High-performance and Hardware-aware Computing

Proceedings of the Second International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'11)

San Antonio, Texas, USA, February 2011
(In Conjunction with HPCA-17)

Rainer Buchty
Jan-Philipp Weiß
(eds.)

KIT Scientific Publishing

Impressum

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe
www.ksp.kit.edu

KIT – Universität des Landes Baden-Württemberg und nationales
Forschungszentrum in der Helmholtz-Gemeinschaft

KIT Scientific Publishing 2011
Print on Demand

ISBN 978-3-86644-626-7

Optimized Replacement in the Configuration Layers of the Grid ALU Processor

Ralf Jahr, Basher Shehan, Theo Ungerer
University of Augsburg
Institute of Computer Science
86135 Augsburg, Germany
Email: {jahr, shehan, ungerer}@informatik.uni-augsburg.de

Sascha Uhrig
Technical University Dortmund
Robotics Research Institute
44221 Dortmund, Germany
Email: sascha.uhrig@tu-dortmund.de

***Abstract*—The Grid ALU Processor comprises a reconfigurable two-dimensional array of ALUs. A conventional sequential instruction stream is mapped dynamically to this array by a special configuration unit within the front-end of the processor pipeline. One of the features of the Grid ALU Processor are its configuration layers, which work like a trace cache to store instruction sequences that have been already mapped to the ALU array recently.**

Originally, the least recently used (LRU) strategy has been implemented to evict older configurations from the layers. As we show in this paper the working set is frequently larger than the available number of configuration layers in the processor resulting in thrashing. Hence, there is quite a large gap between the hit rate achieved by LRU and the hit rate achievable with an optimal algorithm. We propose an approach called qdLRU to enhance the performance of the configuration layers. Using qdLRU closes the gap between LRU and an optimal eviction strategy by 66% on average and achieves a maximum performance improvement of 390% and 5.06% on average with respect to the executed instructions per clock cycle (IPC).

***Index Terms*—Trace Cache, Replacement Strategy, Post-link Optimization, Feedback-directed Optimization, Coarse-Grained Reconfigurable Architecture**

I. Introduction

Within this paper, we present an optimization for the Grid ALU Processor (GAP), which has been introduced by Uhrig et al. [1]. It brings together a superscalar-like processor front-end and a coarse-grained reconfigurable architecture, i.e. a reconfigurable array of functional units (FUs). The front-end consisting of instruction fetch and decode unit is extended with a new configuration unit. This unit maps the instructions from the instruction stream dynamically and at run-time onto the array of FUs. Mapping of instructions and execution of instructions in the array run in parallel until there is a reason to flush the array and restart the mapping process. The mapping which has been built until this moment is called a configuration.

These configurations can be buffered in so-called configuration layers, which are formed by some memory cells very close to all the FUs. The configuration layers are very similar to trace caches. If a part of a program, i.e. a configuration, is already stored in the configuration layers it can be executed faster because it does not have to go through the front-end first, so instruction cache misses cannot occur. The timing inside the array is optimized, too. Because of this, it is a worthwhile goal to increase the usage of the configuration layers. Analyzing the execution of benchmarks we came to the conclusion that for some of them our default replacement strategy LRU works unexpectedly bad, even worse than replacing a random configuration layer (we call this strategy RANDOM). So LRU is in some cases not clever at all and humbles the execution speed.

The main contributions of this paper are (1) the analysis and comparison of the behavior of well-known replacement algorithms when applied to the replacement in the configuration layers and (2) the introduction and analysis of qdLRU. QdLRU improves the hit rate of LRU by adding flags to the program code based on a feedback-directed approximation of the working sets.

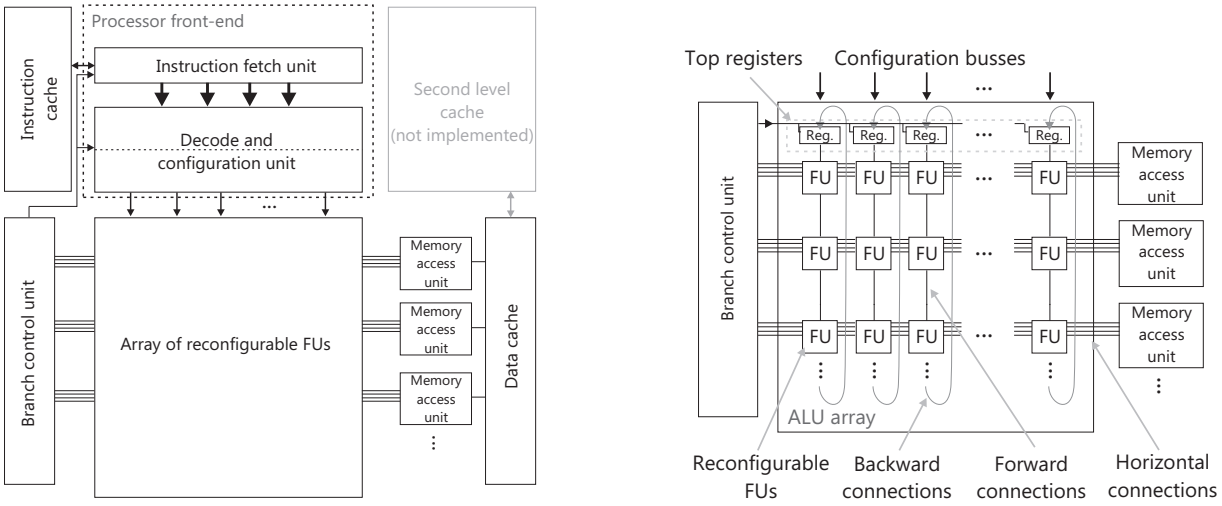
After giving a short introduction of the target platform in Section II, we discuss some basics about replacement strategies in Section III. The extended version of LRU called qdLRU is introduced in Section IV and evaluated in Section V. Related work is presented in Section VI and Section VII concludes the paper.

II. Target Platform: The Grid ALU Processor

The Grid ALU Processor (GAP) has been developed to speed up the execution of conventional single-threaded instruction streams. To achieve this goal, it combines the advantages of superscalar processor architectures, those of coarse-grained reconfigurable systems, and asynchronous execution.

A superscalar-like processor front-end consisting of fetch- and decode units is used together with a novel configuration unit (see Figure 1(a)) to load instructions and map them dynamically onto an array of functional units (FUs) accompanied by a branch control unit and several load/store units to handle memory accesses (see Figure 1(b)).

The array of FUs is organized in columns and rows. Each column is dynamically and per configuration assigned to one architectural registers. Instructions are assigned to the columns whose register match the instructions' output registers. The rows of the array are used to model dependencies between instructions. If an instruction B is dependent of an instruction A, it will be mapped to a row below the row of A. This way it is possible for the in-order configuration unit to also "issue"



(a) Block diagram of the GAP core

(b) General organization of the ALU array

Fig. 1. Architecture of the Grid ALU Processor

dependent instructions without the need of complex out-of-order logic. A bimodal branch predictor is used to effectively map control dependencies onto the array.

Execution starts in the first row of the array. The dataflow is performed asynchronously inside the array of FUs and it is synchronized with the clock of the branch control unit and the L/S units by so-called timing tokens [1].

Whenever either a branch is miss-predicted or execution reaches the last row of the array with configured FUs the array is cleared and the configuration unit maps new instructions starting from the first row of the array. In order to save configurations for repeated execution all elements of the array are equipped with some memory cells which form configuration layers. Typically, 2, 4, 8, 16, 32, or 64 configuration layers are available. The array is quasi three-dimensional and its size can be written as `columns x rows x layers`.

With this extension it has to be checked before mapping new instructions if the next instruction to execute is equal to the first instruction in any of the layers. If a match is found, the corresponding layer is set to active and execution continues there. If no match is found, the least recently used configuration layer is cleared and used to store the new configuration. In all cases, the new values of registers calculated in columns are copied into the registers at the top of the columns.

To evaluate the architecture a cycle- and signal-accurate simulator has been developed. It uses the Portable Instruction Set Architecture (PISA), hence the simulator can execute the identical program files as the SimpleScalar simulation tool set [2] (but it is not based on it). Detailed information about the processor are given by Uhrig et al. [1] and Shehan et al. [3].

III. Towards an Improved Policy

Several basic terms of replacement strategies with respect to the GAP architecture are discussed in this section.

A. Measuring the Performance of a Replacement Strategy

To analyze the performance of a replacement policy, we suggest two measures. The total hit rate h_{total} of the layer subsystem, which is the number of accesses of layers which can be found in the configuration layers a_{hit} divided by the total number of accesses a_{total}. The total hit rate h_{total} can also be understood as the sum of the hit rate by re-accessing the identical configuration subsequently $h_{loop} = a_{loop}/a_{total}$, which is independent from the number of layers available, and the hit rate contributed by the layer subsystem $h_{layer} = a_{layer}/a_{total}$ for all other accesses:

$$h_{total} = \frac{a_{hit}}{a_{total}} = \frac{a_{loop}}{a_{total}} + \frac{a_{layer}}{a_{total}} = h_{loop} + h_{layer}$$

A replacement policy can influence only the hit rate of the layer subsystem h_{layer}. For a given benchmark, h_{loop} has the same value for all replacement strategies.

An optimal offline replacement algorithm (named OPT in the remainder) has been introduced by Belady [4] and it can be used as upper bound. In other words, no (online) replacement policy can achieve a better hit rate than this offline policy, which chooses the element for eviction that will be reused as the last one of all elements in the future.

Another offline algorithm has been mentioned by Temam [5] with the goal to maximize the number of instructions which can be accessed without cache misses. As upper bound for the performance of a replacement policy the algorithm OPT is a much more feasible measure because in the GAP, the penalty caused by activating the front-end of the processor when a new configuration must be build is much higher compared to the time, which is saved when some additional instructions can be found in a layer.

The second measure to evaluate a replacement policy is the performance of the whole system, which is e.g. described by the number of instructions executed per clock cycle (IPC).

B. Known algorithms and their performance

Figure 2 shows the simulated average total hit rate h_{total} of different well-known page replacement policies and the optimal offline policy OPT for 15 benchmarks of the MiBench Benchmark Suite [6] executed on the GAP with an array of width 12 and height 12 and a varying number of configuration layers. The hit rate achieved with one layer is equal to h_{loop} and replacement policies do not have any influence on it. The additional hit rate achieved with more than one layer is contributed by the layer subsystem, i.e. h_{layer}. As expected, LRU performs slightly better than FIFO. The out-performance of RANDOM over LRU is surprising as this replacement strategy behaves quite dumb by evicting random elements.

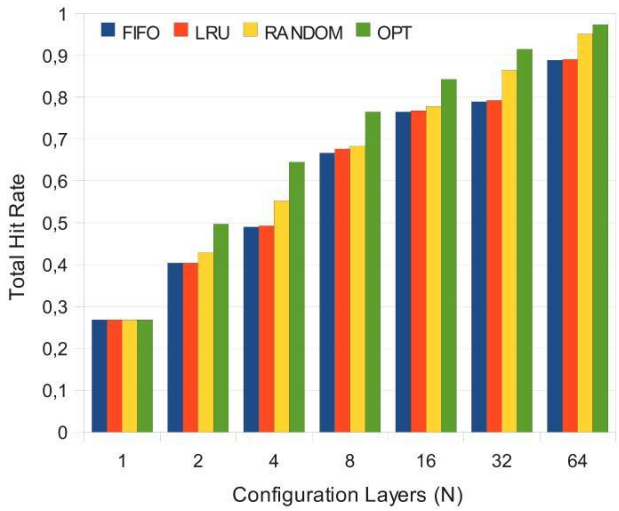


Fig. 2. Average total hit rate over 15 benchmarks for several numbers of configuration layers of GAP configured with an array of 12x12xN functional units

A more detailed view on the hit rate achieved with FIFO, LRU, RANDOM and OPT for the 15 benchmarks on GAP with a 12x12x16 array is presented in Figure 3. Six benchmarks reach a nearly optimal total hit rate close to 1.0 while benchmarks like `auto-qsort`, `netw-dijkstra`, `offi-stringsearch`, `secu-rijndael-encode` and others show a larger gap between the policies.

Table I shows the gap Δ between the performance of LRU and OPT. The second column is the difference Δ of the hit rates of LRU and OPT. In the third column, the achievable improvement with respect to the hit rate of LRU is shown. Closing this gap is the goal of our work.

C. Thrashing as major drawback of LRU

As first step towards an improved replacement policy we designed a graphical representation showing configuration layer accesses and the state of the layers. This type of graphics is referred to as *access plot* in the following. As example, the first 1000 accesses in the layer subsystem are shown in Figure 4(a) and 4(b). To render an access plot, the accesses to configurations are recorded in the GAP simulator. Next,

TABLE I
DIFFERENCE OF THE HIT RATE BETWEEN LRU AND OPT

Layers	Δ	Δ / LRU
1	0	0%
2	0.09	23%
4	0.15	31%
8	0.09	13%
16	0.08	10%
32	0.12	16%
64	0.08	9%
AVG	0.09	15%

each configuration is given an incrementing number (ID). Each access to a configuration is numbered, too. To plot an access, its coordinates are determined by its access number (X-axis) and the number of the accessed configuration (ID, Y-axis). If an already available configuration is accessed, it is colored green (or dark grey, if printed black/white only). If the configuration is not available it is colored red (or black, if printed black/white only). If the target configuration is the same as the one requested by the last access, the current configuration will not be modified. The reuse of the current configuration is not shown in the image because it has no interference with the replacement policy. The content of the layers at the time an access is done is displayed by vertical light gray pixels.

Although only a short part of the total program execution time is visible in Figure III-C it is sufficient to show some important facts. First, there is only a small number of different array configurations compared to the number of actually executed configurations which can be recognized by the width/hight relationship of the plot. Second, several patterns appear very often. They differ mostly by the number of configurations which they contain and the locality [7] of the accesses of the configurations:

- Sequentially executed code, e.g. the starting sequence: The reuse distance is extremely high or unlimited.
- Small loops, i.e. a small number of repeatedly executed configurations: The reuse distance is comparable to the number of configurations of the loop; the number of configurations is smaller than the number of configuration layers.
- Large loops, i.e. a larger number of repeatedly executed configurations: The reuse distance is comparable to the number of configurations of the loop; the number of configurations is larger than the number of configuration layers.
- Program phases: They consist of multiple loops and phases of sequentially executed code. As they contain lots of configurations the locality is only medium. The reuse distance can vary.

In a more detailed analysis we came to the conclusion that the well-known and already mentioned eviction polices show for these patterns always a very similar behavior. For sequential code, none of the strategies can achieve hits because configurations with infinite (or very long) time since a last usage are loaded. Small loops can be handled very well

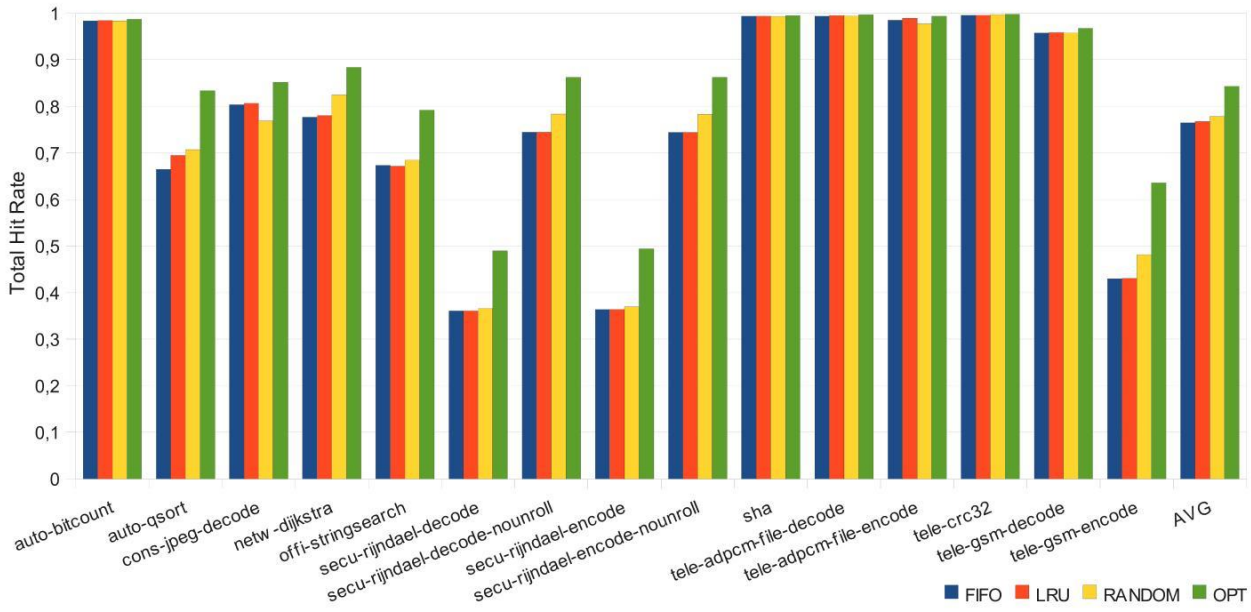


Fig. 3. Total hit rate for the benchmarks run on GAP with a 12x12x16 array

by LRU and OPT. RANDOM does not perform competitive because it evicts random configurations that might be needed again very soon. Large loops are the sticking point. If LRU is used, thrashing (see. e. g. [8]) can occur because the algorithm has to evict configurations that are part of the beginning of the loop and will be reused soon. The hit rate drops to zero. RANDOM performs better because it keeps at least some of the configurations of the loop in the layers. The OPT algorithm does the following: In every iteration of the loop it does not throw away some of the configurations, but keeps them for the next iteration. In Figure 5 the access plots for LRU, RANDOM and OPT can be seen. LRU produces only misses while RANDOM provides some and OPT a reasonable number of hits.

In short, for the replacement of the configuration layers thrashing is the main issue on the way to close the gap between LRU and OPT. Increasing the number of configuration layers would solve this problem to some extend but the number of layers is restricted by their hardware complexity and more complex applications would tend to thrashing again. Therefore, we look for a better replacement strategy to prevent thrashing or at least limit its effects.

IV. QdLRU Strategy

The main idea of this new approach is to explicitly add flags to those configurations which would cause thrashing. Those marked configurations are then immediately replaced by another configuration. In other words, they are inserted at the least recently used position instead of the most recently used position. Because the marked configurations are dropped quickly we call the strategy *quick drop LRU* (qdLRU).

As example assume the configurations c_0 to c_{47} are in a working set $W := [c_0, ..., c_{47}]$ and GAP has 32 configuration layers. Repeated executions of W would cause thrashing (when using LRU as replacement strategy) because the size of the working set $|W| = 48$ is larger than 32 which is the number of layers available in the GAP. If W is executed for a longer time with LRU then h_{layer} drops to 0 because of thrashing. The optimal offline strategy would buffer 31 of the 48 configurations, hence: $h_{layer} \approx 31/48 = 0.64583$. With qdLRU you get the same and optimal hit rate for this thrashing-risky situation.

A. Adding flags to instructions

QdLRU is a feedback-directed optimization. The basis to be able to calculate which instructions shall be marked is a trace of addresses of the first instructions of the configurations executed during a program execution. To get this trace file we use the cycle-accurate simulator available for the GAP.

The next step is to find the so-called configuration lines. A configuration line C, e.g. $C = \{c_0, ..., c_{47}\}$, represents one of the diagonal lines in Figure 5 and is very similar to a working set. To generate the set of all configuration lines $\mathbb{C} = \{C_0, ..., C_j\}$ and their usage counters the heuristics described in Listing 1 is used.

These configuration lines represent the working sets with the "smallest degree of reuse". To construct them, we assume that, whenever a configuration is already available in the working set and different from the last handled configuration, a branch back to the start of the current working set is performed.

Afterwards, several configurations within each configuration line are selected as candidates to be dropped quickly. For this, the configuration lines are split into two groups, one group $\mathbb{C}_{short}$ contains all lines whose length is smaller than the

Listing 1. Algorithm to configuration lines

```
input: list<configuration> trace

#define line list<configuration>
set<line> all_lines
map<line, int> line_counters

line current_line = {}
configuration last_configuration

foreach(configuration item in trace)
    if(item == last_configuration)
        // Do nothing
    else if(item ∉ current_line)
        current_line += item
        last_configuration = item
    else
        all_lines += current_line
        line_counters[current_line]++
        current_line = {}
        last_configuration = item
```

number of layers in the processor and the other group $\mathbb{C}_{long}$ contains all the other configuration lines, those configuration lines are too long to fit into the layers without evictions. With having prepared these groups the following algorithm is performed:

1) Select a configuration line `item` from $\mathbb{C}_{long}$.
2) Select from `item` the configuration with the least usage in $\mathbb{C}_{short}$, mark its first instruction.
3) Select all configuration lines from $\mathbb{C}_{long}$ where the number of all configurations minus the number of all marked configurations in the line is smaller than the number of layers of the processor. Move them to $\mathbb{C}_{short}$.
4) If $\mathbb{C}_{long}$ is not empty, restart the algorithm with step 1.

By this heuristic, we select configurations in a manner that they influence as little as possible the execution of configuration lines that fit into the layers. If a configuration line fits into the layers, but one of its configurations is marked, than this can humble the hit rate of this configuration line extremely.

In the last step, our post-link optimization tool GAPtimize (introduced in [9]) is used to mark the first instruction of the selected configurations with a special *drop quickly* flag. This flag directs the configuration layer subsystem of GAP to drop the configuration starting with the actual instruction quickly.

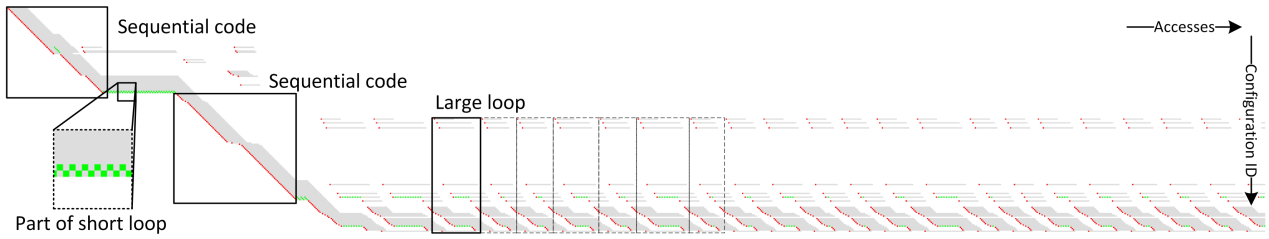
B. Executing the modified binary

When implementing qdLRU, changes are necessary both in hardware and in software. The changes in hardware are very simple. All which has to be done is to make sure that either a configuration beginning with a marked instruction is inserted in the least recently used position in the LRU access queue or that, when looking for a layer for eviction, it is first looked for a configuration layer starting with a marked instruction and then replacing this layer.

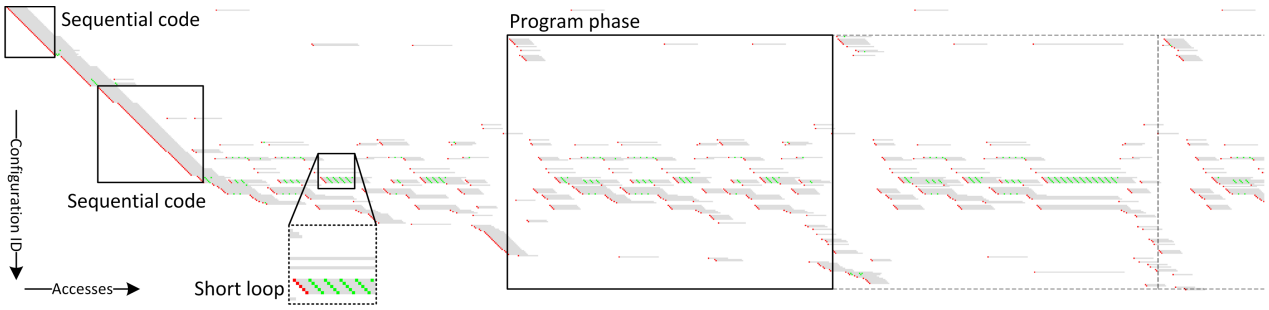
If a program is executed on the GAP which has not been optimized (and is hence without flags), then qdLRU behaves exactly like LRU, which still offers reasonable performance. This graceful degradation is one of the requirements of all techniques used for the GAP.

V. Evaluation

For the practical evaluation we rely on the cycle-accurate simulator which has been developed for the GAP and was extended to support qdLRU. As the hardware complexity of GAP can vary very much because of different sizes of its ALU array, we set it to a fixed size of 12 columns and 12



(a) Access plot for the first 1000 accesses of configuration layers for benchmark stringsearch



(b) Access plot for the first 1000 accesses of configuration layers for benchmark qsort

Fig. 4. Access plots (see Section III-C) for GAP with 12x12x16 array and LRU as replacement policy; some patterns are marked and labeled.

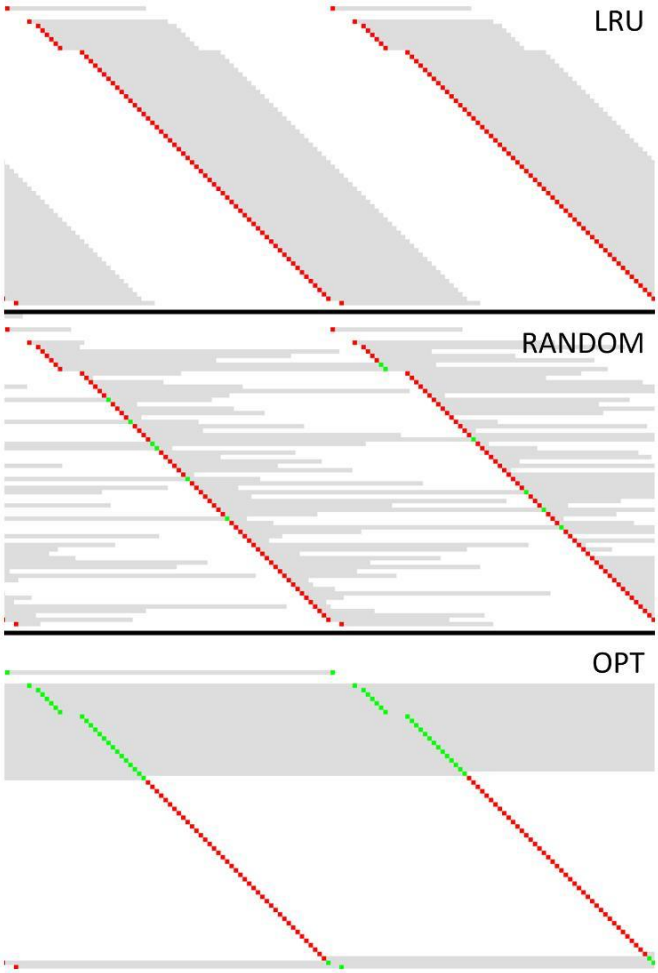


Fig. 5. Access plots (see Section III-C) for LRU, RANDOM and OPT (from top to bottom) dealing with a large loop (rijndael on GAP with 12x12x32 array)

rows, which is a realistic size. The performance of the qdLRU policy was evaluated using a varying number of layers. The configuration of the GAP is in the following abbreviated as `columns x rows x layers`.

We use integer-focused benchmarks from the MiBench benchmark suite [6] which have been compiled with GCC for the PISA instruction set architecture (see [2]) with optimizations turned on, i.e. `-O3`. These benchmarks are analyzed and modified with GAPtimize, our tool for feedback-directed post-link optimizations.

For 15 benchmarks, including benchmarks where we expected only little or no change, we achieve an improvement in performance measured by the IPC of 5.06% on average for qdLRU compared to LRU. The highest improvements are achieved for 32 and 64 layers, where we get improvements of 9.48% and 9.41% respectively (compare Figure 8). These values do not seem to be very brilliant which is mainly caused by the fact that for most of the benchmarks we cannot expect the improvement to be very high due to a very small gap

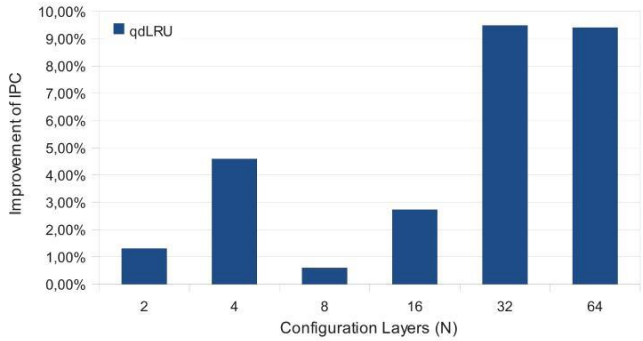


Fig. 6. Average IPC improvement for 15 benchmarks executed on GAP and configurations 12x12xN with qdLRU compared to LRU

between LRU and OPT. Details are shown in Figure 6.

In the top part of Figure 7, more details on the IPC are displayed for some selected benchmarks where thrashing is an issue. The most impressive numbers can be seen for the benchmarks secu-rijndael-decode and secu-rijndael-encode with a maximum improvement of 390 % for the IPC achieved with qdLRU compared to LRU.

This improvement of the IPC is mainly based on the improved hit rate of the layer subsystem. The total average improvement of the hit rate over 15 benchmarks and all configurations is 0.06. This average hit-rate improvement seems to be small but it has to seen in relation to the maximal possible improvement (0.09) which can be achieved with the optimal algorithm OPT (see Section III-B).

For selected benchmarks which might cause thrashing with LRU the total hit rate can be seen in the bottom part of Figure 7. Here again, the benchmarks secu-rijndael-encode and secu-rijndael-decode show supreme results as thrashing is here a very critical problem when using LRU.

In Figure 8 we show the average performance of LRU and qdLRU compared to OPT and RANDOM. QdLRU shows better performance than LRU and RANDOM. The gap between LRU and OPT can be closed with qdLRU by 65.97 % on average, varying between 48,38 % for 2 layers and 78,05% for 32 layers (see Figure 9).

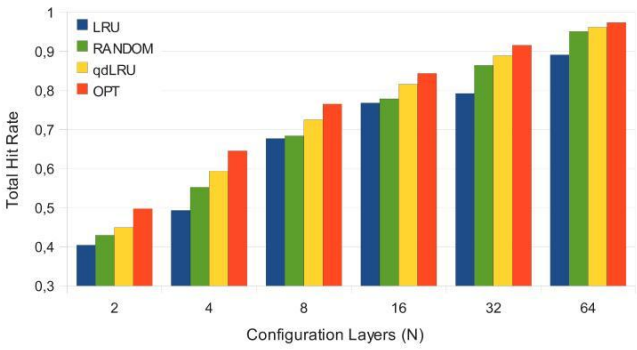


Fig. 8. Simulated total hit rate for LRU, RANDOM, qdLRU and OPT (average over 15 benchmarks, GAP with 12x12xN functional units)

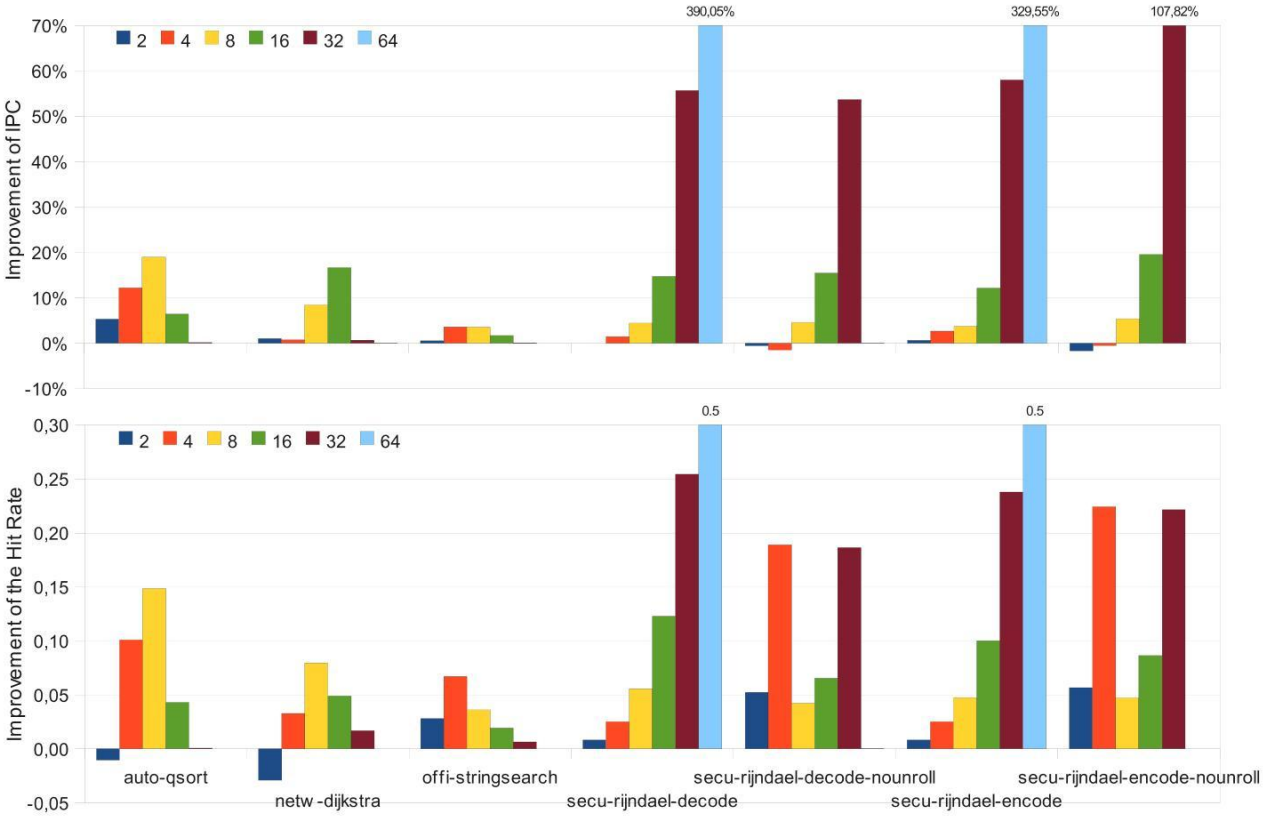


Fig. 7. Relative improvement of the IPC (top) and improvement of the hit rate (bottom) for selected thrashing-risky benchmarks run on GAP with configurations 12x12xN for qdLRU compared to LRU

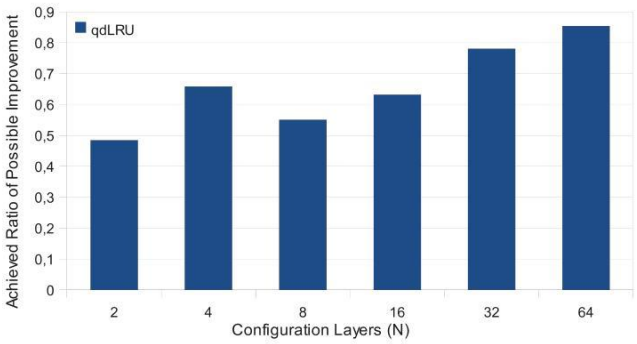


Fig. 9. Ratio of the gap between qdLRU an OPT which can be covered by qdLRU (average over 15 benchmarks, GAP with 12x12xN functional units)

If the memory latency (which has been set to 24 clock cycles) is increased, we expect the effect of an improved hit rate to have stronger implications on the system performance. On the other side, if the size of the also available instruction cache is increased, the effect of increasing the hit rate will decrease because the penalty to load instructions from the instruction cache will decrease due to less instruction cache misses.

VI. Related Work

As mentioned before the configuration layers of the GAP are used to buffer configurations. They work like a cache and a replacement policy is implemented to find the element to evict if space is needed to load a new one. For the GAP, finding a suitable replacement strategy for the configuration layers has not yet been explored. Nevertheless, related work can be found mainly dealing with caches in general and trace caches.

The main difference of the replacement problem for configuration layers compared to general caches is the extremely small number of configurations layers compared to the large number of lines in caches. Because of this, thrashing-risky situations where the working set is larger than the number of available lines are much more frequent for configuration layers than for caches.

Our goal is to find a lightweight solution which can make use of the software infrastructure already available and used for other platform-specific code optimizations, e.g. static speculation [9]. Because of this, our attention is drawn to techniques using both hardware- and software techniques.

As hardware-only solutions, two classes of strategies are known. The first are well-known algorithms which can be implemented with small or reasonable effort in hardware. Some of those strategies are FIFO, RANDOM, WsClock [10] and LRU. From these strategies, LRU is deemed to be the superior

one. Together with OPT as upper bound the performance of LRU, FIFO and RANDOM have been compared for our situation in Section III-B.

The second class of algorithms are the Dynamic Insertion Policy (DIP) proposed by Qureshi et al. [11] and the Shepherd Cache proposed by Rajan etc al. [12]. Both share the property that they require additional hardware effort. In our experiments, we also got for our particular situation performance numbers at most comparable to LRU for the Shepherd Cache. The DIP is only applicable if it can select between LRU and BIP with extreme parameters to prevent thrashing. The suggested approach to divide the configuration layers into two sets does not seem to be applicable due to the small number of configuration layers. The small number of lines prevents using strategies like ARC [13] where the lines are split into two sections and handled in different ways.

Some other techniques have also been proposed (see e.g. [14]) but most of them either require large changes of the hardware and/or are not supposed to work well because the low number of layers available in the GAP normally restricts the eventual gain in performance caused by replacement strategies.

Trace caches as introduced by Rotenberg et al. [15] work for superscalar processors very similar to the configuration layers because they are used to buffer parts of a program flow, too. To our knowledge, nobody has yet been working on thrashing situations in this context.

VII. Conclusion and Future Work

We introduced a software-supported replacement strategy for the configuration layers of the GAP processor, which are used like a trace cache to buffer instructions sequences ready for execution. So far, LRU is used as replacement strategy which offers an unsatisfying performance for several benchmarks. Strangely enough, LRU shows for some benchmarks even worse performance than RANDOM, a strategy evicting a random element. The main reason for this is thrashing, which can happen if the elements of a working set are processed repeatedly and sequentially, i.e. there is a huge degree of locality, and the set contains more configurations than the GAP provides configuration layers. In this case, the hit rate achieved with LRU collapses.

To overcome this issue, we proposed a replacement strategy called qdLRU and a heuristic to approximate the working sets in software. Based on working sets we select some configurations which are evicted immediately from the configuration layers. With this, we can draw the behavior of qdLRU nearer to the optimal strategy OPT. The performance measured by the IPC for qdLRU is on average 5.06% higher than the performance achieved by LRU. A peak improvement of 390% is gained for secu-rinjdael-decode caused by a peak improvement of the hit rate of 0.5.

This approach could be adapted for all situations in which a replacement strategy is needed for a small number of complex elements with many thrashing-risky situations. The introduced strategy requires only very little changes of the hardware when LRU has already been implemented. It also supports graceful degradation back to LRU.

As future work, we propose to work on the detection of working sets. The rule which has been introduced is simple and effective. Nevertheless, there are situations where this rule cannot find a sufficient solution. Hence, to find better solutions it should be thought about the scope of the working sets. From our point of view, it is important that the configurations in a working set should be executed repeatedly in the same order. If this restriction is weakened, the scope of working sets could be enlarged which must be handled carefully but might lead to further improved results. Concluding, it might be possible to find better solutions with biologically inspired algorithms, e.g. ant algorithms or genetic algorithms. Linear programming should also be taken into consideration.

References

[1] S. Uhrig, B. Shehan, R. Jahr, and T. Ungerer, "The two-dimensional superscalar gap processor architecture," *International Journal on Advances in Systems and Measurements*, 2010.

[2] D. Burger and T. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, June 1997.

[3] B. Shehan, R. Jahr, S. Uhrig, and T. Ungerer, "Reconfigurable grid alu processor: Optimization and design space exploration," in *Proceedings of the 13th Euromicro Conference on Digital System Design (DSD) 2010, Lille, France*, 2010.

[4] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems*, vol. 5, no. 2, pp. 78–101, 1966.

[5] O. Temam, "Investigating optimal local memory performance," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 218–227, 1998.

[6] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *4th IEEE International Workshop on Workload Characteristics*, pp. 3–14, December 2001.

[7] P. J. Denning, "The locality principle," *Commun. ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[8] P. Denning, "Thrashing: its causes and prevention," in *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. New York, NY, USA: ACM, 1968, pp. 915–922.

[9] R. Jahr, B. Shehan, S. Uhrig, and T. Ungerer, "Static speculation as post-link optimization for the grid alu processor," in *Proceedings of the 4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)*, 2010.

[10] R. W. Carr and J. L. Hennessy, "WSCLOCK - a simple and effective algorithm for virtual memory management," in *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1981, pp. 87–95.

[11] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 381–391.

[12] K. Rajan and G. Ramaswamy, "Emulating optimal replacement with a shepherd cache," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 445–454.

[13] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[14] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Where replacement algorithms fail: a thorough analysis," in *CF '10: Proceedings of the 7th ACM international conference on Computing frontiers*. New York, NY, USA: ACM, 2010, pp. 141–150.

[15] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 24–35.