Dynamic Tuning of Configurable Architectures: The AWW Online Algorithm

Chen Huang, David Sheldon, and Frank Vahid*
Department of Computer Science and Engineering, University of California, Riverside, USA
*Also with the Center for Embedded Computer Systems, UC Irvine
{chuang/dsheldon/vahid}@cs.ucr.edu

ABSTRACT

Architectures with software-writable parameters, or configurable architectures, enable runtime reconfiguration of computing platforms to the applications they execute. Such dynamic tuning can improve application performance, as well as energy. However, reconfiguring incurs a temporary performance cost. Thus, online algorithms are needed that decide when to reconfigure and which configuration to choose such that overall performance is optimized. We introduce the adaptive weighted window (AWW) algorithm, and compare with several other algorithms, including algorithms previously developed by the online algorithm community. We describe experiments showing that AWW results are within 4% of the offline optimal on average. AWW outperforms the other algorithms, and is robust across three datasets and across three categories of application sequences too. AWW improves a non-dynamic approach on average by 6%, and by up to 30% in low-reconfiguration-time situations.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Adaptable architectures, heterogeneous systems.

General Terms: Algorithm, Performance, Design.

Keywords: Configurable architecture, tuning, FPGAs, cache, dynamic optimization, runtime configuration, online algorithms.

1. INTRODUCTION

Configurable architectures enable runtime tuning of computing platforms to their running applications. Tuning can substantially improve performance or energy.

Runtime configurable architecture constructs include memory hierarchies whose caches may be shut down or have their total size, line size, associativity, or replacement policy adjusted, buses that may be resegmented or have their widths or protocols adjusted, soft-core processors on field-programmable gate arrays (FPGAs) that may be reinstantiated with different datapaths or any processor that may have certain datapath units shut down, processors with adjacent FPGA units that may have particular coprocessors loaded, and scalable processors whose supply voltage and frequency may be varied.

Tuning a configurable architecture to an application can substantially improve performance or energy. For a configurable architecture, each distinct application that runs on that architecture may run most efficiently with a particular configuration, running inefficiently in other configurations. Figure 1 shows execution time for three EEMBC embedded benchmark applications running on a SimpleScalar MIPS processor with a 2 Kbyte direct-mapped instruction cache having three possible line size configurations: 16, 32, or 64 bytes. An application with much spatial locality (e.g., TBLOOK01) is faster with the largest line size, while another application (PUWMOD01) is faster with the smallest line size, and a third runs fastest using the middle line size. If these three applications run on one processor, reconfiguring the cache for the currently-executing application may yield 40% better performance than using a single configuration for all three applications. Figure 2 illustrates running a particular application sequence, each instance shown on the x-axis with the applications' first letter, with a fixed 32-byte line size versus with a reconfigurable line size. The figure shows how total runtime for the latter may be less if reconfiguration time is fast, but may actually be more if reconfiguration is slow.

While some configurable architecture constructs may be reconfigured with little runtime reconfiguration overhead, such as voltage scalable processors, other constructs require non-negligible reconfiguration time. For example, reconfiguring a memory hierarchy may involve flushing of dirty cache words. Reinstantiating an FPGA soft-core processor may involve time to save the processor's execution context, swap in a new partial FPGA bitstream, and restore the processor context.

Figure 1: Applications' best configurations typically differ.

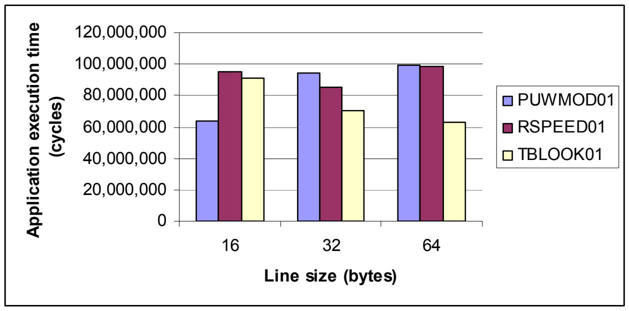
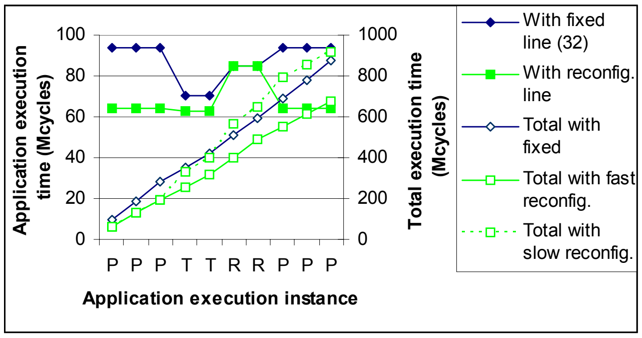


Figure 2: Reconfiguring can lead to lower total execution time, if reconfiguration time isn't too slow.



A problem is to determine, as applications arrive for execution, when to reconfigure an architecture and to which configuration. Reconfiguring incurs performance loss due to reconfiguration time, but also performance gains due to tuned application execution. The gain must outweigh the loss for benefit. If the schedule of application executions is known a priori, and with pre-determined performance information for every application in each configuration, a straightforward dynamic programming algorithm can determine the optimal reconfiguration schedule.

For many modern embedded systems, however, the schedule of application executions is not known a priori, but rather is determined by end-user usage patterns. In this case, an algorithm must be used that makes reconfiguration decisions with incomplete information about the future schedule of application executions.

In this paper, we cast the reconfiguration problem as an online optimization problem, namely, as a metrical task system. The key contribution is the Adaptive Weighted Window (AWW) algorithm, which obtains better optimization than previous algorithms, and is robust across a broad range of possible scenarios of reconfiguration time versus application runtimes and of application schedules. We define the problem, discuss related work, and describe previous and new algorithms including AWW. We describe experiments using the algorithms on real and synthetic benchmarks, showing the AWW algorithm to usually achieve results closest to optimal. The algorithm is also simple to implement. Thus, the AWW algorithm should be suitable for dynamic tuning of a wide range of existing and future configurable architectures.

2. ARCHITECTURE RECONFIGURATION

2.1 Problem Definition

We define the *architecture reconfiguration problem* as follows. Given are:

- The application set *A = {a1, a2, ..., an}* and a set of architecture configurations *C = {c1, c2, ... cm}*.
- An execution time matrix *E* of dimension *n x m*. *E(i,j)* is the execution time of application *ai* on configuration *cj*.
- The reconfiguration time *R* to change from any configuration *ck* to any other configuration *cl*.

Runtime consists of a sequence of application executions *S*, such as *S = <a2, a1, a1, a3, a2, a1>*, but typically much longer with hundreds or thousands of application instances. We define *S[k]* as the application type at position *k* in the application sequence *S*; in the above sequence, *S[1]* is *a2*, *S[2]* is *a1*, and *S[3]* is *a1*.

Assumed is that each application has a start and finish and cannot be preempted. Iterating applications can be accounted for by redefining the application as a single iteration and then scheduling repeated executions of the redefined application.

The problem is to choose a configuration for every application in the sequence to minimize total time *T*, yielding a configuration schedule. Every configuration change in that schedule is a reconfiguration incurring time *R*. Total time *T* is the sum of application execution times on the corresponding configuration in the schedule, plus the time for all reconfigurations.

2.2 Limitations

The above problem definition assumes that each application's execution time on each configuration is known. For some architectures, the number of possible architecture configurations could be large and thus hard to precompute, though this situation can be alleviated by using a configuration subset that reasonably covers the configuration space [14]. A priori knowledge also diverges from one research avenue in configurable architectures that seeks to make the configurations and tuning invisible to the application designer. However, dynamically collecting runtime data on applications and architectures may help address that problem. A second limitation is that applications cannot be preempted, a topic for future work. A third limitation is the assumption that application execution times on a configuration, and reconfiguration times, are constants. However, execution times for an application on a particular configuration may vary depending on runtime data. Reconfiguration times may differ among pairs of configurations (e.g., adding a datapath unit may be costlier than removing one), and further could depend on which applications ran earlier (e.g., more cache flushing may be needed after some applications than after others). Treating these times as constants involves considering worst cases or average cases instead. Despite the limitations, the definition is close to many task-based problem definitions, and seems suitable for a variety of embedded systems. Future work may seek to extend the definition to other scenarios.

2.3 Metrical Task System

An online problem is one that must be solved given data piece by piece, in contrast to offline problems in which all input data is available. A metrical task system, defined in [5], is a well-known formulation of a class of online problems. The problem involves a task system *(S,d)* for processing sequences of tasks. *S* is a set system states, and *d* is a cost matrix where *d(i, j)* is the cost of changing from state *i* to state *j*, assumed to satisfy the triangle inequality, and assumed to have 0s on the diagonal. In a metrical system, state transition costs are symmetric, i.e., *d(i, j)* equals *d(j, i)*. The cost of processing a task depends on the system state, and thus a task can be viewed as a vector *T=(T(1), T(2), ..., T(j))*, where *T(j)* is the (possibly infinite) cost of processing the task while in state *j*. A schedule for a sequence *T1, T2,..., Tk* of tasks is a sequence *s1, s2,...,sk* of states where *si* is the state in which *Ti* is processed. The cost of a schedule is the sum of all task processing costs and the state transition costs incurred. An on-line scheduling algorithm is one that chooses *si* only knowing *T1T2…Ti*.

The architecture reconfiguration problem (AR) can be mapped to the metrical task system problem (MTS). AR's applications correspond to MTS' tasks, and configurations to states. Each row of AR's execution time matrix corresponds to an MTS *T* vector. AR's reconfiguration time *R* can be used to fill in MTS' cost matrix with identical values (with the diagonal 0s), thus satisfying symmetry and triangle inequality requirements. AR's sequence of application executions corresponds to MTS' task sequence, AR's schedule to MTS' schedule, and AR's total time to MTS' cost.

3. RELATED WORK

A common configurable architecture parameter involves caches, which contribute greatly to system performance and power. Albonesi [1] proposed a configurable cache in which ways could be shut down. Malik [11] added the ability to configure each way as instruction, data, both, or off, for unified caches. Zhang [15] further added the ability to concatenate ways into one larger way, and to vary the line size. While most previous cache tuning work emphasizes static tuning of such caches, Balasubramonian [2] proposed a dynamic cache tuning approach involving enlarging or shrinking a configurable cache based on cache miss thresholds and program phase changes detected by counters, and measuring whether changes made improvements. Gordon-Ross [8] developed a dynamic cache tuning approach for Zhang's configurable cache, intermittently activating a previously-developed cache tuning search heuristic, with activation frequency governed by a feedback control method. Both those dynamic approaches differed from the approach in this paper by their not assuming awareness of specific applications executing or of application execution times on possible configurations. Their advantage versus this paper's approach is transparency, at the cost of less possible optimization.

Configurable processors have also been proposed. The most common involves voltage/frequency scalable processors. Sekar [13] extends such platforms by also introducing configurability of data, which can be in an on-chip memory or off-chip, and by introducing a custom algorithm for dynamically partitioning data and choosing voltage/frequency based on the presently active task set. Kumar [9] proposes having multiple diverse binary-compatible cores on a single general-purpose processor chip. Applications would be scheduled onto a specific core based on the best match as well as on current workload, and thus remapping applications upon arrival of a new application can be viewed as reconfiguration.

Configurable bus research includes Lahiri's [10] in which a system components' communication transactions are monitored, and adjustments then made to configurable bus parameters like priorities and DMA mode.

The metrical task system problem has been the focus of much online algorithm research since its definition in 1992 [5]. Many such works focus on developing K-competitive algorithms – algorithms guaranteed not to be worse than a factor K from the offline optimal – or extending the problem definition (e.g., [3][6]).

4. ALGORITHMS

We introduce several algorithms for architecture reconfiguration. In the complexity analyses, *n* is the number of application types, and *m* the number of configuration types. Complexity is defined as deciding on a configuration for one application instance in *S*.

4.1 Offline Optimal

The offline optimal algorithm is given the complete application sequence *S*. A dynamic programming algorithm can be formed, using the following recurrence relation to fill the table of Figure 3:

$$L_j^k = \min\{L_j^{k-1}, \min_{h \neq j}(L_h^{k-1}) + R\} + E(S[k], j)$$

L_j^k is the minimal total execution time, up to and including the application in position *k* in *S*, for choosing configuration *j* to execute that application. In other words, the minimal execution time at position *k* for configuration *j* is obtained either by using *j* for the previous application (the L_j^{k-1} term) or by using a different configuration *h* having the smallest time (the L_h^{k-1} term) and reconfiguring to *j* (thus incurring time *R*), whichever is less, plus the current application's time on configuration *j* (the $E(S[k], j)$ term). Stored with each *L*, though not shown, is the previous configuration (either *j* or some *h*) as determined above, so that the algorithm records the configuration schedule to obtain the minimal time. The time complexity of this offline optimal algorithm is *O(m²)* for each application in the input sequence, or *O(Km²)* for the entire sequence, where *K* is the total length of the input sequence *S*.

Figure 3: Table for dynamic programming algorithm.

	S[1]	S[2]	...	S[K]
c1	L_1^1	L_1^2		L_1^K
c2	L_2^1	L_2^2		L_2^K
...				
cm	L_m^1	L_m^2		L_m^K

4.2 Greedy

A simple online algorithm always changes to the configuration that is best for the current application, ignoring reconfiguration time. Such an algorithm is useful primarily for comparison purposes, representing a naive value to compare to along with the other extreme of the "best case" value of the offline optimal algorithm. The time complexity for the Greedy algorithm is *O(1)*.

4.3 Work Function

The Work Function algorithm [5], defined for MTS, is similar to the offline optimal dynamic programming algorithm, but for an application sequence up to and including the current application only. The algorithm computes the dynamic programming table of Figure 3 incrementally as each application is encountered, choosing the configuration having the lowest L_j^k execution time for the current *k*. Time complexity is *O(m²)*.

4.4 Marking Algorithm

The Marking algorithm [5] was also defined for MTS. It maintains a counter for each configuration, and uses phases. At a phase start, counters are reset to 0, and a random configuration *cj* is selected. When an application *ai* runs on configuration *cj*, the counter for configuration *cj* is incremented by the execution time of *ai* on *cj*, namely by *E(i,j)*. If the counter for *cj* reaches some threshold *X*, the configuration is changed to a random configuration whose counter is less than *X*. If no such configuration exists, a new phase is started. The intuition of this algorithm is to rotate among configurations (since the best is not known), staying longer in configurations that execute applications fast and thus don't have their counters increased rapidly. Time complexity is *O(m)*.

4.5 Window Algorithm

Online algorithms defined for MTS typically focus on the K-competitive ratio, which guarantees results no worse than K times the offline optimal for theoretically worst case ("adversarial") input sequences. Our goal instead was to perform well for typical, while broad, input sequences (as theoretically worst case inputs are rare in practice). We thus developed additional algorithms.

The first algorithm we developed for the architecture reconfiguration problem assumes temporal locality, meaning that the future will be similar to the recent past. The number of applications *s* considered into the past is called the window size. The Window algorithm, shown in Algorithm 1, finds the configuration that would have yielded the smallest time for the application sequence appearing in the previous *s* applications in the application sequence, followed by the current application.

Algorithm 1: Window Algorithm

```
Window (k, s) returns configuration j
    s: window size.  k: current position in sequence S
  for each configuration j
      $T_j^k \leftarrow \sum_{h=k-s+1}^{k} E(S[h], j)$
      if $j \neq$ Current configuration then $T_j^k \leftarrow T_j^k + R$
  return j corresponding to minimum $T_j^k$ obtained
```

T_j^k is the time of configuration *j* to execute the current window's applications. The time complexity of the algorithm is *O(m)*. The following relation, $T_j^k = T_j^{k-1} - E(S[k-s], j) + E(S[k], j)$, keeps complexity low by incrementally computing the next *T* from the previous *T*, avoiding complexity proportional to *s*.

4.6 Two-Way Window (TWW) Algorithm

A possible improvement to the Window algorithm attempts to more accurately predict the future. The Two-Way Window (TWW) algorithm maintains an application transition matrix *M(y,z)*. Each entry counts the number of times application *az* has followed *ay*. Given the current application *ai*, the algorithm determines the most probable next application *aj* by examining the matrix counts. With *aj*, the algorithm determines the most probable *ak*, and so on, for the desired future window size. The algorithm then determines the best configuration for the window that includes the past *u* applications, the current application, and the future *v* (predicted) applications. We set *u* and *v* to 10. The time complexity of TWW is *O(mv),* where *v* is the size of the future window.

4.7 AWW Algorithm

For the Window algorithm, choosing the best window size is challenging. A larger size is more stable, but a smaller size gives more weight to the near past, which may be more likely to reflect the near future. A hybrid uses a larger window while giving more weight to the recent window part. Such weighing can be achieved by multiplying application *ax*'s execution time by $z^{distance}$, where *z* is a constant between 0 and 1, and *distance* is the number of applications between *ax* and the current application. Thus, the further back in time that application *ax* was run, the less influence it has. We call this algorithm the Weighted Window algorithm.

Choosing the best *z* is hard. If execution times are large relative to reconfiguration time, a small *z* is preferred to give more weight to the current application. In other words, small reconfiguration times enable frequent reconfigurations. But, if reconfiguration time is large, a large *z* (near 1) is preferred, to resist frequent reconfigurations, looking more evenly at the past application sequence, presumably predicting the long term future.

The discussion leads to the idea of defining *z* as *(1-y)*, where *y* is adapted to the extent to which execution times are greater than the reconfiguration time *R*. We define *y* as the fraction of application/configuration pairings whose execution time exceeds the reconfiguration time, namely:

for all i (application types) and j (configuration types)
y = (#E(i,j) such that E(i,j)>R) / (i *x* j)

Because *z* is thus adapted to the application execution times and reconfiguration time, we refer to this as the *adaptive weighted window (AWW)* algorithm, shown in Algorithm 2.

Algorithm 2: AWW Algorithm

```
AWW (k, s) returns configuration j
    s: window size.   k: current position of S
  y = (#E(i, j) | E(i, j) > R) / (i*j)
  for each configuration j
      $T_j^k \leftarrow \sum_{h=k-s+1}^{k} E(S[h], j) \cdot (1-y)^{k-h}$
      if $j \neq$ Current configuration then   $T_j^k \leftarrow T_j^k + R$
  return   j corresponding to minimum $T_j^k$ seen
```

Tj is the time of configuration *j* to execute the application in the window. The time complexity is *O(m).* The relation:
$T_j^k = (T_j^{k-1} - E(S[k-s], j) \cdot (1-y)^{S-1}) \cdot (1-y) + E(S[k], j)$
enables incremental computation of T from the previous T, to avoid complexity proportional to the window size *s*.

An alternative approach to adapt *z* could be to define *y* as the geometric mean of the difference of application execution times and the reconfiguration time *R* (other definitions are possible).

Computing *z* based on application execution times and the reconfiguration time has the added benefit of adapting to changes if those times were dynamically recorded and the *E* and *R* items were dynamically updated. We did not do such dynamic updates in our experiments, but this may be an interesting avenue for future work.

5. EXPERIMENTS

We compared the developed algorithms on four data sets (applications and configurations), described in upcoming subsections. For each data set, to evaluate the algorithms across a spectrum of application sequence scenarios, we created a generator capable of creating three categories of application sequences:

- Random: Applications are randomly inserted into the sequence.
- Biased: We defined two percentages A and B, and then generated the sequence such that A percent of the applications executed B percent of the time. We used A=20% and B=80%.
- Periodic: We defined a length T, and generated a random subsequence of length T that then repeats. We used T=15.

Each sequence's length was 1,000. For all experiments, because sequences involve some random ordering, we generated 10 sequences, and report the average.

We developed a simulator in C++ that reads all problem input and an application sequence, and that determines the total application execution time that would result from running each algorithm on the sequence. The running time of the algorithms themselves is not included in that execution time, being negligible for our particular application and platform scenarios; for other scenarios, algorithm runtimes may be significant, especially for the Work Function algorithm whose complexity is quadratic.

5.1 Cache Tuning

We obtained data from Gordon-Ross [8] for a dynamically configurable cache. The data consisted of application execution times for 36 applications on all 18 possible configurations of a configurable cache (i.e., 648 data items). The benchmarks were from Powerstone, MediaBench and EEMBC. Example cache configurations included a 2 Kbyte, 16-byte line size, 1-way (direct mapped) cache and an 8 Kbyte, 64-byte line size, 4-way set-associative cache, and various configurations between these extremes. Data was derived through exhaustive simulation using SimpleScalar. Rather than choosing one cache reconfiguration time, to evaluate the algorithms across a range of potential platforms, we considered cache reconfiguration times (mainly from cache flushing) from 0.01 ms to 100 ms. Application runtimes ranged from 20 ms to 500 ms. Figure 4 summarizes results.

5.2 FPGA Soft-Core Tuning

We obtained data from Sheldon [12] for a MicroBlaze configurable FPGA soft-core processor. The data consisted of 15 applications on 64 configurations, or 900 data points. Benchmarks were from Powerstone and EEMBC. Configurations included the base processor alone, the base processor plus all optional datapath components (floating-point unit, multiplier, barrel shifter, cache, etc.), and various configurations in between these extremes. The data was originally collected for offline tuning, but could be utilized in a dynamic tuning approach on an FPGA supporting runtime partial reconfiguration. To evaluate across a range of FPGA platforms, we considered

Figure 4: Cache tuning: Resulting execution times (seconds) for the various online algorithms for random (left), biased (center), and periodic (right) application sequences, for reconfiguration times ranging from 0.01 ms to 100 ms.

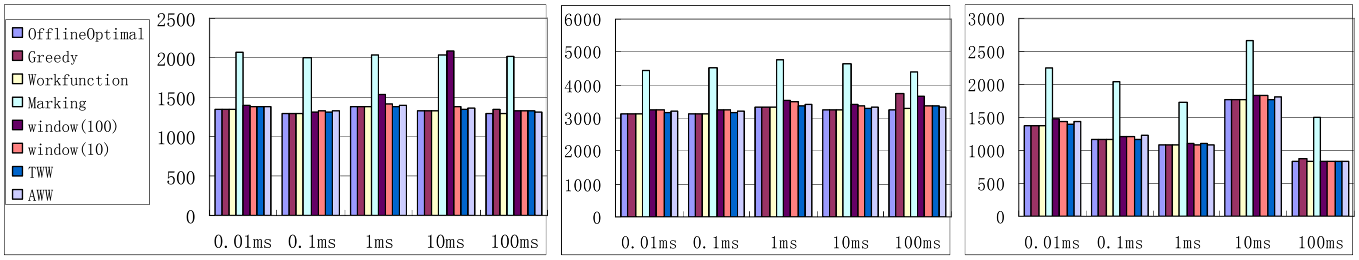


Figure 5: FPGA soft-core tuning: Resulting execution times (seconds) for the various online algorithms for random (left), biased (center), and periodic (right) application sequences, for reconfiguration times ranging from 5 ms to 200 ms.

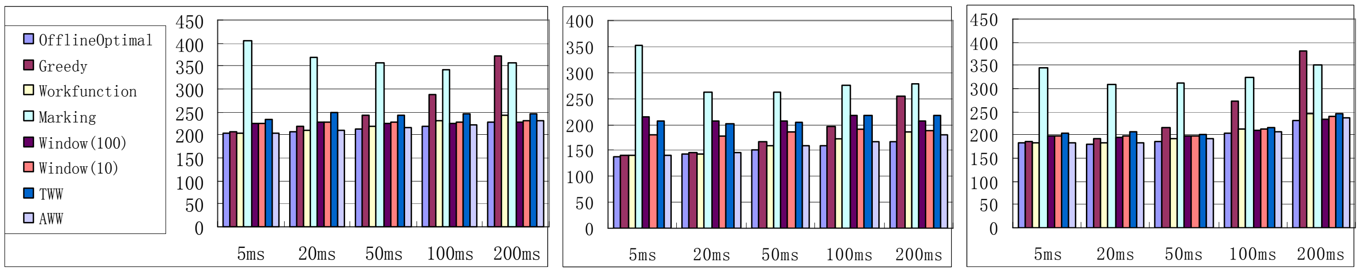
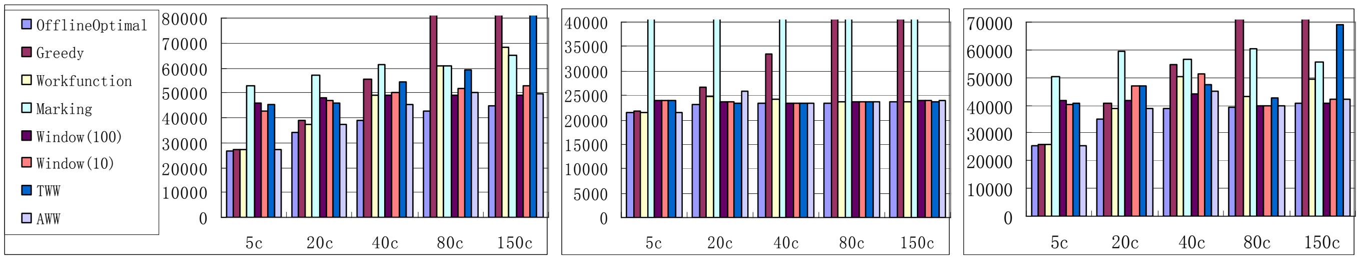


Figure 6: Synthetic cases: Resulting execution times (cycles) for the various online algorithms for random (left), biased (center), and periodic (right) application sequences, for reconfiguration times ranging from 5 cycles to 150 cycles. Certain values extend off the chart top.



reconfiguration times (due to FPGA bitstream loading and processor context store/restore) from 5 ms to 200 ms. Application runtimes ranged from 60 ms to 2,000 ms. Figure 5 shows results.

5.3 Synthetic Datasets

To further evaluate the algorithms across a range of scenarios, we generated a randomized dataset with 5 applications and 10 configurations, having cost matrix E of {38,58,394,36,69}, {91,73,98,72,49}, {18,93,58,37,44}, {83,27,38,55,36}, {78,35,84,59,19}, {89,45,28,50,22}, {29,40,95,38,66}, {85,16,48,67,34}, {20,49,87,239,60}, {29,59,27,86,90}}. The random nature of the data provides a greater challenge to tuning algorithms. Figure 6 summarizes results. A second dataset involved 10 applications on 4 configurations. Results were similar to Figure 6, and thus omitted for space reasons.

5.4 Evaluation

For all datasets, AWW performed best, within 2% of the offline optimal for the real data sets and 6% for the synthetic datasets, or 4% overall. AWW performed well across all three application sequence categories, with its worst case being 12% from optimal for the random (and hence most difficult) sequence category with synthetic data (which was also random). TWW sometimes very slightly outperformed AWW, but such benefit was outweighed by significantly worse TWW results for certain scenarios. The Window algorithm, with sizes of 10 or 100, was often close to AWW, but did poorly for the biased FPGA soft-core scenarios and for the synthetic (more randomized) datasets. The Work Function algorithm was usually close to AWW, but did poorly for the synthetic datasets. Marking was not competitive, which can be explained by its seeking to improve the theoretical worst case. The Greedy algorithm's inferiority was apparent for the larger reconfiguration times, when its "always reconfigure" approach hurts the most.

We recorded the number of reconfigurations incurred by each algorithm, summarized for the FPGA soft-core experiments in Figure 7. Observing the number of reconfigurations provides insight into each algorithm's behavior. AWW tends to match the offline optimal algorithm's number, sometimes only slightly different. Work Function often performed many more reconfigurations, while still remaining competitive in total execution time in many cases. Greedy of course performed the most reconfigurations, reconfiguring every time the current application's best configuration differed from the previous application's best configuration.

To determine algorithm runtimes, we ran each for a 50,000 application sequence on a 2 GHz PC. The offline optimal required 256 ms and Work Function 248 ms. AWW and Window(10) required

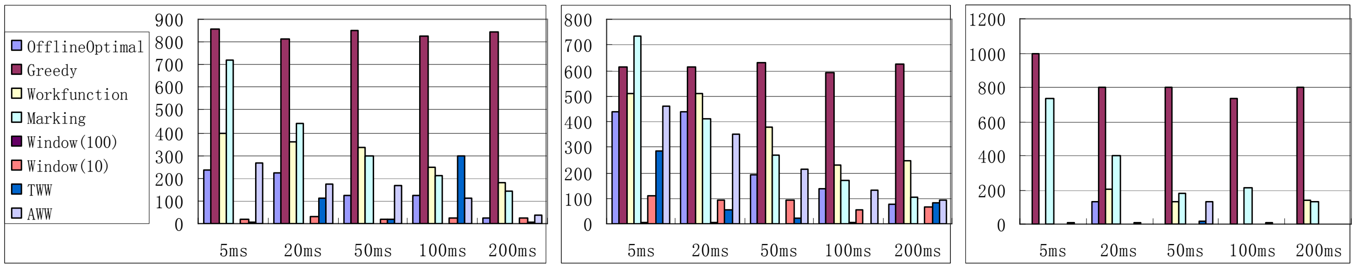


Figure 7: Total number of reconfigurations for random (left), biased (center), and periodic (right) application sequences, with reconfiguration times ranging from 5 ms to 200 ms, for the FPGA soft-core tuning experiments.

4 ms, while Marking and Window(100) required 8 ms. TWW required 840 ms. Greedy was near 0 ms. AWW not only outperformed the other algorithms, but was faster than all but Greedy. AWW required only 4 ms/50,000 = .08 microseconds to decide whether to reconfigure for the current application.

For further comparison, we implemented a non-dynamic configuration algorithm that stays in the single configuration having the lowest average runtime across all application types. AWW was 6% better on average, and up to 30% better for small reconfiguration times, when frequent reconfiguring can be done with less time overhead. For large reconfiguration times, the non-dynamic algorithm nearly equaled AWW, slightly better (1%-2%) in some cases. Plots are omitted for space reasons.

6. CONCLUSIONS

Dynamic tuning will become increasingly important as configurable architectures proliferate. An adaptive weighted window (AWW) algorithm achieves excellent results for a wide range of scenarios, including various application sequence patterns (even the bad case of random sequences), and situations of low or high reconfiguration times. AWW involves a straightforward implementation, and outperforms the well-known Work Function online algorithm, while also having the lower time complexity of $O(m)$ rather than $O(m^2)$ (explainable by our different goal of achieving good results for realistic scenarios versus theoretical worst case results). If m, the number of possible configurations, is large, subsetting could possibly reduce m with minimal performance loss [14]. AWW improves over a non-dynamic approach by 6% on average, and up to 30% for small reconfiguration times.

Future work may consider variability in application and configuration times, energy costs, preemptable applications, and other extensions.

To facilitate reproduction, comparison, and extension of this work, the complete datasets used in this paper are available at http://www.cs.ucr.edu/~vahid for an indefinite period of time.

7. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation (CNS-0614957).

REFERENCES

[1] D.H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. Journal of Instruction Level. Parallelism, May 2000.

[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor. Int. Symp. on Microarchitecture (MICRO), 2000, pp 245-257.

[3] Y. Bartal, A. Blum, C. Burch, and A. Tomkins. A polylog(n)-competitive algorithm for metrical task systems. ACM Symp. on Theory of Computing, 1997, pp. 711-719.

[4] A. Blum and C. Burch. On-line learning and the metrical task system problem. Journal of Machine Learning, Vol. 39, No. 1, April 2000, pp. 35-58.

[5] A. Borodin, N. Linial, and M.E. Saks. An optimal on-line algorithm for metrical task system. Journal of the ACM (JACM), Volume 39, Issue 4 (Oct. 1992), pp. 745 – 763.

[6] W.R. Burley and S. Irani. On algorithm design for metrical task system. Algorithmica, 1997, Vol. 18, pp. 461-485.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to algorithms. MIT Press, 2001.

[8] A. Gordon-Ross and F. Vahid. A self-tuning configurable cache. Design Automation Conference (DAC), 2007, pp. 234-237.

[9] R. Kumar, D. Tullsen, N. Jouppi, P. Ranganathan. Heterogenous chip multiprocessors. IEEE Computer, Nov. 2005, pp. 32-38.

[10] K. Lahiri , A. Raghunathan, G. Lakshminarayana, and S. Dey. Communication architecture tuners: a methodology for the design of high-performance communication architectures for systems-on-chips. ACM/IEEE Design Automation Conf. (DAC), 2000, pp. 513-518.

[11] A. Malik, B. Moyer and D. Cermak. A low power unified cache architecture providing power and performance flexibility. Int. Symp. on Low Power Electronics and Design (ISLPED), June 2000.

[12] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, D.M. Tullsen. Application-specific customization of parameterized FPGA soft-core processors. IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Nov. 2006, pp. 261-268.

[13] K. Sekar, K. Lahiri, S. Dey. Dynamic platform management for configurable platform-based system-on-chips. Int. Conf. on Computer Aided Design (ICCAD), 2003, pp. 641-648.

[14] P. Viana, A. Gordon-Ross, E. Keogh, E. Barros, F. Vahid, Configurable cache subsetting for fast cache tuning. IEEE/ACM Design Automation Conference (DAC), July 2006, pp. 695 - 700.

[15] C. Zhang, F. Vahid and W. Najjar. A highly-configurable cache architecture for embedded systems. International Symposium on Computer Architecture (ISCA), 2003, pp. 136-146.