Automated Data Cache Placement for Embedded VLIW ASIPs

Paul Morgan[1], Richard Taylor, Japheth Hossell, George Bruce, Barry O'Rourke
CriticalBlue Ltd
17 Waterloo Place, Edinburgh, UK
+44 131 524 0080
{paulm, richardt, jexh, georgeb, barryo} @criticalblue.com

ABSTRACT
Memory bandwidth issues present a formidable bottleneck to accelerating embedded applications, particularly data bandwidth for multiple-issue VLIW processors. Providing an efficient ASIP data cache solution requires that the cache design be tailored to the target application. Multiple caches or caches with multiple ports allow simultaneous parallel access to data, alleviating the bandwidth problem if data is placed effectively. We present a solution that greatly simplifies the creation of targeted caches and automates the process of explicitly allocating individual memory access to caches and banks. The effectiveness of our solution is demonstrated with experimental results.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – *Cache memories;* C.1.1 [**Processor Architectures**]: Single Data Stream Architectures – *VLIW Architectures*

General Terms
Algorithms, Design, Performance, Theory

Keywords
Cache, cache optimization, embedded applications, ASIP.

1. INTRODUCTION
Embedded systems often employ application-specific instruction processors (ASIPs) that have been tailored to the domain in which they will be employed. In the interests of maximizing performance and minimizing energy consumption it is desirable to exploit instruction level parallelism inherent in the code. Employing a VLIW processor provides an ideal mechanism for extracting this parallelism. However, a significant number of instructions in many applications are loads or stores, in our experiments typically around 30% of all instructions, therefore data memory bandwidth issues are often a significant bottleneck to successfully exploiting instruction-level parallelism. Thus it is necessary to instantiate and effectively utilize data cache units that allow multiple concurrent accesses to maximize data bandwidth.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

Access patterns for the instruction cache tend to be much more structured and predictable than those for the data cache leaving more scope for performance improvement in successful data cache configuration and data allocation. The key to achieving an optimal solution is maximally exploiting both temporal and spatial locality in memory accesses, which are application dependent. Factors such as cache size, bank configuration and number of ports present a highly configurable architecture. Multiple ports allow simultaneous access to a single cache, different banks hold different data sets within the cache, and multiple caches can have different properties each suited to different data access patterns within the application. Effectively utilizing cache architectures, both in terms of selecting the hardware configuration and optimizing data allocation to exploit maximum benefit from the chosen configuration, is a challenging and time consuming task.

We present an automated solution by way of a software tool for guiding the creation of a suitable hardware configuration and allocating data to optimally utilize the selected configuration. This is achieved by automatically generating and analyzing the memory trace of an application, taking advantage of the memory access information available at design and compile time to produce a more efficient allocation than would be possible by performing dynamic allocation using run-time logic. We provide a library of cache blocks to allow a wide range of architectures to be created tailored to the target application. Our tool guides the user towards an ideal hardware solution by performing allocation and analysis on a selection of candidate architectures, producing comparative results for each candidate architecture.

This document is presented as follows. First we examine a selection of related work in Section 2. In Section 3 we list the hardware blocks created to build our caches, and detail the software allocation algorithm used to optimize data allocation to the cache. In Section 4 we undertake experiments to show the cache performance benefits of our solution. Finally we present our conclusions and suggest future work that could be undertaken to further our research in Section 5.

2. RELATED WORKS
A great deal of research on the topics of cache configuration and mapping has been undertaken in the past with many of the methods being proposed targeted at application-specific architectures. Givargis [3] recognized that better cache performance can be obtained by considering the target application during the design phase of an ASIP. Similarly, Panda et al. [8]

[1] Paul Morgan is also based at the Institute for System Level Integration, Livingston, UK.

demonstrated a method of optimizing memory hierarchy, including data cache, for application-specific designs.

Single cache optimizations such as varying line size, set associativity or replacement algorithm have been covered for several goals, such as energy [14][15] or hit rate [7]. For application-specific architectures it is often beneficial to have one or more additional caches with a different configuration to the first, depending on the nature of the application being executed. A well-researched technique is that of the scratchpad memory [4][8], a small area of storage in which elements can be placed without disrupting the main cache. Gordon-Ross et al. [4] extend the analysis to a two-level cache hierarchy, proposing a simultaneous exploration technique for both cache levels that trades off power requirements and performance.

Sudarsanam and Malik [12] addressed the issue of memory bank assignment to optimize for simultaneous access in ASIPs with a tool called SPAM. This work tackles a similar problem to what we face but is targeted at single cache ASIPs with two identical banks whereas our tool targets highly configurable architectures that can have multiple caches of different types each with different sized banks.

Grun et al. have produced excellent work on memory architecture exploration in [5] culminating in a tool called APEX. This work considers the entire memory of an embedded system, rather than focusing on the data cache and does not provide for the parallel data access requirements of multiple-issue VLIW processors.

What we propose is to provide a library of customizable cache blocks that can be tailored at design time to a suitable configuration for the target application. Allocation of code to caches and banks is automated by a tool we have designed using a software algorithm that attempts to find an optimized solution for the selected hardware caching architecture taking into consideration the parallel access requirements of a multiple-issue VLIW processor. This approach allows multiple possible candidates for the hardware configuration to be quickly examined for suitability, overcoming the problem of attempting to simultaneously optimize both hardware configuration and software mapping, a problem which could not be solved in reasonable time with our level of configurability.

To the best of our knowledge no previous work has explored an automated software mapping for highly configurable hardware cache architectures as proposed here.

3. CACHE ALLOCATION

To facilitate the creation of an application-specific data cache, we provide a library of highly configurable cache blocks to allow our cache to be optimized for a wide range of applications. There are currently four cache styles in our library; three window caches and one direct-mapped static cache. A cache unit may contain a number of independent banks, each of which may hold a different data set. Using multiple banks allows different data areas to be held, addressed independently and accessed simultaneously depending on available ports.

Window caches hold a contiguous region of memory in each bank, and automatically attempt to keep the correct addresses in the cache by pre-fetching data from main memory in the background when accesses are ascending or descending and are nearing the edge of the cached region. The three window caches are distinguished by their port configuration, one with a single read/write port, the second with an additional read only port and the third with two read/write ports. Additional ports increase the complexity of the cache so the trade-off between area and performance must be considered. Window caches require no tag overhead due to the cached memory region being contiguous, significantly reducing the area footprint, and the pre-fetch mechanism greatly improves performance on favorable access patterns.

Static caches provide a more conventional direct-mapped cache, with the addition of software placement of data into banks. Such caches are simpler than window caches, with no pre-fetch mechanism, and anything between 8 and 64 lines each of 64 words to provide a more suitable cache for accesses of a sparsely spread pattern, with sizes of 2k, 4k, 8k or 16k bytes. All static caches are single-ported, utilizing around 40-45% less area than a dual-ported window cache depending on configuration options.

Each of the four cache units has further parameterized configuration options to ensure maximum flexibility to adapt to any application. The size of each window cache is configurable in powers of two from 512 to 64k words, with 1, 2, 3 or 4 banks. In addition to design-time choices, the number of banks and bank size ratios can be dynamically configured by the host at runtime.

All our cache blocks are directly mapped taking advantage of the lower latency, smaller area requirements and reduced power consumption offered compared with set- or fully-associative caches, as tag comparisons are not required with direct-mapped caches. We rely upon an effective software allocation and the pre-fetching abilities of our window caches to minimize cache misses that would otherwise be inherent in a direct-mapped cache.

The aforementioned cache blocks provide an enormous range of configuration options. There are 32 possible valid combinations of each window cache, and 4 static cache options, meaning a dual-cache design offers 1296 configurations. It would be an intractable task to attempt to fully automate the selection of an optimal cache configuration that meets all the required criteria for any custom application. Therefore the user selects a number of candidates for the cache configuration from the provided hardware blocks based upon area and performance requirements, and the type of application being accelerated.

The target application is run with a representative data set, and a memory access trace is automatically generated. The trace is then analyzed to determine ranges of memory that show independence in either the spatial or temporal ranges. Instructions are partitioned into groups whose access patterns interfere both spatially and temporally. Each group is allocated to a cache bank according to the algorithm described below. Hardware cache coherency logic ensures that the memory hierarchy will always be valid for any access pattern regardless of the memory configuration, relieving the allocation algorithm of this concern and providing resilience to any future changes in the executed code. Additional logic ensures that any location in the memory hierarchy can be accessed from any port, although an interference stall penalty is incurred if the data is cached in a location other than the identified bank allowing time for the hardware to transparently fetch the data from the correct bank. The software analysis optimizes the allocation of memory regions to the

available cache configurations providing post-allocation performance statistics on each candidate to guide the selection process towards an optimal solution.

3.1 Allocation Algorithm

The aim of the allocation algorithm is to assign grouped memory access instructions to appropriately sized banks to minimize cache misses, and minimize interference between groups by assigning concurrently active ranges into different banks where possible. In cases where both a window cache and static cache are available, the algorithm attempts to select the most appropriate cache type for each group. A flow diagram overview is illustrated in figure 1 opposite. The tool examines the original program and lists all load/store instructions, then records from the memory access trace the range of addresses accessed by each instruction.

An interference graph is built with nodes representing load and store instructions, with instructions accessing overlapping address ranges with respect to cache line boundaries being merged into a single node. An interference edge is added between nodes that access data in the same activation range (a temporal run-time value calculated by the algorithm dependent upon the varying access density at the trace point of current analysis), identifying those nodes as being simultaneously "live", which is analogous to a register allocation interference graph [2].

Critical analysis is then performed to identify memory accesses that may require to be issued in parallel by the VLIW processor. These accesses are identified by performing a scheduling step on a fully optimized version of the most executed portions of code, forming critical access groups (CAGs) from accesses issued on the same cycle. This information is added to the interference graph such that if instructions in a CAG are located on the same node then that node's criticality is set to a value representing the number of simultaneous accesses that must be issued from the node. That node can then be allocated if possible to a bank with the required parallel access capability. If instructions in the CAG span multiple nodes then the criticality value is applied to the edge linking those nodes, indicating that those nodes should be allocated to banks with sufficient ports to satisfy the criticality constraints of both nodes simultaneously.

Nodes are sorted into priority order depending upon their memory access frequency to be assigned to available cache banks, starting with the most important node. Each bank's attributes, such as its type (window or static), the bank size, and the number of ports the bank is accessible through, are known to the allocation algorithm and are used to influence the selection of a bank for each node.

Choosing whether a node should use a window cache is a crucial step in the algorithm, as significant performance benefits are possible for access patterns amenable to window caching but performance can be degraded for unsuitable access patterns. Analyzing the entire memory trace and recording the frequency of all sequential accesses would be extremely slow and memory hungry, therefore selection of the cache type for each node is based upon the access proportion of that node. This is calculated as the number of accesses represented by that node divided by the address range accessed by the node. Nodes with access proportion above a threshold based upon available window and static banks are likely to perform sequential accesses so are earmarked for window caching. Nodes below the threshold will be allocated to static banks. We have found this approach to work well for most applications although a more robust and comprehensive algorithm for cache type selection is under development.

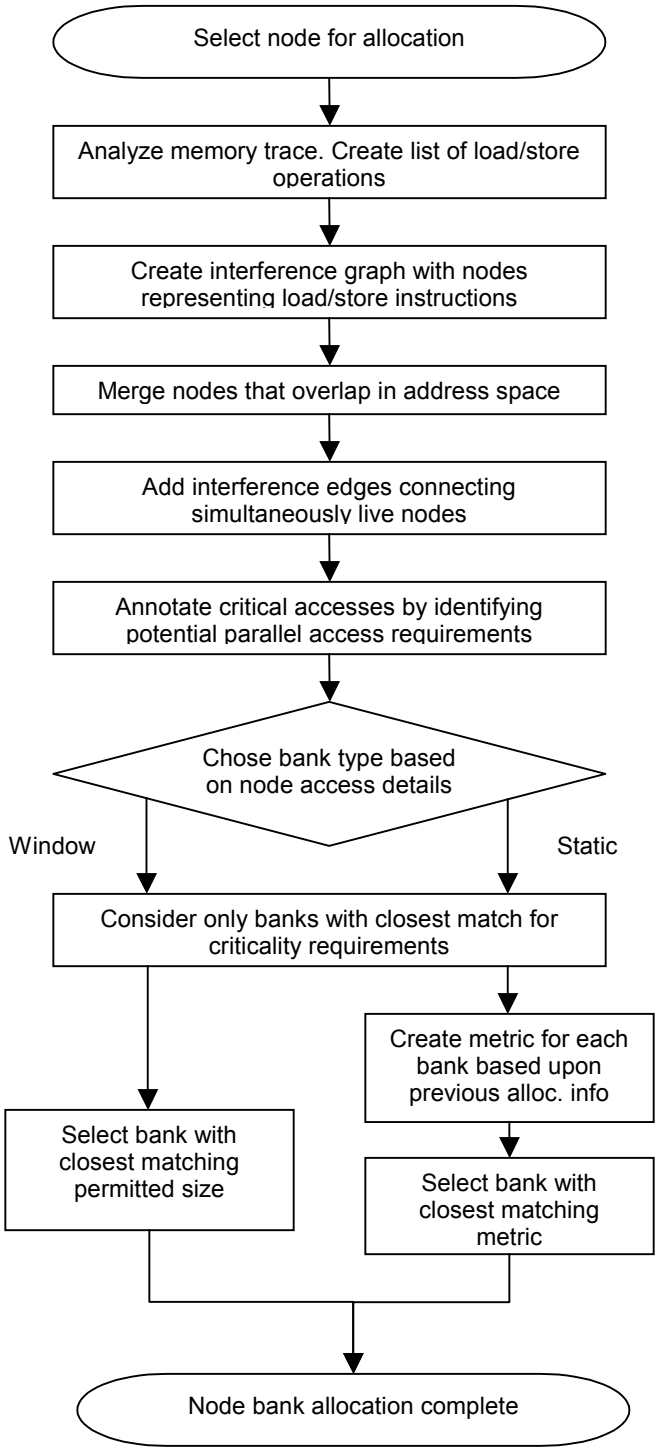

Figure 1. Allocation algorithm flow chart

Once the type of cache has been selected, the next step is choosing the bank that will be assigned to each node. The

criticality value determined previously, including analysis of neighboring nodes connected by critical edges, indicates the optimal number of ports for a node. The banks with the closest number of ports are selected for further consideration and all others are disregarded for that node. The remaining steps are dependent upon whether the node under consideration is targeted towards a window or a static cache. For window caches a preferred size is calculated based upon the total available size of window cache banks multiplied by the proportion of the overall access count generated by that node. The bank with size most closely matching the preferred size is selected.

Allocating static cache banks to nodes is more complex, requiring the generation of a metric for each bank to aid selection. All non-critical edges connected to nodes designated for static cache allocation are removed. This allows banks to be assigned to nodes that were connected by non-critical edges and is permitted because the metric generated for each bank contains information about where that bank has been previously allocated. Whenever a static bank is allocated a record of the lines used by the allocation is stored along with the tag(s) used for each line. Thus when the metric for a subsequent allocation is generated it consists of: the proportion of lines that negatively interfere with a previous allocation (two address lines map to the same cache lines with different tags); the proportion of lines that positively interfere with a previous allocation (similar to negative interference but with the same tags); and the proportion of the address range accessed by the node that does not completely fit into the cache. The bank with the best metric is selected for allocation. If two banks have identical metrics then the smaller bank is selected. If they are the same size a deterministic choice is made.

As nodes are assigned to banks in priority order less important nodes may be assigned to banks that do not necessarily fit their access pattern. The assumption is that a memory configuration can be found that allocates the most important accesses to suitable banks and any remaining accesses will have less influence on the overall performance of the memory. If there are no suitable banks available for a particular node, that node is assigned to the *default* bank which is designated the first time it is required. The default bank is chosen as the static cache bank with the least allocated accesses; if no static cache is available, the least accessed window cache bank is chosen instead. Once selected the default bank is then fixed for the rest of the allocation.

4. EXPERIMENTAL RESULTS

To evaluate our architecture and coupled allocation algorithm, we verify the performance of the system running real-world applications using instruction-set simulators (ISS). For our architecture we use a custom simulator that is part of our tool, and results are shown from an ARM920T using the ARMulator simulator. The ARM was chosen as it has 16Kb data cache arranged into a 64-way set associative configuration and mapped using a content addressable memory (CAM) [11] giving it a high level of adaptability for different applications. The ARM results are provided simply as a reference rather than a direct comparison, as our architectures are targeted towards specific applications in each case whereas the ARM is general-purpose. In addition, as we are targeting a multiple-issue VLIW processor that completes each experiment in fewer cycles, our target system places much higher demands on the data cache than the ARM processor.

By configuring ARMulator to produce verbose statistics during simulation, we can monitor cache activity such as hits, misses and fetches, for both instruction and data caches. We use our tool with a selection of potential cache configuration candidates which the tool cycles through performing allocation and producing results relating to the cycle count and cache hits and misses. Our tool is compatible with the ARM instruction set and can therefore utilize the same compiled code as that used on ARMulator.

To keep the design exploration simple and within the bounds of a realistic cache area for the selected applications, we limit the choices to one window cache or one static cache, or one of each, with a maximum size of 16Kb in total. Window caches are considered with varying bank numbers of 1, 2, 3 or 4, and have one read/write port and one read port. Static caches have a single read/write port. Even with this relatively small selection of that possible from the available hardware blocks, there are still a significant number of combinations to explore. The use of our tool greatly speeds this process, helping guide the user towards an optimal solution. Run-times for these examples are in the range of 2-10 minutes on a 2.8 GHz Pentium 4 PC with 1 Gb RAM.

To ensure that the results reflect the true cost of the miss penalties for each architecture, we have included an estimated number of stall cycles which indicates the number of bus cycles that the AMBA AHB bus consumes fetching or writing back cache lines. These estimates are based on factors such as initial transfer latency, burst transfer rate, cache line size, and bus contention. Our caches have been designed such that they do not increase the latency of accesses, maintaining overall system performance. Details of the AMBA AHB specification can be found in [1].

We ran several applications considered to be relevant to real-world embedded systems, which are also amenable to speedup on a VLIW ASIP and are therefore applicable to our target system. These are applications that we have previously targeted to some of our ASIP designs as part of other projects but in future we plan to extend our tests to relevant applications from the MediaBench suite. To ensure that results reflect only the monitored function, the caches are flushed and cleaned before entering the function so that the cache will have a cold start. This is achieved by inserting dedicated cache control assembly instructions immediately prior to entering the function. All caches were configured to use a write-back policy meaning that only a miss on a read or write requires a cache line to be synchronized with main memory causing a stall. One exception is an interference miss where the desired data is cached, but a read/write attempt is made on a bank or port other than where the data resides. The logic will automatically reference the correct location, but a shorter stall may be necessary in this case and this is taken into account in our "stall cycles" figures in the results.

We present the results of our experiments for each application. As expected some of our potential candidates did not produce competitive results, so due to space restrictions and the large number of candidates these were pruned and will not be considered further. Results for the three best candidates in terms of area, performance and energy, are shown for each application.

The selected candidates were synthesized for a TSMC 0.18μm process using Artisan memories to obtain the area requirements of each architecture including logic area. We then performed worst-case dynamic power analysis with high switching activity rates at

200MHz using Synopsys Power Compiler for logic cells and CACTI [10] for SRAM cells. Real-world power is likely to be lower as these figures are intended only for rough comparison between our architectures. The power figure for multi-ported architectures is shown per-port, as this provides a more realistic representation of the energy contribution over the entire application. This is because the instantaneous power of a dual-port cache performing two simultaneous accesses will be higher than that of a single-port cache, but the single-port cache will require two accesses on separate cycles to achieve the same result. More accurate integrated energy modeling within the tool based upon cache activity is a planned future development.

Worst-case dynamic power figures for a cache equivalent to that in the ARM were estimated using a combination of CACTI and the information in [16] regarding CAM-tag lookup caches. Area information for the ARM cache is not publicly available so we estimate the cache area based upon its configuration and available data on the arm architecture. The result appears to be high, but it agrees with the value calculated by extrapolating the difference in areas quoted by ARM for the ARM9 with different cache sizes. For comparison, the area of a 16K 4-way set associative cache with one bank and a 32 byte line size is 2.28mm^2.

For reference the logic overhead of our first cache architecture (Custom1 in section 4.1 below) is under 8%, relatively low even allowing for our pre-fetch logic due to lack of tag lookup overhead. Actual logic overhead will vary depending upon cache configuration and memory technology used.

4.1 Color Interpolation

The first application is a color interpolation function with approximately 300 lines of C. It performs integer colorization of Bayer-encoded images commonly produced by digital image sensors. It inputs an 8-bit intensity encoded bitmap image and outputs the full 24-bit image using interpolation. A detailed overview is available at [6]. This function relies heavily upon array manipulations therefore placing significant demands on the memory subsystem that must be satisfied to achieve good performance. The input image is CIF resolution (352x288) with file size 100Kb. The architectures selected for the color interpolation application are as follows:

- Custom1 – 4k Static cache; 8k Window cache, 1 bank
- Custom2 – 8k Static cache; 8k Window cache, 1 bank
- Custom3 – 16k Window cache, 1 8k bank + 2 4k banks

Results for this application are shown in Table 1.

Table 1. Results for color interpolation (access count 2822608)

Cache	Misses	Hit Rate	Stall Cycles	Area	Power
ARM	319830	88.67%	4797450	3.69mm^2	206mW
Custom1	29527	98.95%	236216	1.46mm^2	104mW
Custom2	14898	99.47%	119184	1.74mm^2	107mW
Custom3	123	**99.99%**	**3321**	1.95mm^2	129mW

Using our tool, we find the optimal configuration for this application is a single 16K window cache with one 8k bank and two 4k banks, resulting in a hit rate greater than 99.99%. This is largely due to the effectiveness of the pre-fetching mechanism fitting well with the bank configuration and access pattern. The allocation algorithm ensures that interferences between simultaneous accesses to different memory locations are minimized by allocating those locations to separate banks.

4.2 Run-Length Encoding

The second application is a run-length encoding function, a basic lossless compression algorithm that is simple to implement (approx. 200 lines of C) and has low computational and storage requirements. For this experiment, we compress an arbitrary data stream stored in a text file with a size of 50Kb. The architectures selected for the RLE application are as follows:

- Custom1 – 2k Static; 4k Window cache, 1 2k + 2 1k banks
- Custom2 – 4k Static cache; 4k Window cache, 2 2k banks
- Custom3 – 4k Static cache; 8k Window cache, 2 4k banks

Results for this application are shown in Table 2.

Table 2. Results for RLE function (access count 930914)

Cache	Misses	Hit Rate	Stall Cycles	Area	Power
ARM	109538	88.23%	1643070	3.69mm^2	206mW
Custom1	26	99.99%	702	0.93mm^2	90mW
Custom2	20	99.99%	540	1.14mm^2	97mW
Custom3	16	**99.99%**	**432**	1.47mm^2	104mW

Our architecture with a combination of one window cache and one static cache performs very well with only 6K total cache size. Further small improvements are possible with optimal performance realized at 12K total cache size. The access patterns of this application suit both a static and multi-bank window cache being implemented. Our tool performs allocation to the available caches and banks, and allows the user to decide the area/performance tradeoff between the possible solutions.

4.3 FIR Filter

Finally, we implement an integer FIR filter with 6 taps and supply a 20Kb input data stream. Signal processing places a high demand on the memory subsystem, therefore good cache performance is reflected in good overall performance for these applications. Architectures chosen for the FIR Filter application are as follows:

- Custom1 – 2k Window cache, 1 bank
- Custom2 – 2k Static cache; 2k Window cache, 1 bank
- Custom3 – 2k Window cache, 2 1k banks

Results for this application are shown in Table 3.

Table 3. Results for FIR Filter (access count 458718)

Cache	Misses	Hit Rate	Stall Cycles	Area	Power
ARM	67580	85.27%	1013700	3.69mm^2	206mW
Custom1	65536	85.71%	1769472	0.39mm^2	49mW
Custom2	263	99.94%	2104	0.64mm^2	83mW
Custom3	10	**99.99%**	**270**	0.40mm^2	49mW

The FIR filter test clearly shows the benefit of utilizing the flexibility of our architecture and the effectiveness of our

allocation mechanism. The 2K single-bank window cache results in a considerable number of miss cycles, but adding a 2K static cache shows a vast improvement. Going back to a single 2K cache, but with two banks produces the optimal result while maintaining a low area requirement.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented a software tool for guiding the creation of a cache configuration for application-specific VLIW architectures and automating data placement into that cache. Our experimental results show that this approach provides significant improvements over what would be possible using a conventional cache with placement performed at run-time, while at the same time keeping area and energy requirements low. Using window caches allows tag overhead to be eliminated and coherency issues are greatly reduced, but maintaining performance requires careful selection of the architecture and effective placement of data. The problem of effectively utilizing tailored cache architectures is solved by our automated solution that analyzes the code and performs allocation with the aim of optimizing cache efficiency.

Our allocation algorithm is being continually evolved. Particular effort is being focused at identifying more accurately the suitability of allocating ranges to window cache banks. We have not yet integrated energy optimization into our algorithm; currently our approach aims to lower system energy by reducing cache misses thus minimizing costly bus accesses [13]. A more detailed energy analysis and optimization is a prime interest in our ongoing research since the cache subsystem can account for up to 50% of the energy consumption in typical embedded processors such as the ARM920T [9]. As part of the continuing development of our tool, we are currently integrating data cache energy analysis as part of the automated flow, and plan to provide optimizations that may be traded off against performance and/or area criteria at the user's prerogative. Additionally, although we have focused on the data cache for performance optimization, the wide instruction cache in a VLIW processor provides great scope for energy savings; therefore we are in the process of exploring energy optimizations for the instruction cache with a view to integrating this functionality into the tool.

6. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers who provided constructive comments. Paul Morgan's contribution to this work is partially funded by EPSRC.

7. REFERENCES

[1] ARM Limited. AMBA Specification Rev. 2.0, 1999, pp 3-1 – 3-58.

[2] Chaitin, G. Register Allocation and Spilling via Graph Coloring. Proceedings of the 1982 Symposium on Compiler Construction, June 1982, pp. 98-105.

[3] Givargis, T. Improved Indexing for Cache Miss Reduction in Embedded Systems. Proceedings of Design Automation Conference, 2003, pp. 875 – 880.

[4] Gordon-Ross, A., Vahid, F., Dutt, N. Automatic Tuning of Two-Level Caches to Embedded Applications. Proceedings of Design, Automation and Test in Europe Conference, Volume 1, February 2004, pp. 208 – 213.

[5] Grun, P., Dutt, N., Nicolau, A. Access Pattern-Based Memory and Connectivity Architecture Exploration. ACM Transactions on Embedded Computing Systems, Vol. 2, No. 1, February 2003, pp 33–73.

[6] Kimmel, R. Demosaicing: Image Reconstruction from Color CCD Samples, IEEE Transactions on Image Processing, Volume 8, Issue 9, September 1999, pp 1221 – 1228.

[7] Megiddo, N., Modha, D.S. Outperforming LRU with an Adaptive Replacement Cache Algorithm. Computer, Volume 37, Issue 4, April 2004, pp. 58 – 65.

[8] Panda, P.R., Dutt, N.D., Nicolau, A., Catthoor, F., Vandecappelle, A., Brockmeyer, E., Kulkarni, C., De Greef, E. Data Memory Organization and Optimizations in Application-Specific Systems, IEEE Design & Test of Computers, Volume 18, Issue 3, May-June 2001 , pp. 56 – 68.

[9] Segars, S. Low Power Design Techniques for Microprocessors, International Solid State Conference, February 2001, pp 34 – 35.

[10] Shivakumar, P., Jouppi, N. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Compaq Western Research Laboratory report, August 2001.

[11] Sloss, A., Symes, D., Wright, C. ARM System Developer's Guide - Designing and Optimizing System Software, Morgan Kaufmann publishers, 2004, pp 403 – 457.

[12] Sudarsanam, A., Malik, S. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. ACM Transactions on Design Automation of Electronic Systems, April 2000, pp 242–264.

[13] Verma, M., Wehmeyer, L., Marwedel P. Efficient Scratchpad Allocation Algorithms for Energy Constrained Embedded Systems. Workshop on Power-Aware Computer Systems, December 2003.

[14] Zhang, C., Vahid, F., Najjar, W. A Highly Configurable Cache Architecture for Embedded Systems. Proceedings of the 30th Annual International Symposium on Computer Architecture, June 2003, pp 136 – 146.

[15] Zhang, C., Vahid, F., Najjar, W. Energy Benefits of a Configurable Line Size Cache for Embedded Systems. Proceedings of the IEEE Computer Society Annual Symposium on VLSI, February 2003, pp. 87 – 91.

[16] Zhang, M., Asanovic, K. Highly-Associative Caches for Low-Power Processors. Kool Chips Workshop, 33rd International Symposium on Microarchitecture, December 2000.