where N, the number of cores and caches, is a design choice for the hardware platform, the number of global and

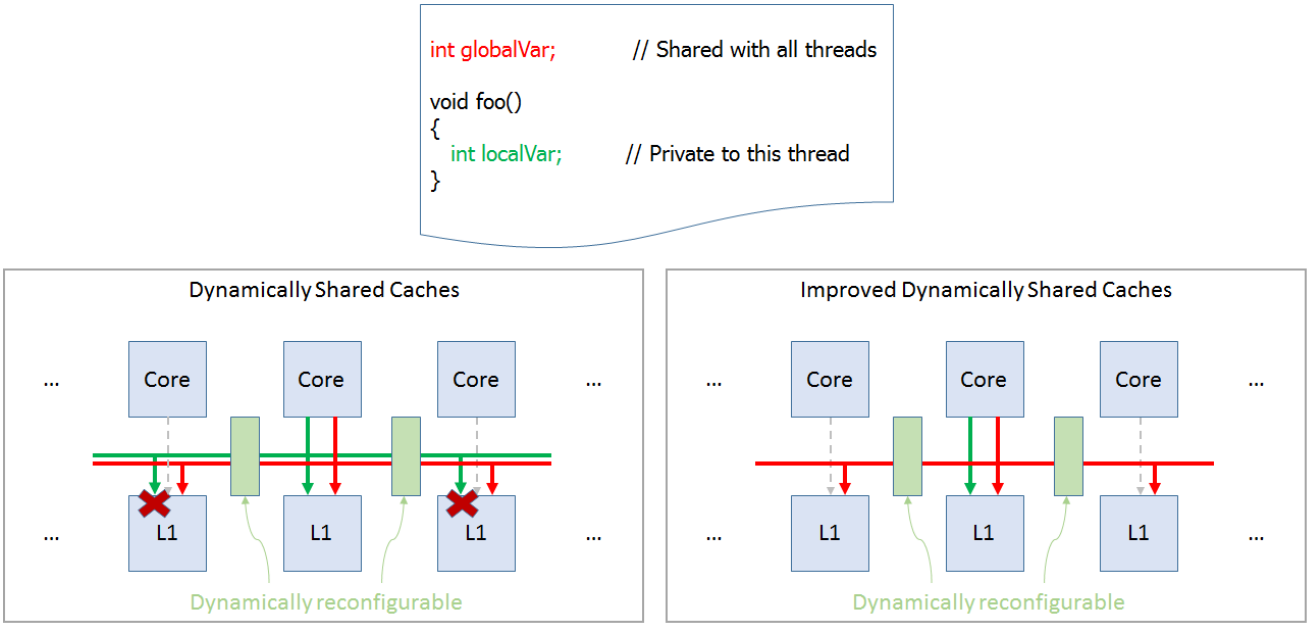

Fig. 2. Diagram explaining the concepts of dynamically shared caches on the left (previous work), and our improved version. As can be seen, a reference to a memory value that is local to a thread can never hit in another core's cache (depicted by the red cross). The access does consume energy and create contention on the shared caches. By broadcasting only those requests to memory that potentially resides in other core's caches, the same improvements in cache hit ratio's can be achieved.

local *Acc*esses are a program characteristic (see Table I), and *Red*uction is the reduction in total traffic to all the N cache blocks in the system that is caused by a thread. This reduction will lower the bandwidth requirements on the cache blocks and can lead to energy savings.

IV. Implementation

Shared access to memories can be achieved in two ways: (1) by increasing the number of access ports that can be used independently and (2) by arbitrating between requests. Increasing the number of ports is very expensive in terms of circuit area, which is one of the reasons why the shared higher-level caches are much slower than private first-level caches (e.g., L2 access latency is 21 cycles compared to 4 cycles for the L1 in an ARM A15 chip). Arbitrating between requests will mean that one of the requests will be stalled while the other is being handled.

As the ρ-VEX cache design already has multiplexers in place to connect the cores to the cache blocks (see Figure 1), adding the broadcast mechanism is considerably less complex than adding an access port for each core to all cache blocks. In case one of the cores performs a cache request broadcast, the request is forwarded to all blocks in the same way as when the cache is running in single-core mode, and the result is forwarded to the requesting core. The other cores' cache accesses are delayed using the same logic that handles cache misses. An alternative design could ignore the broadcast request if the cache is busy, which would increase the probability of the broadcast resulting in a miss but decrease the total hit penalty.

Our implementation only considers read accesses at this point, because the ρ-VEX caches are write-through. The processor is configured as a 4-core 2-issue VLIW. The 4 cache blocks participate in the shared setup, as a sharing degree of 4 has been found to be the most effective by [3].

V. Evaluation

To measure the efficacy of the improved dynamic cache sharing concept, a SoC simulator, written in C, is utilized that simulates a reconfigurable ρ-VEX core with dynamic L1 cache that is connected to a main memory through a simple round-robin bus model. We are using the characteristics of the ρ-VEX FPGA prototype for evaluations, where the core is running on 80 MHz, L1 caches have 1 cycle latency, main memory has 12 cycles penalty for a read miss and 8 cycles for a write miss. The L1 instruction cache has 16 cycles miss penalty. To estimate energy utilization, we used CACTI [7]. To attempt to accurately model the energy utilization of the different types of accesses, we have used a model of a 4KiBdirect mapped L1 cache with a line size of 4 bytes for the data cache blocks. The instruction cache blocks are 8KiBdirect-mapped L1 caches with a 32 byte line size and the main memory is a 512MB DDR3.

A number of benchmark programs from the target application domain is executed on the simulator using 3 different cache behaviors: the default (baseline), a fully shared mode where all data cache read accesses are broadcasted to all cache blocks, and (our approach) a global-only shared mode where only data cache accesses that will potentially access globally shared data are broadcasted to all cache blocks. The target application domain in this case is multi-threaded programs that have some level of communication between the threads.

Workloads without any communication do not benefit from cache sharing and therefore are not evaluated.

A. Benchmarks

Five parallel implementations of well-known algorithms with different memory usage behaviors (accesses to shared and private addresses) were designed. Table 1 depicts the main characteristics regarding communication of each benchmark, obtained by using the PIN Tool [8].

- *Gauss*
  The Gauss method is a technique for solving the n equations of the linear system of equations $Ax = b$, where A is the matrix of coefficient m x n; x is the vector of variables, and b the vector of terms [9].
- *Jacobi*
  The Jacobi method consists of an iterative algorithm to determine the solution of linear systems involving a large percentage of zero coefficients. Assuming a linear system $Ax = b$, where A is the matrix of coefficient m x n; x is the vector of variables, and b the vector of terms; the goal is to find an approximate result for x through the convergence of the vectors [9].
- *LU*
  LU-Decomposition uses the Doolittle Method (a widely used algorithm [9]) to perform the Lower-Upper Decomposition.
- *OddEven* - Odd/Even sort is a sorting algorithm based on bubble sort that compare pairs of elements. In a first step, the indexed pairs are analyzed (odd, even). If the value of the first element is greater than the second (even), they are exchanged. In a second step, the same thing is done, but now with the inverted pattern (even, odd). These two steps are alternately repeated for *N/2* iterations, where *N* is the number of elements in the vector.
- *Turing*
  Turing Ring describes a space system that predators and prey interact in one location. The system consists of the simulation of iteration and evolution of predators and prey through the use of differential equations, and the evolution is according to the neighboring cells [10].

The applications were implemented using the C language. Since the way the parallel algorithm is written may influence its behavior during execution, we have followed the guidelines indicated by [11] and [12]. Therefore, the applications were parallelized using the fork-join model, where the master thread is in charge of initializing the data, calculating the workload division and starting the other threads.

All benchmarks were ran using multiple input sizes, which largely determine the cache performance. We expect that, as the input sizes grow, the working set of the algorithms will become too large, and cache performance will start to degrade. When this happens, we expect the effectiveness of the shared cache will degrade as well. Note that this is not a shortcoming of the concept as cache performance is an important factor for every program and scalability in this regard is a characteristic of the algorithm and/or its implementation.

TABLE I
CHARACTERISTICS OF THE BENCHMARKS REGARDING MEMORY ACCESSES WHEN EXECUTING 4 THREADS

Benchmarks	Private		Shared		Total
	Write	*Read*	*Write*	*Read*	
Gauss	49.02%	22.49%	8.93%	19.56%	100%
Jacobi	54.08%	22.86%	4.08%	18.98%	100%
LU-Decomposition	62.68%	22.79%	0.37%	14.16%	100%
Odd-Even Sort	30.46%	24.18%	21.56%	23.80%	100%
Turing Ring	41.79%	24.89%	27.94%	5.38%	100%

B. Results

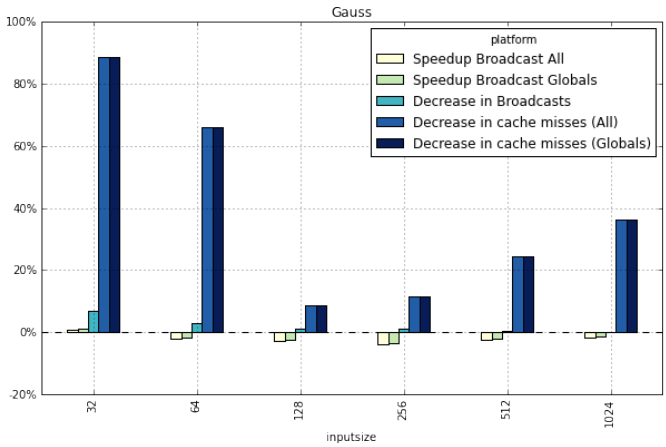

Fig. 3. Gauss benchmark results

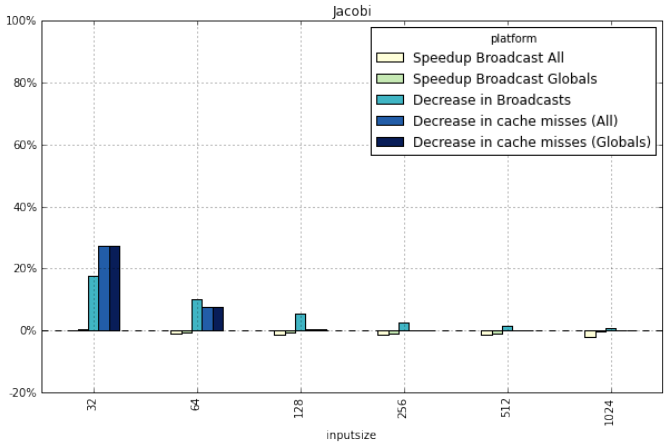

Fig. 4. Jacobi benchmark results

Figures 3 to 7 depict the performance and cache results of the benchmarks on the 3 platforms. Interestingly, there is very little performance difference in most cases, in contrast to what is measured by [3]. This might be caused by the relatively heavy broadcast hit penalty we assume (1 cycle compared to a single-cycle access latency for non-shared caches) and the penalty we incorporate for when a broadcasted request collides with other requests (instead of assuming a multi-banked setup).

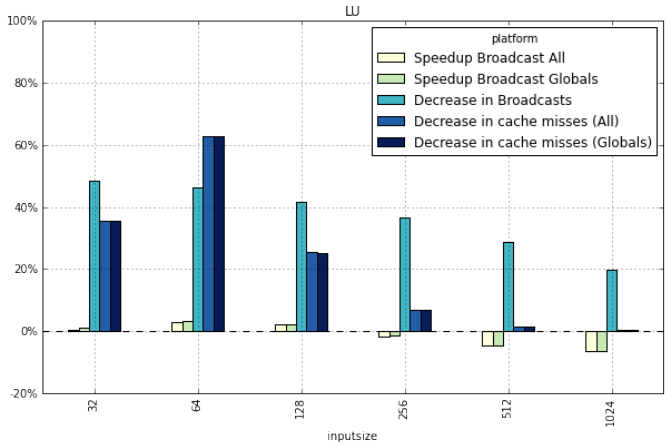

Fig. 5. LU benchmark results

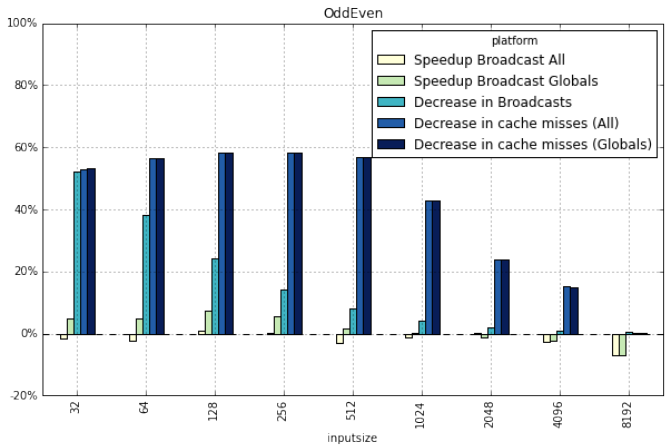

Fig. 6. Odd/Even benchmark results

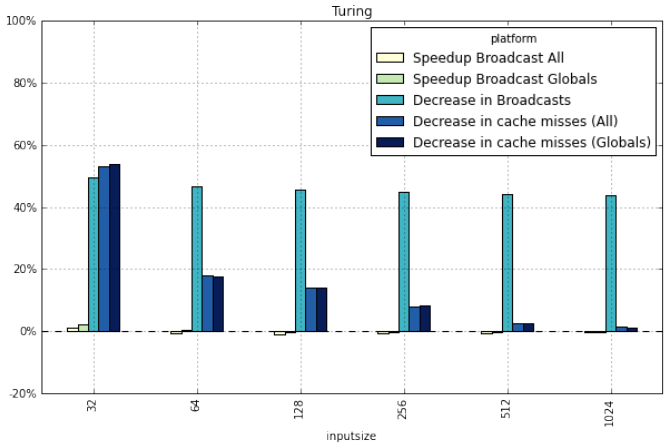

Fig. 7. Turing benchmark results

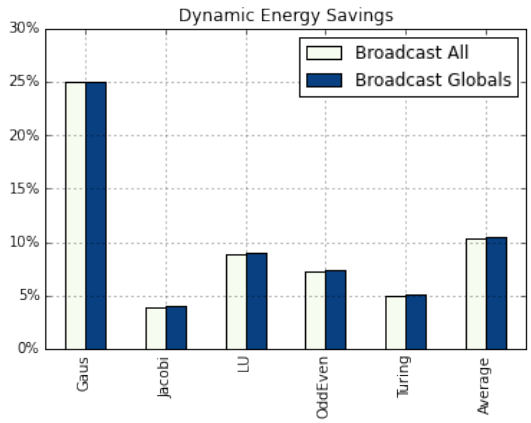

Fig. 8. Dynamic energy savings compared to ρ-VEX standard cache (non-shared). In this graph, the energy savings by not broadcasting local variables is insignificant next to the savings because of the miss rate decrease.

It can be seen that our cache scheme has a slight advantage and in the OddEven benchmark, it changes a *reduction* in performance into an *improvement* in performance. The results for reductions in data cache read misses show that, as expected, both shared cache behaviors perform very similar. Finally, the results show considerable decreases in broadcasted requests of up to 90% and on average 26%.

Most of the benchmarks show a large drop in hit rates as the input size increases, and the efficiency of the proposed cache shows a correlation with it. This confirms the expectation that the concept needs a certain hit rate to function properly (as the probability of a word residing in memory is decreasing, so is the probability to find it in other cache blocks). The Turing benchmark is showing some counter intuitive results because both the hit rates and reduction in cache broadcasts for the improved cache are fairly stable over different input sizes, but the effectiveness of cache sharing does in decreasing misses diminishes. Figure 8 Shows the average dynamic energy savings for the 5 benchmarks. These results are mostly influenced by the number of cache misses, as these consume considerably more energy than core cycles and cache hits. We have plotted only dynamic energy usage because the static energy consumption stays very constant between the different setup (as there is limited difference in performance).

Important to note is that the VEX architecture targets the media processing domain and is equipped with a large number of registers. This causes the number of local accesses to be relatively low compared to architectures such as x86 and ARM, that have limited amounts of registers (and will therefore need to use the stack more often).

VI. Conclusion

Sharing caches can decrease the number of cache misses and thereby increase performance and/or decrease energy utilization. Earlier shared cache systems broadcast all accesses to the caches, including local accesses. Our work introduces a distinction between local and global data that is supported by the processor architecture (by distinguishing accesses relative to the stack pointer register). Using this concept, we have shown a cache system that only broadcasts the global cache accesses (that potentially reside in the caches of the other

cores) while keeping most local accesses private. Our results show that the decrease in cache misses and the resulting energy saving is similar to broadcasting all cache accesses, while the number of broadcasts required to achieve this is reduced by 21% on average. Furthermore, results show that the performance is in many cases dependent on whether the working set fits in the caches. Exploiting the characteristics of the ρ-VEX platform, the cost of adding the dynamic cache sharing functionality is small because the processor already has much of the required logic in place in order to support its dynamic adaptability.

VII. Related Work

A. Dynamic Processors

The realization that dynamic workloads require dynamic computing platforms has inspired several academics to design dynamic systems. Dynamic processors (examples being [2] [13] [14]) try to target diverse code where, usually, the goal is to provide high performance for single-threaded programs with high ILP and high throughput for multi-threaded programs with high TLP (Thread-Level Parallelism). In other words, these processors can adapt to the characteristics of the running program. The ρ-VEX is no different in this regard. What is different, is that 1) it targets the embedded domain and 2) uses a VLIW style architecture that allows a natural mapping of tasks/threads to datapaths. The compiler has already created instruction bundles, that all utilize a certain number of datapaths. If there are unused datapaths, they can be assigned to run another thread or switched off to conserve energy. This way, there is no need for very large instruction windows and dependency checking circuitry (needed by the ρ-VEX's superscalar counterparts in the general-purpose domain) that becomes even more complex for dynamic processors. The ρ-VEX will be discussed in Section II-A.

B. Dynamic Caches

This work touches upon the areas of reconfigurable caches, cache partitioning, and cache sharing. In particular, the dynamic (run-time) variants are the most relevant and will be discussed. A large body of work has been dedicated to *dynamic caches*, partly because there are multiple ways in which a cache can be dynamic. Many caches have been designed that are reconfigurable in the level of associativity, size, and line size [15] [16]. *Cache partitioning* can be viewed from both the software and hardware point of view. From the software's perspective, it is possible to analyze where data used by programs (or even functions within programs) will be mapped into the cache. In this manner, contention can be identified at compile-time and by modifying the layout of the data in the binaries, cache behavior can be improved (see for example [17]). This is not related to this work, as we are proposing a hardware cache sharing mechanism. A hardware's perspective (of a dynamically partitioned cache) is presented in [18], where storage areas can be partitioned and used for different purposes (e.g., cache, local memory/scratchpad, prefetch buffer, and lookup tables/instruction reuse buffer). This work discusses caches that are partitioned in order to change the level of *sharing* between cores.

The level of cache sharing determines whether all processor cores have access to a private block of cache memory or if they are all attached to the same cache but have to contend for access. There are numerous design points in between (such as sharing a cache between pairs of cores), a popular design being a private L1 cache and shared L2 caches (or in the general-purpose domain: private L1 and L2 caches and a very large shared L3). In most systems, these are design-time characteristics.

Academics have designed systems where the level of sharing can be modified at run-time (i.e., dynamically). Some use a setup where the storage capacity is divided between different (unrelated) processes ([19], [20] and [21]). This is not related to this work as we are targeting multi-threaded applications that communicate data between cores. In [3], a design is proposed where 16 cores can access a pool of cache banks. The degree of sharing (private, shared with a group of n processors or fully shared with all processors) is dynamically reconfigurable. The trade-off to be made here is that higher degrees of sharing may reduce misses, but will increase hit latencies. in [22], a similar concept of detecting private memory regions is discussed, where the directory based coherency can be switched off. Our work focuses on a simpler setup with a dynamic processor that has write-through caches with only bus snooping for coherency. Furthermore, our scheme uses the stack register to distinguish private accesses instead of requiring Operating System support.

References

[1] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 111–122. [Online]. Available: http://dx.doi.org/10.1109/PACT.2004.15

[2] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, "MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, Dec 2012, pp. 305–316.

[3] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA Substrate for Flexible CMP Cache Sharing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 8, pp. 1028–1040, 2007.

[4] F. Anjam, M. Nadeem, and S. Wong, "Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[5] A. Brandon and S. Wong, "Support for Dynamic Issue Width in VLIW Processors Using Generic Binaries," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 827–832.

[6] A. Brandon, J. Hoozemans, J. van Straten, A. Lorenzon, A. Sartor, A. C. S. Beck, and S. Wong, "A Sparse VLIW Instruction Encoding Scheme Compatible with Generic Binaries," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015, pp. 1–7.

[7] CACTI 6.5, http://www.hpl.hp.com/research/cacti.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[9] W. H. Press, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge university press, 2007.

[10] J. Paudel and J. N. Amaral, "Using the Cowichan Problems to Investigate the Programmability of X10 Programming System," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. ACM, 2011, p. 4.

[11] I. Foster, "Designing and Building Parallel Programs," 1995.

[12] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[13] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 422–433.

[14] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse, "The MAJC Architecture: A Synthesis of Parallelism and Scalability," *IEEE Micro*, vol. 20, no. 6, pp. 12–25, 2000.

[15] C. Zhang, F. Vahid, and W. Najjar, "A Highly Configurable Cache Architecture for Embedded Systems," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 136–146.

[16] L. Chen, X. Zou, J. Lei, and Z. Liu, "Dynamically Reconfigurable Cache for Low-Power Embedded System," in *Natural Computation, 2007. ICNC 2007. Third International Conference on*, vol. 5. IEEE, 2007, pp. 180–184.

[17] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 340–351.

[18] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable Caches and their Application to Media Processing," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, June 2000, pp. 214–224.

[19] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26. [Online]. Available: http://dx.doi.org/10.1023/B:SUPE.0000014800.27383.8f

[20] H. Dybdahl and P. Stenström, "An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 2007, pp. 2–12.

[21] W. Wang, P. Mishra, and S. Ranka, "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-time Multi-core Systems," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 948–953. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024935

[22] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 93–104.